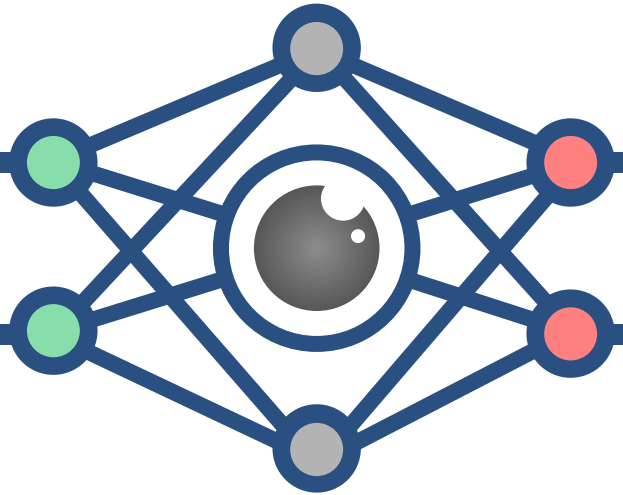
CS3485

Deep Learning for Computer Vision



Lec 8: Transfer Learning and Residual Nets

Announcements

- Change office hours? Email me if needed.
- Lab2 was graded lest week. Let me know if you have questions or complaints ;)
- A little info about the exams (more on it next week).

(Tentative) Lecture Roadmap

Basics of Deep Learning

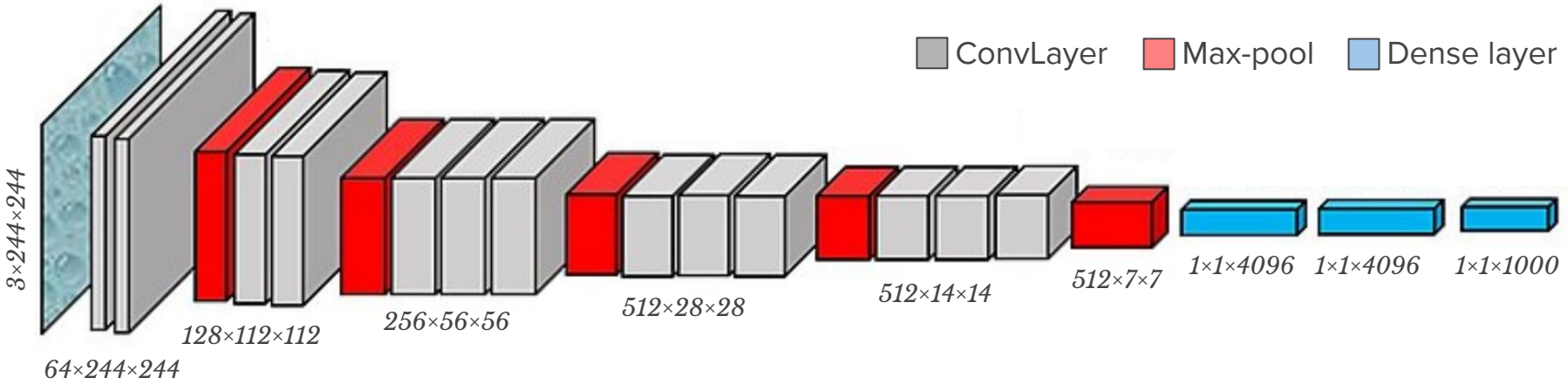
- Intro to Computer Vision
- Linear Classifiers and Perceptron
- Multilayer Perceptron and Intro to Deep Learning
- Optimization and Regularization
- Pytorch
- Convolutional Neural Networks
- Data Augmentation and Deep CNNs
- Transfer Learning and Residual Nets
- Inception Net and what CNNs learn
- Adversarial Examples and Self-supervision

Computer Vision Tasks

- Intro to Object Detection
- Fast Object Detection
- Intro to Image Segmentation
- Applications of Detection and Segmentation
- Autoencoders
- Image Generation with GANs
- Advanced GANs
- Transformers and ChatGPT
- Image Generation by Prompt

Recap: VGG16 network

- Last time, we saw the VGG16 network that produced incredible results on the ImageNet challenge using a somewhat simple **architecture**.
- Despite its simplicity, this network is very slow to train (the original VGG model was trained in *2-3* weeks), mainly because of its sheer amount of parameters: *135* million!

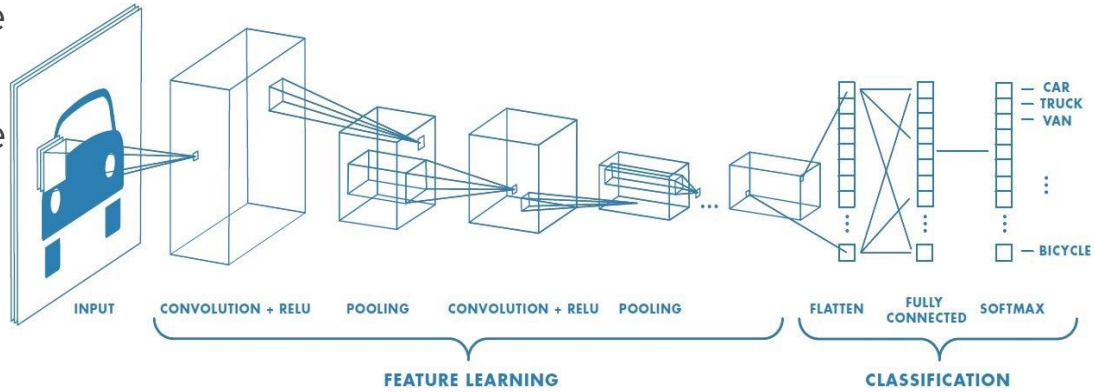


Deep Nets in practice

- Beyond the size problem found in the VGG16 architecture, it also faces what is called the **Vanishing Gradient problem**, which makes its training slower.
- Today, we'll see how make the deep networks useful in practice for computationally low budget applications using of a technique called **transfer learning**, which takes advantages of pre trained neural networks.
- We'll also see how to overcome the Vanishing Gradient problem via what are called **Residual Networks**.

