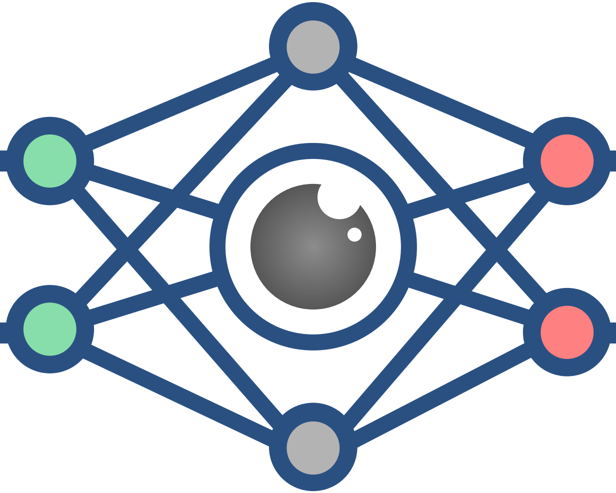
CS3485

Deep Learning for Computer Vision



Lec 8: Data Augmentation and Deep CNNs

Announcement

- Lab2:
  - Lab grades are out, let me know if you have any questions!
  - Don’t forget the default parameters!
- Quiz #3 later today!

(Tentative) Lecture Roadmap

Basics of Deep Learning

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

Deep Learning and Computer Vision in Practice

- Intro to MLOps
- Intro to Object Detection
- Fast Object Detection
- Intro to Image Segmentation
- Applications of Detection and Segmentation
- Autoencoders
- Image Generation with GANs
- Advanced GANs
- The Attention Mechanism
- Transformers and ChatGPT
- Image Generation by Prompt
- Misc. Topics

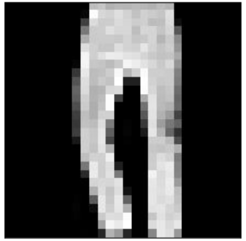
Ways to improve

- Last time, we saw that we can improve the classification task in the FashionMNIST dataset by using **Convolutional Neural Networks**.
- Despite our final classification outcome being pretty good, we can still improve it in some ways that we haven't tried last time:
    - By adding **regularization** (dropout, for example) and **Batch Normalization** to the network.
    - By training the network for **longer** (more than 5 epochs).
    - By tuning some of the network constants (also called **hyperparameters**), such as the optimizer's learning rate, the batch size, the number of strides and the padding of each ConvLayer.
    - By trying different amount of units/filters per layer to be learned.
    - By using **data augmentation**.
    - By adding **more layers** and making the network able to learn more complex image features.
- Today, we'll focus our efforts on the last two options: we'll see how making the **data (the input)** or the **network (the model)** "richer" can improve our classification performance.

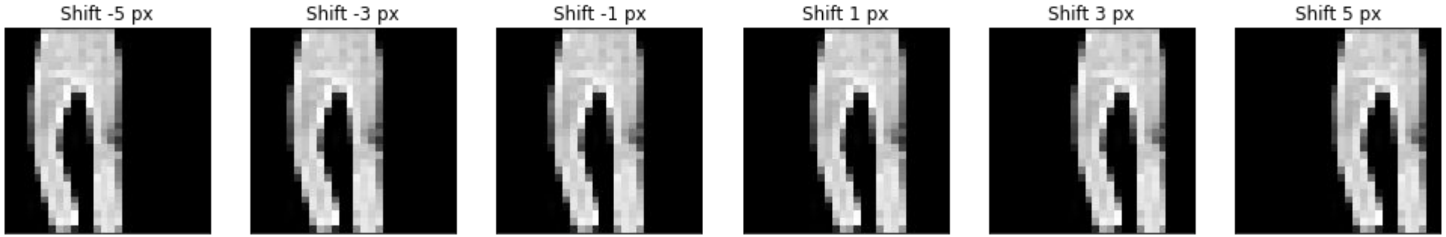
Issues with Shifting

- Last time, we saw that CNNs do much better at classifying Fashion MNIST data than simple Multilayer Perceptrons.
- Today, we’ll check how well their classifier works when we slightly change some of the images in a way that their classes would still be recognizable.
- This happens when you shift the image below some pixels to the right and to the left:

No shift



Various shifts



In these examples, the original class (“trouser”) shouldn’t become less recognizable because of the shifts.