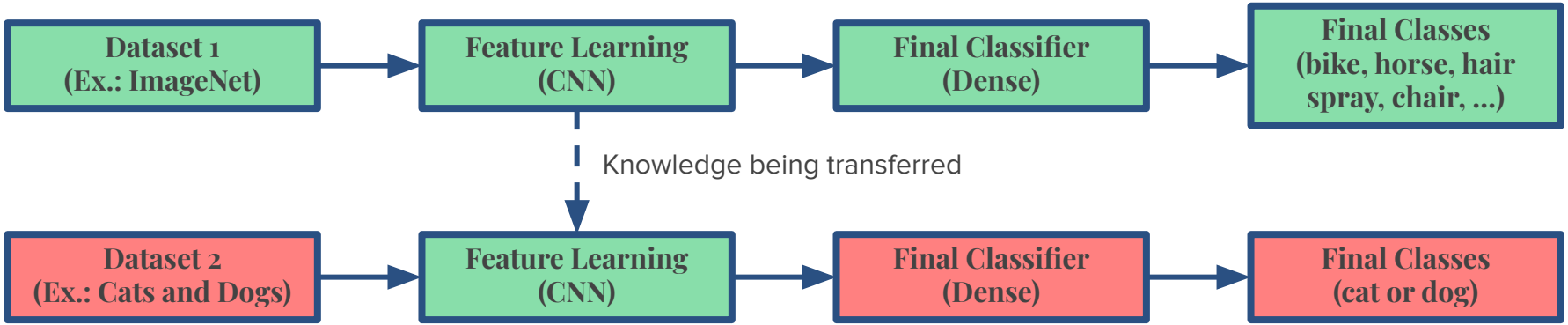
Transfer Learning

- **Transfer learning** is a technique where knowledge gained from one task is leveraged to solve another similar task.
- Specifically to the image classification task, we can make use of the visual features learned by one network trained on some dataset and **tune it** so that learned feature extractor can be used in our specific dataset.
- Put in another way, we reuse the **pre-trained weights** from a network on its feature learning phase and only replace the final dense layers so the final model fits our data.



Transfer Learning

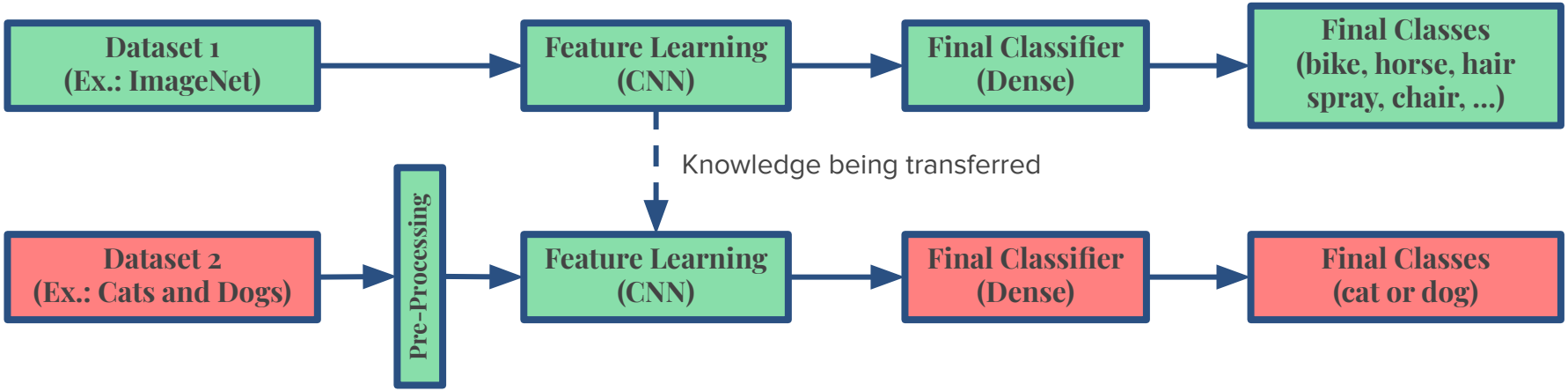
- An example of this would be like done below, where the knowledge of what was learned by a CNN while trained on given dataset (like ImageNet) is transferred to another problem that does not involve the same kinds of images from the first dataset.



- From this perspective, the main purpose of the final dense layers on the second problem is to adapt that knowledge (the CNN output) to the new classes. This process is called **fine tuning**.

Transfer Learning: Preprocessing

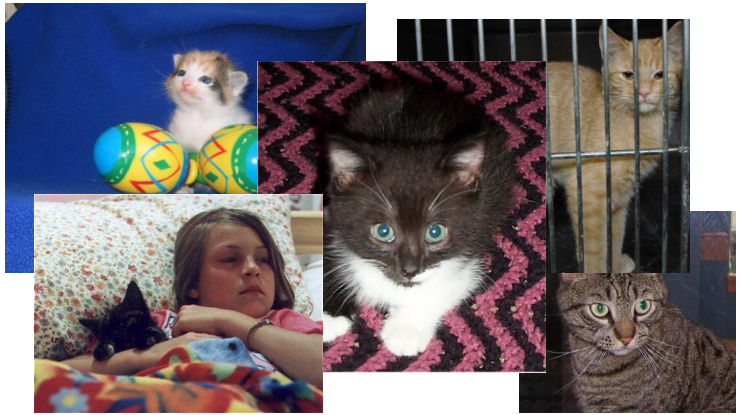
- In order for transfer learning to work in its full potential, some preprocessing of the new dataset has to be done before it enters in the pre-trained network.
- This ensures that the data coming follows **the same format** of the data it was trained on.



- In the case of the VGG net, we need the images to RGB of size 244×244. We also normalize them to follow the same mean and std from the ImageNet data.

Cats vs. Dogs Dataset

- To try out transfer learning, we'll use a famous dataset called **Cats vs. Dogs**.
- The dataset contains 12491 and 12470 photos of various sizes of cats and dogs, respectively, under various challenging settings.



- It is organized in two folders (test and train data), each with a `cat` and a `dog` subfolder.
- Our objective is to use VGG16 trained on ImageNet to learn a classifier for this dataset.

Cats vs. Dogs dataset class

- Since we are not using one of PyTorch's datasets, our Dataset class is more complicated:

```
from glob import glob
import torchvision.transforms as transforms
import torchvision.io as io

class CatsDogs(Dataset):
   def   init  (self, folder):
       cats, dogs = glob(folder+'/cats/*.jpg'), glob(folder+'/dogs/*.jpg')
       self.img paths = cats[:500] + dogs[:500]
       self.normalize = transforms.Normalize( mean=[0.485, 0.456, 0.406],
                                              std=[0.229, 0.224, 0.225])
       self.resize = transforms.Resize(( 224, 224))
       self.labels = [fpath.split( '/')[-1].startswith('dog')
                      for fpath in self.img_paths]
   def   len  (self):
       return len(self.img paths)
   def   getitem  (self, ix):
       img = io.read image( self.img paths[ix])
       label = torch.tensor([ self.labels[ix]])
       img =  self.resize(img/255.)
       img =  self.normalize(img)
       return img.float().to(device) , label.float().to(device)
```

Cats vs. Dogs dataset class

- Since we are not using one of PyTorch's datasets, our Dataset class is more complicated:

```
from glob import glob
import torchvision.transforms as transforms
import torchvision.io as io

class CatsDogs(Dataset):
    def __init__(self, folder):
        cats, dogs = glob(folder+'/cats/*.jpg'), glob(folder+'/dogs/*.jpg')
        self.img_paths = cats[:500] + dogs[:500]
        self.normalize = transforms.Normalize( mean=[0.485, 0.456, 0.406],
                                               std=[0.229, 0.224, 0.225])
        self.resize = transforms.Resize(( 224, 224))
        self.labels = [fpath.split('/')[-1].startswith('dog')
                       for fpath in self.img_paths]
    def __len__(self):
        return len(self.img_paths)
    def __getitem__(self, ix):
        img = io.read_image( self.img_paths[ix])
        label = torch.tensor([ self.labels[ix]])
        img = self.resize(img/255.)
        img = self.normalize(img)
        return img.float().to(device) , label.float().to(device)
```

Fetch for all image paths under each folder using the library `glob`.

Only use the first 500 images of each class, for a total of 1000.

We'll use this function to change the mean and standard deviation of each channel in each datapoint to match those statistics from ImageNet

We'll also use this function to resize the images to the size originally used by VGG16.

This creates the vector of labels according to which folder each file is located in.

Cats vs. Dogs dataset class

- Since we are not using one of PyTorch's datasets, our Dataset class is more complicated:

```
from glob import glob
import torchvision.transforms as transforms
import torchvision.io as io

class CatsDogs(Dataset):
    def __init__(self, folder):
        cats, dogs = glob(folder+'/cats/*.jpg'), glob(folder+'/dogs/*.jpg')
        self.img_paths = cats[:500] + dogs[:500]
        self.normalize = transforms.Normalize( mean=[0.485, 0.456, 0.406],
                                               std=[0.229, 0.224, 0.225])
        self.resize = transforms.Resize(( 224, 224))
        self.labels = [fpath.split( '/')[-1].startswith('dog')
                       for fpath in self.img_paths]
    def __len__(self):
        return len(self.img_paths)
    def __getitem__(self, ix):
        img = io.read_image( self.img_paths[ix])
        label = torch.tensor([ self.labels[ix]])
        img = self.resize(img/255.)
        img = self.normalize(img)
        return img.float().to(device) , label.float().to(device)
```

Reads the image in a given path

Turns the image label into a tensor

Resizes and normalizes the image

Cats vs. Dogs dataset and VGG16 model

- Now, say that you divided the dataset in a folder for training data and another for testing. You can then create the train and test data loaders as follows:

```
train_data_dir, test_data_dir = 'training_set',  'test_set'

train = CatsDogs(train_data_dir)
test = CatsDogs(test_data_dir)
trn dl = DataLoader(train, batch size=32, shuffle=True)
test_dl = DataLoader(test, batch_size=32, shuffle=True)
```

- With the data done, go to the creation of our model. We'll use the VGG16 network with pretrained weights when it was trained on the ImageNet dataset.
- In PyTorch, we specify this requirement by setting the parameter `weights` on `models.vgg16` to the set of weights we are want, in this case `IMAGENET1K_V1`.

```
model = models.vgg16(weights=models.VGG16_Weights.IMAGENET1K_V1)
```

Checking the model

- As per good practice, we print the model to check what it provides us:

```
print(model)
```

```
VGG(
  (features): Sequential(
       (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
       (1): ReLU(inplace=True)
       (...)
       (29): ReLU(inplace=True)
       (30): 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)
  )
)
```

Changing the model

- The model has three submodels: `features`, `avgpool` and `classifier`. Our goal is:
  a. make the `features` module's weights fixed,
  b. adapt the last two modules to our problem and learn their weights.
- For a, we just need to make the parameters of the `features` module **not learnable** by setting their `requires_grad` option to false:

```
for param in model.features.parameters():
    param.requires_grad = False
```

- For b, we replace VGG16's original AvgPool layer with one that returns a tensor of size $512 \times 1 \times 1$ and replace its classifier layer with one that outputs a simple scalar (which is enough for us, since our dataset only has two classes)

```
model.avgpool = nn.AdaptiveAvgPool2d(output size=(1, 1))
model.classifier = nn.Sequential(nn.Flatten(),
                                 nn.Linear(512, 128), nn.ReLU(), nn.Dropout(0.2),
                                 nn.Linear(128, 1), nn.Sigmoid())
```

Changing the model

- The model has three submodels: `features`, `avgpool` and `classifier`. Our goal is:
  a. make the `features` module's weights fixed,
  b. adapt the last two modules to our problem and learn their weights.
- For a, we just need to make the parameters of the `features` module **not learnable** by setting their `requires_grad` option to false:

```
for param in model.features.parameters():
    param.requires_grad = False
```

- For b, we replace VGG16's original AvgPool layer with one that returns a tensor of size $512 \times 1 \times 1$ and replace its classifier layer with one that outputs a simple scalar (which is enough for us, since our dataset only has two classes)

```
model.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
model.classifier = nn.Sequential(nn.Flatten(),
                                 nn.Linear(512, 128), nn.ReLU(), nn.Dropout(0.2),
                                 nn.Linear(128, 1), nn.Sigmoid())
```

Note that we have to add the sigmoid at the end, because of the loss we'll use this time (BCE).

Model Summary

- We can now check the summary of our new VGG16-based model:

```
from torchsummary import summary
summary(model.to(device),(1,28,28))
```

- Note the the new model has only 65k parameter to be learned compared to the original 135 million from VGG16.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 244, 244]           1,792
              ReLU-2         [-1, 64, 244, 244]               0
            Conv2d-3         [-1, 64, 244, 244]          36,928
              ReLU-4         [-1, 64, 244, 244]               0
         MaxPool2d-5         [-1, 64, 122, 122]               0
               (...)                      (...)             (...)
             ReLU-30        [-1, 512, 15, 15]               0
        MaxPool2d-31          [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-32          [-1, 512, 1, 1]               0
          Flatten-33                [-1, 512]               0
           Linear-34                [-1, 128]          65,664
             ReLU-35                [-1, 128]               0
          Dropout-36                [-1, 128]               0
           Linear-37                  [-1, 1]             129
          Sigmoid-38                  [-1, 1]               0
================================================================
Total params: 14,780,481
Trainable params: 65,793
Non-trainable params: 14,714,688
----------------------------------------------------------------(...)
```

Loss, optimizer and auxiliary functions

- This time, since we only have two classes (cats and dogs) we'll use the Binary Cross-Entropy loss (BCE), instead our usual Cross Entropy (CE):

```
loss_fn = nn.BCELoss()
```

BCE is handy since we **only need one scalar output**, as with CE we'd need two, since we have two classes.

- We then just reuse the same optimizer and the auxiliary functions as usual:

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    opt.zero_grad() # Flush memory
    batch_loss = loss_fn(model(x), y) # Loss
    batch_loss.backward() # Compute gradients
    opt.step() # Make a GD step
    return batch_loss.detach().cpu().numpy()
```

```
@torch.no_grad()
def accuracy(x, y, model):
    model.eval()
    prediction = model(x)
    pred_class = (prediction > 0.5)
    s = torch.sum((pred_class == y).float()) / len(y)
    return s.cpu().numpy()
```

Train the model

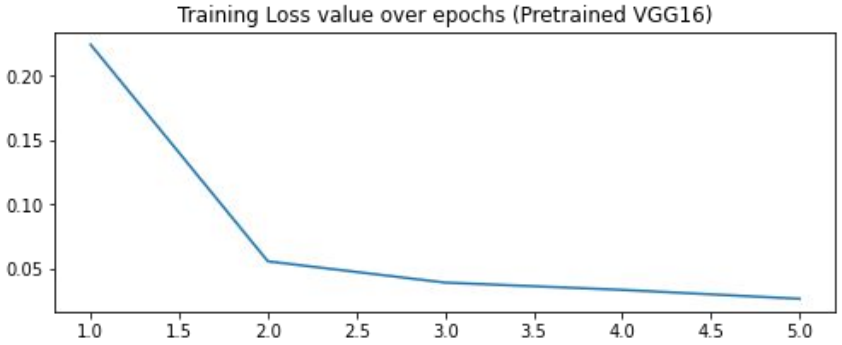
- As before we train the model in a `for` loop for the required number of epochs:

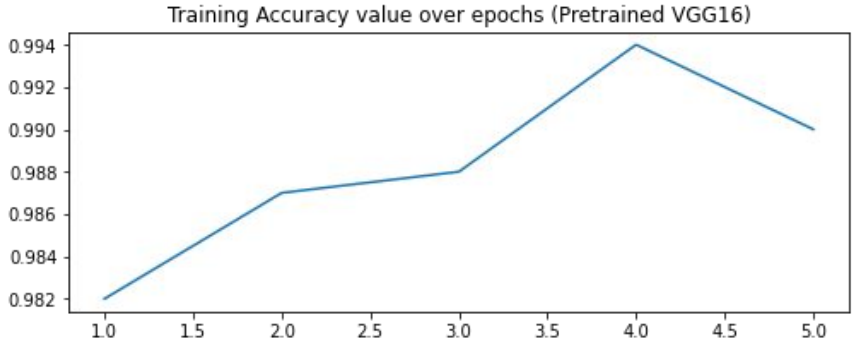
```
import numpy as np
train_losses, train_accuracies, n_epochs = [], [], 5
for epoch in range(n_epochs):
   print(f"Running epoch {epoch + 1} of {n_epochs}")
   train_epoch_losses, train_epoch_accuracies = [], []

   train_epoch_losses = [train_batch(x, y, model, optimizer, loss_fn) for x, y in trn_dl]
   train_epoch_loss = np.mean(train_epoch_losses)

   train_epoch_accuracies = [accuracy(x, y, model) for x, y in trn_dl]
   train_epoch_accuracy = np.mean(train_epoch_accuracies)

   train_losses.append(train_epoch_loss)
   train_accuracies.append(train_epoch_accuracy)
```

Note that, for simplicity, we are using Python's *list comprehension* for training each each batch in each epoch. See previous lectures for a more detailed loop.

Results and timing

- Plotting the training loss and accuracy we get:





- After that, we can then check the test accuracy:

```
epoch_accuracies = [accuracy(x, y, model) for x, y in test_dl]
print(f"Test accuracy: {np.mean(epoch_accuracies)}")
```

```
Test accuracy: 0.9833984375
```

Comparison to a CNN from scratch

- We could also compare this model with a somewhat deep network, trained from scratch on the **whole data set**.
- To create the dataset in an efficient way, we first create a function that returns a ConvLayer with batch normalization and max pooling:

```
def ConvLayer(in_ch, out_ch, kernel_size):
    return nn.Sequential(nn.Conv2d(in_ch, out_ch, kernel_size), nn.ReLU(),
                         nn.BatchNorm2d(out_ch),
                         nn.MaxPool2d(2))
```

- Then, we create our model based on 6 ConvLayers for the feature learning phase and 1 Dense layer for the final classifier (followed by a sigmoid as done before).

```
model = nn.Sequential(ConvLayer(3, 64, 3),
                      ConvLayer(64, 512, 3),
                      ConvLayer(512, 512, 3),
                      ConvLayer(512, 512, 3),
                      ConvLayer(512, 512, 3),
                      ConvLayer(512, 512, 3),
                      nn.Flatten(),
                      nn.Linear(512, 1), nn.Sigmoid())
```

Model parameters and results.

- Printing the summary of the model would show that we need to learn around 10 million parameters.
- Since we are also using all the ~25k $3\times244\times244$ images in the dataset, we'd need a lot of time to train this model.
- For comparison, in FMNIST, we needed around 1 min to train a model of 800k weight on a dataset of ~60k $1\times28\times28$ images.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 222, 222]           1,792
              ReLU-2         [-1, 64, 222, 222]               0
       BatchNorm2d-3         [-1, 64, 222, 222]             128
         MaxPool2d-4         [-1, 64, 111, 111]               0
            Conv2d-5        [-1, 512, 109, 109]         295,424
              ReLU-6        [-1, 512, 109, 109]               0
       BatchNorm2d-7        [-1, 512, 109, 109]           1,024
         MaxPool2d-8          [-1, 512, 54, 54]               0
             (...)                      (...)             (...)
      BatchNorm2d-23            [-1, 512, 3, 3]           1,024
        MaxPool2d-24            [-1, 512, 1, 1]               0
          Flatten-25                  [-1, 512]               0
           Linear-26                    [-1, 1]             513
          Sigmoid-27                    [-1, 1]               0
================================================================
Total params: 9,742,209
Trainable params: 9,742,209
Non-trainable params: 0
----------------------------------------------------------------
```

- The test classification accuracy in 5 epochs is, however, not impressive at ~86%.

Training Model from Scratch vs. Pretrained Model

- **The performance difference is impressive between the two (98% vs. 86%).**
- Despite having so few parameters to learn, the pretrained VGG model adapted to our problem is vastly superior, which brings evidence to **the power of transfer learning**.
- It also showed how we could leverage the information learned from VGG on ImageNet to our problem without needing so many data points/images.
- The reasons why the model trained from scratch was so underperforming can be many, such as:
  a. It didn't train long enough,
  b. The choice of hyperparameter could be improved,
  c. It needed more data to capture the nuances of the difference between cats and dogs.
  d. It suffered from the vanishing gradient problem.
- The problem in d is typical of deep nets, and we'll learn how to tackle it later today.

Exercise (*In pairs*)

Click here to open code in Colab

- The VGG16 network was trained on RBG images. Say we need to learn on dataset of grayscale images, and we'd like to use transfer learning from VGG16 trained on ImageNet. What should we do?
- Say we decided to add a ConvLayer of 1 input channel and 3 output channels **at the beginning** of VGG16 and keep all the other ConvLayers' weights not learnable. Then we just that new layer's and the dense layers' weights. Would this approach work for us?

The Vanishing Gradient problem

- VGG16 (and VGG19) is so good! Why can't we go VGG25, VGG50 or VGG100?
- Unfortunately, training very deep networks in the VGG style is very hard due to what is called the **Vanishing Gradients**, i.e.: the gradient **exponentially decreases and approaches 0 as more layers are being multiplied** during backpropagation.
- Why does that happen? As we did in a previously, consider a network of one fully connected hidden layer and the cross entropy loss on only one datapoint x with label y:

$$u_0(W_0) = l(NN_\theta(x), y) = -y^\top \log(\text{softmax}(W_1 a(W_0 x)))$$

- Again, let $u_1(z) = -y^\top z$, $u_2(z) = log(z)$, $u_3(z) = \text{softmax}(z)$, $u_4(z) = W_1 z$, $u_5(z) = a(z)$, $u_6(z) = z^\top x$. Then have that $u_0 = u_1(u_2(u_3(u_4(u_5(u_6(W_0))))))$ and that:

$$\frac{du_0}{dW_0} = \frac{du_0}{du_1}\frac{du_1}{du_2}\frac{du_2}{du_3}\frac{du_3}{du_4}\frac{du_4}{du_5}\frac{du_5}{du_6}\frac{du_6}{dW_0}$$

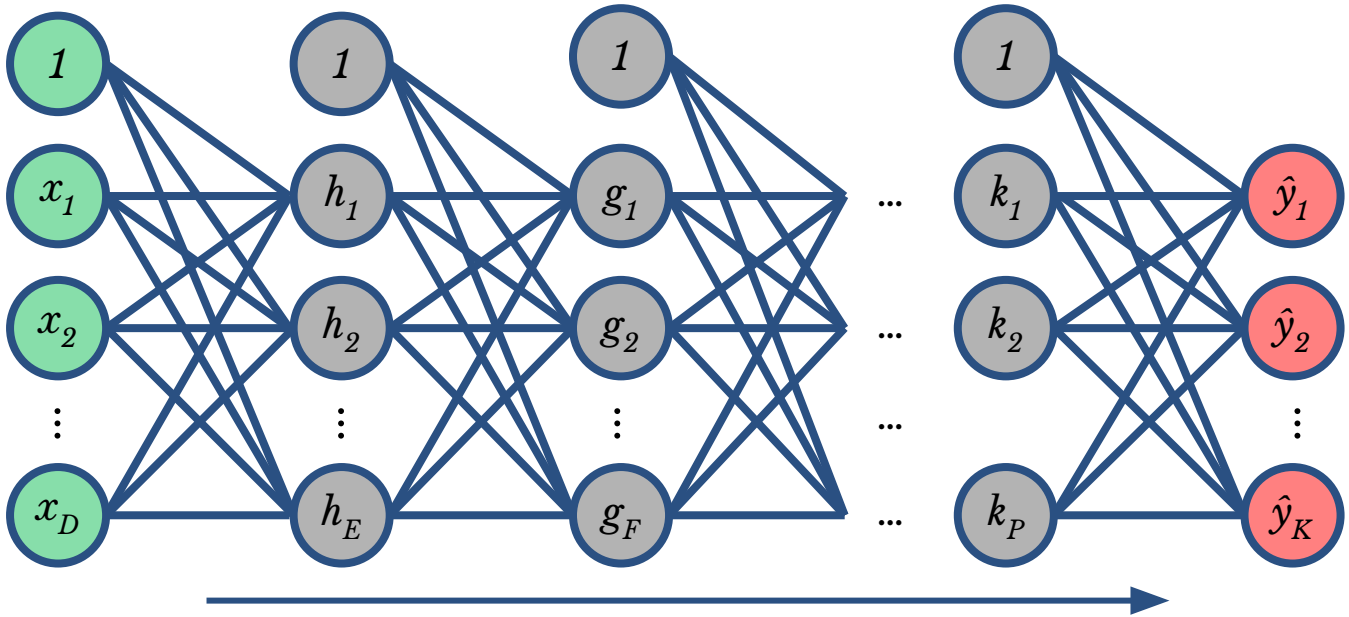
The Vanishing Gradient problem

- That means that in order to find the update for the weights of the first layer, one has to proceed with 6 multiplications due to the chain rule. This implies that:
  - With more layers, we'd need **more multiplications** to get the first layer's weights updated.
  - If many of these multiplications are with numbers (absolute values are lesser than 1*), we'd have a final product **that is very small****.
- *Example*: if we had 15 multiplications whose terms are of absolute value of around 0.7, we'd have a gradient ound $(0.7)^{15} \approx 0.005$.
- Note that this affects more **the weights on initial layers** than the final ones, which makes learning the latter difficult, and hinders the performance of very deep networks.
- **Batch Normalization** and **ReLU activations** help solve the Vanishing Gradient problem, but we can can also change the network architecture to improve learning.

* The vanishing gradient is a big problem for networks with **sigmoid activations**, since its derivative is always < 1. ReLU's are also useful because they don't have this property.
** We can also encounter the **Exploding Gradient** issue, when many of these numbers are **larger** than 1.

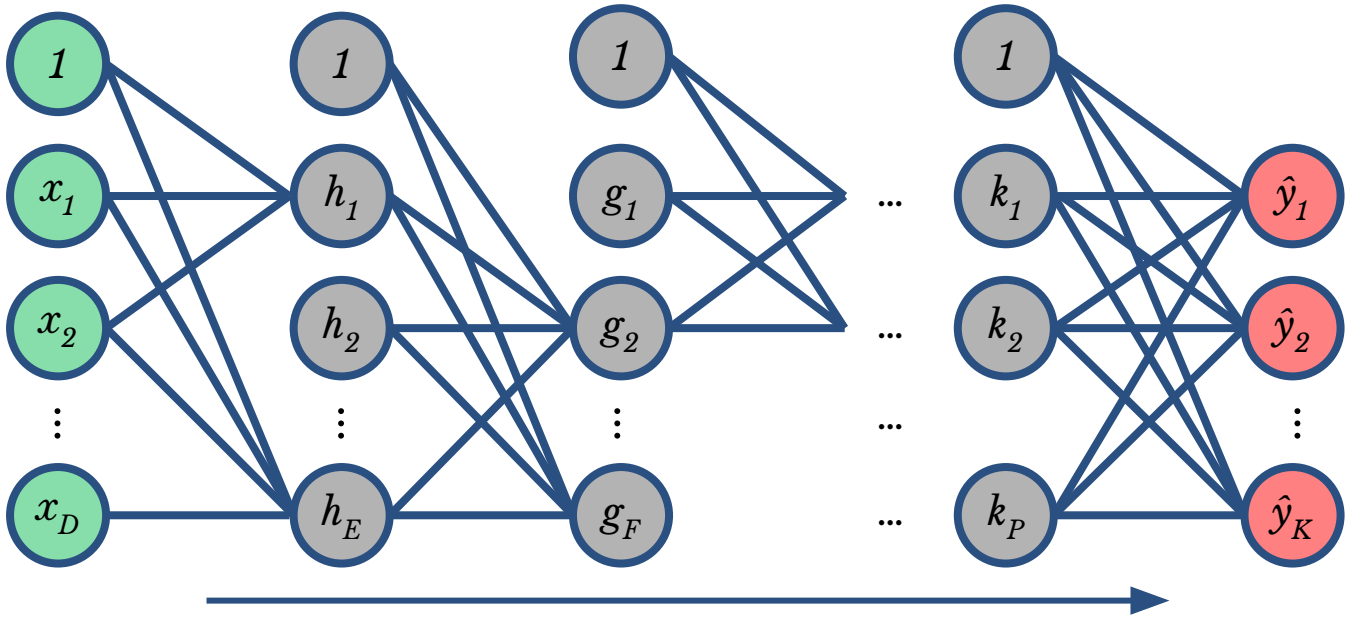
Feed Forward Networks

- All the networks we've seen so far have two things in common: *(1)* the data moves in only one direction and *(2)* there is only one possible sequence of layer to do so.



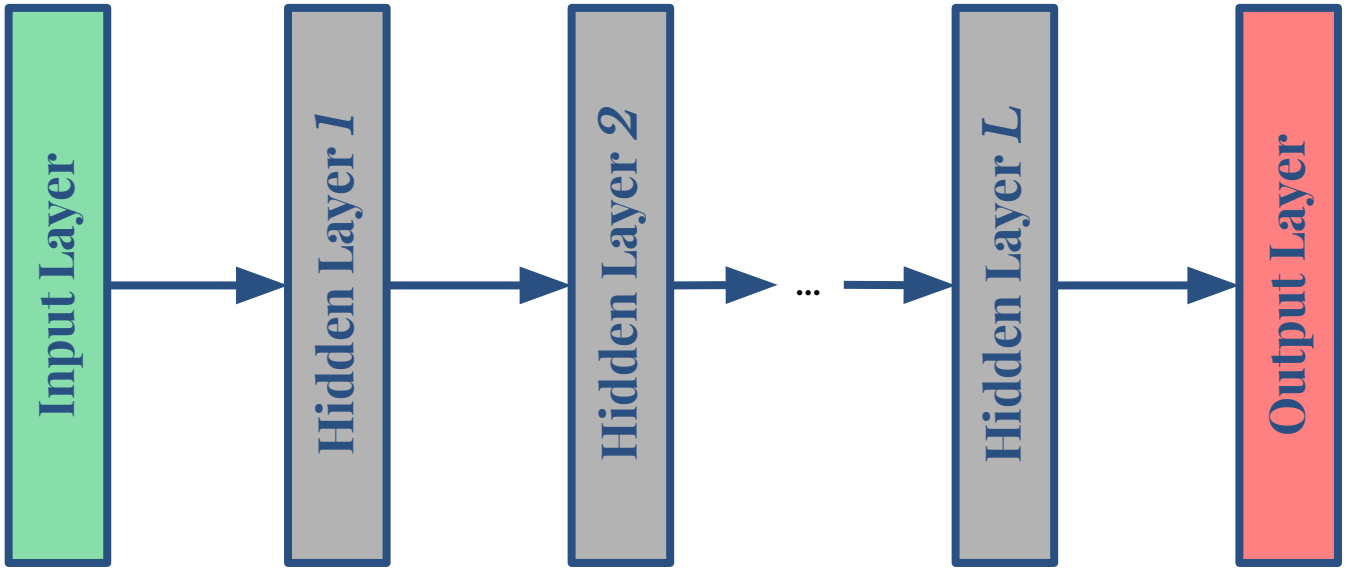
Feed Forward Networks

- This the case of both the Multilayer Perceptron (*last slide*) and the Convolutional Neural Network (*below*), which can be seen as just a sparser version of the MLP.



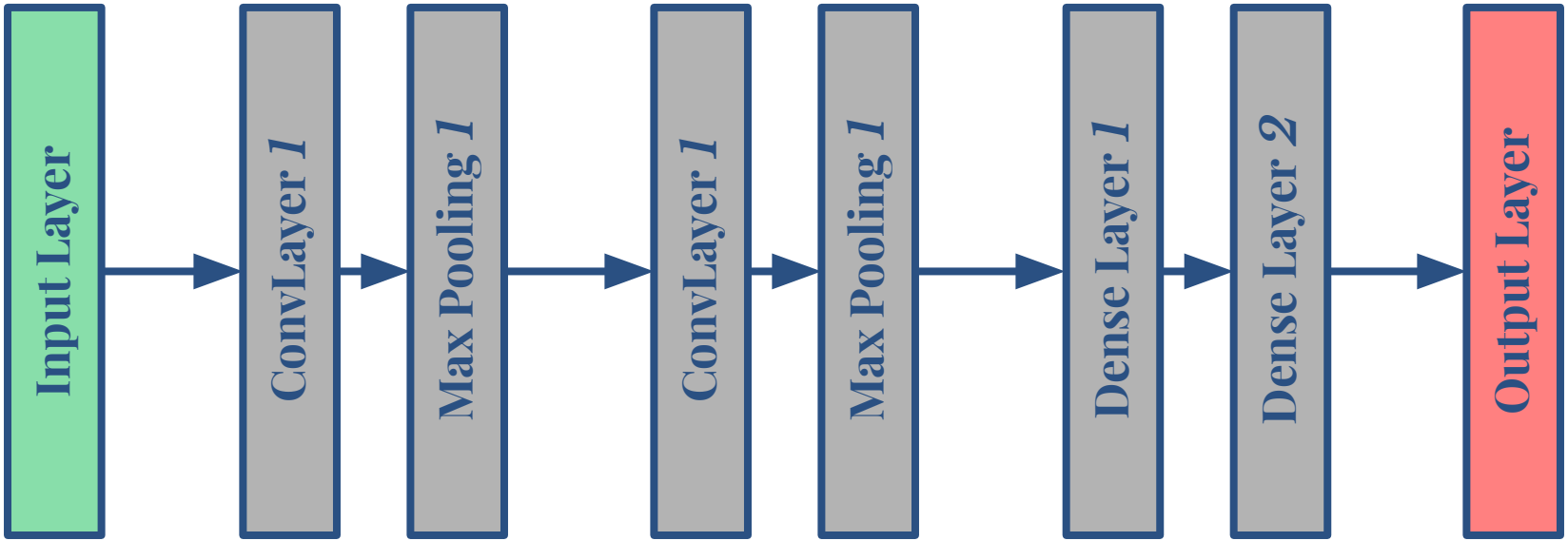
Feed Forward Networks

- The fact that the data moves in one direction makes it a **Feed Forward** network and if there is one sequence of layers it means that it doesn't have **skip connections**.



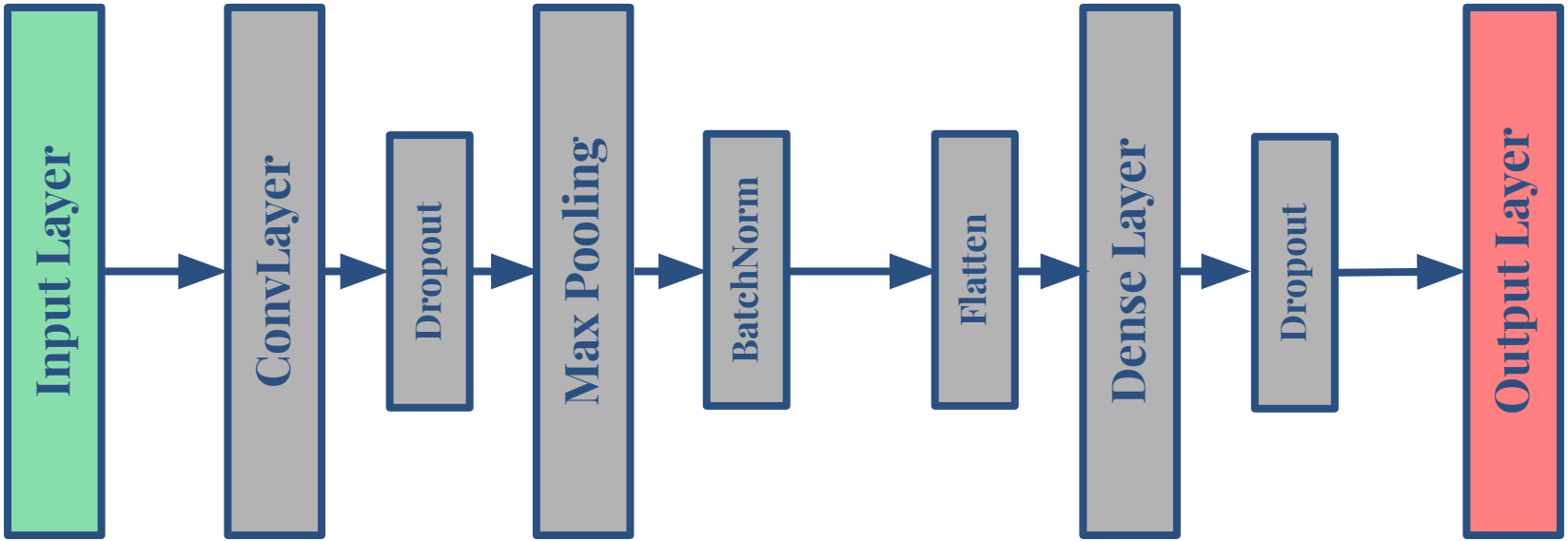
Feed Forward Networks

- The networks with pooling layers or a mix of convolutional and linear (also called dense) layers we've seen so far are still Feed Forward.



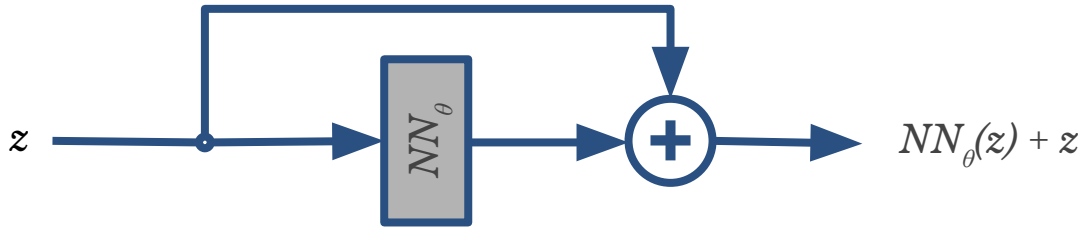
Feed Forward Networks

- Adding features such as Dropout, Batch Normalization and a flattening layer doesn't change the network dataflow.



Residual Neural Networks

- While ReLU and Batch Normalization help solve the vanishing gradient problem, we can also use residual layers to improve our solution.
- A **residual layer** has the following operation, where NN_θ represents a layer, a sequence of layers or an entire network:

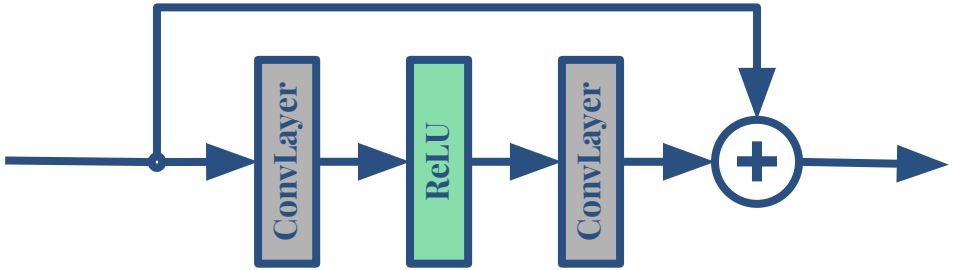


where one has to be careful that the output size of NN_θ **should be the same as its input**.

- The link between the input and the output that skips NN_θ is called a **skip connection**.
- Note that this network allows the information to flow in **two possible directions,** which is not possible in the networks we've seen so far.

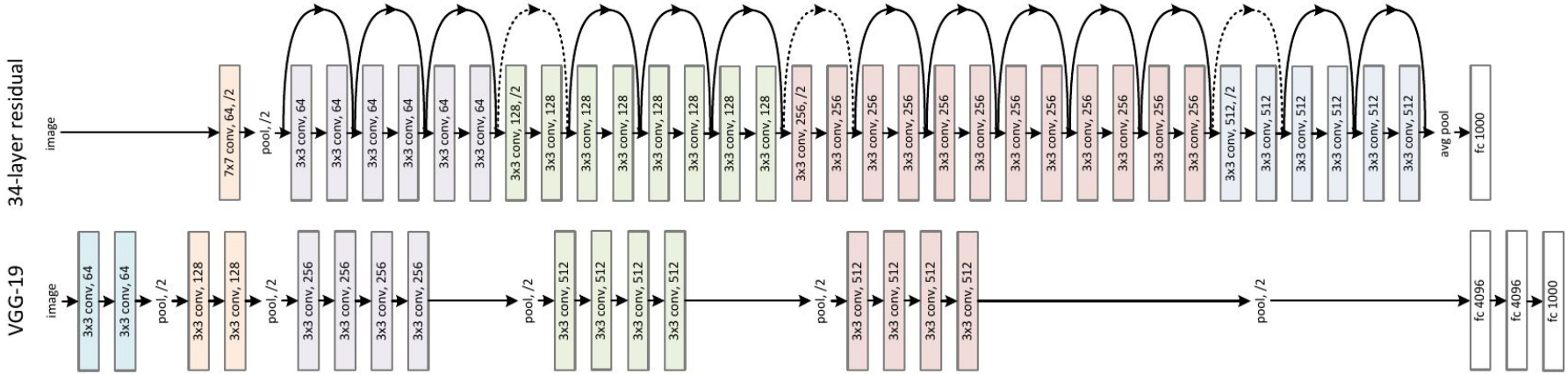
Residual Neural Networks

- Note that $d(NN_\theta(z) + z)/dW_0 = dNN_\theta(z)/dW_0 + dz/dW_0$, where W_0 is are the weights on the first layer of a network.
- *That is*: if the derivative gets diminished when crossing NN_θ, it doesn't get very small as it gets bumped up by the prior derivatives (represented in dz/dW_0).
- This insight led to the development of **Deep Residual Networks** (ResNets), published 2015, which attained a 5.25%, 4.60% and 4.49% classification error rates on ImageNet with networks of 50, 101 **and** 152 **layers**, respectively!
- All these architectures had the following as their main building block:



VGG vs. ResNet

- In the paper that introduced ResNets, the authors also made a comparison between a residual network with 34 layers and VGG19:



- Note that they only used one fully connected layer for the classification step.
- That made the number of parameters in their network be 5x smaller than VGG in this case, also illustrating how good their feature learning phase was.

ResNets in PyTorch

- ResNets are a great example of why PyTorch is very easy when writing more complex networks. To create a residual layer block from ResNet, we can simply write:

```
import torch
import numpy as np

class ResidualBlock(nn.Module):
    def   init  (self, in ch, kernel_size):
        super().  init  ()
        padding = kernel_size - int(np.ceil(kernel_size / 2))    # Adding padding to keep image sizes the
                                                                 # before and after the convolutions
        self.ConvLayer0 = nn.Conv2d(in_ch, in_ch, kernel_size,
                                    padding=padding)             # Note that the num. of channels in the
        self.ConvLayer1 = nn.Conv2d(in ch, in ch, kernel_size,   # input is the same as in the output.
                                    padding=padding)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.ConvLayer1(self.relu(self.ConvLayer0(x))) + x   # Simply applies the residual formula
        return x
```

Pretrained ResNets

- Just as we did with VGG16, we can also use pre-trained ResNets. For example, for the ResNet50, which 50 layers, we use `models.resnet50()`:

```
model = models.resnet50(weights=models.ResNet50_Weights.IMAGENET1K_V1)
```

  where `models.ResNet50_Weights.IMAGENET1K_V1`are the same weights that attained 5.25% error rate on the ImageNet challenge.
- Similarly, we can also load ResNet101 and ResNet152, whose summaries are:

```
summary(models.resnet101().to(device),(3,224,224))
```

```
(...)==========================================================
Total params: 44,549,160
Trainable params: 44,549,160
Non-trainable params: 0
----------------------------------------------------------(...)
```

```
summary(models.resnet152().to(device),(3,224,224))
```

```
(...)==========================================================
Total params: 60,192,808
Trainable params: 60,192,808
Non-trainable params: 0
----------------------------------------------------------(...)
```

- Note: ResNets use many more layers, but fewer weights than VGG16 with ~ 135 mi.

Exercise (*in pairs*)

- When defining the Residual Layer, we added the following line to keep image sizes the same:

```
padding = kernel_size - int(np.ceil(kernel_size / 2))
```

  Explain how this line is necessary for that goal and how it attains it. When does this solution break (*hint*: try out different kernel sizes) and explain why it breaks.
- Say you'd like to create a ResNet9, with an initial ConvLayer, followed by 3 Residual Layers. How would you do it using the code in here and `nn.Sequential`?