Trying out the CNN on the shifted images

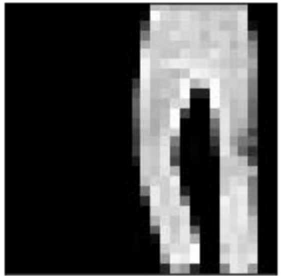
- Let's see how the model trained in the last class predicts the classes of the trouser shifted 1 to 5 pixels to the right and to the left*:

```
softmax = nn.Softmax() # Define the softmax function (remember that the more does not output probs.)
idx = 24300  # The index of the trouser from the last slide

preds = []
for ix in range(-5, 6):
   img = x_train[ix] # Read the desired image

   img_rolled = np.roll(img, px, axis=1) # Roll the image by "px" pixels

   img_rolled = torch.Tensor(img_rolled / 255.) \    # Scale and reshape the image to the image
                             .view(-1, 1, 28, 28) \ # format used during learning. Register the
                             .to(device)            # result to the GPU

   pred = softmax(model(img_rolled)) # Apply our learned model to predict the class probabilities

   preds.append(pred.cpu().detach().numpy()) # Post process the prediction and then save it to the list
```

* In the code above, we are using the variable names and libraries from the previous class. It's like its continuation.

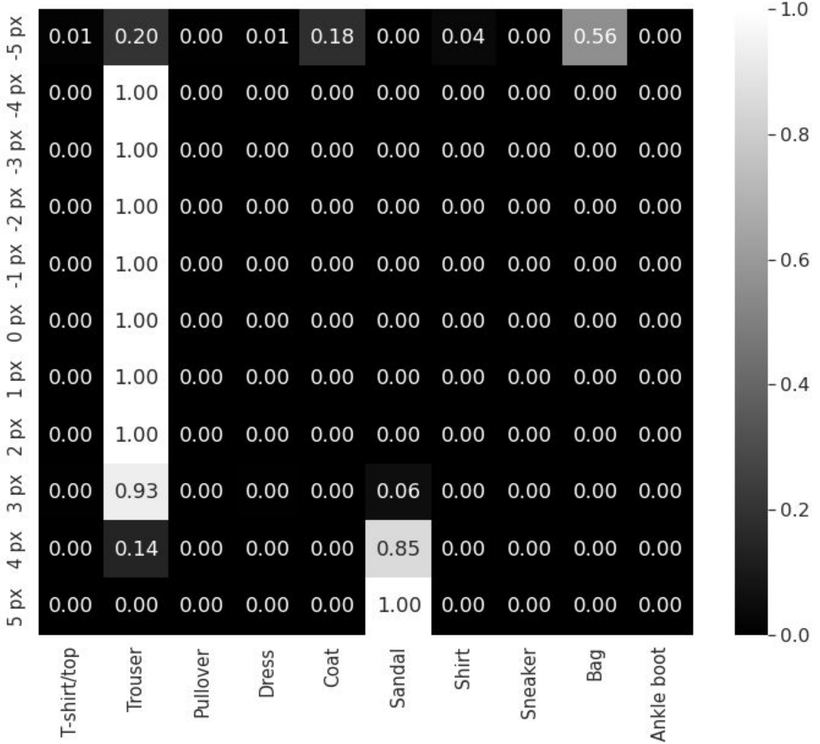
Trying out the CNN on the shifted images

- Now we can plot the probabilities of each shifted image to belong to each of the *10* possible classes.
- For most shifts, the network finds the right class “trouser”.
- But, unexpectedly, the network **makes very bad guesses** for the images shifted closer to the border of the image.



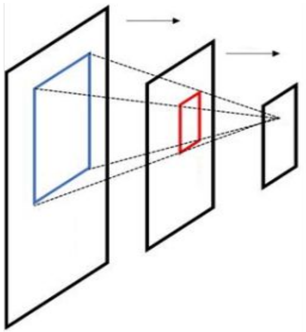
- In fact, **it seems to be sure** that the image on the left is a *sandal*!
- What can we do to fix this?

Prob. of each class for various shifts (CNN)



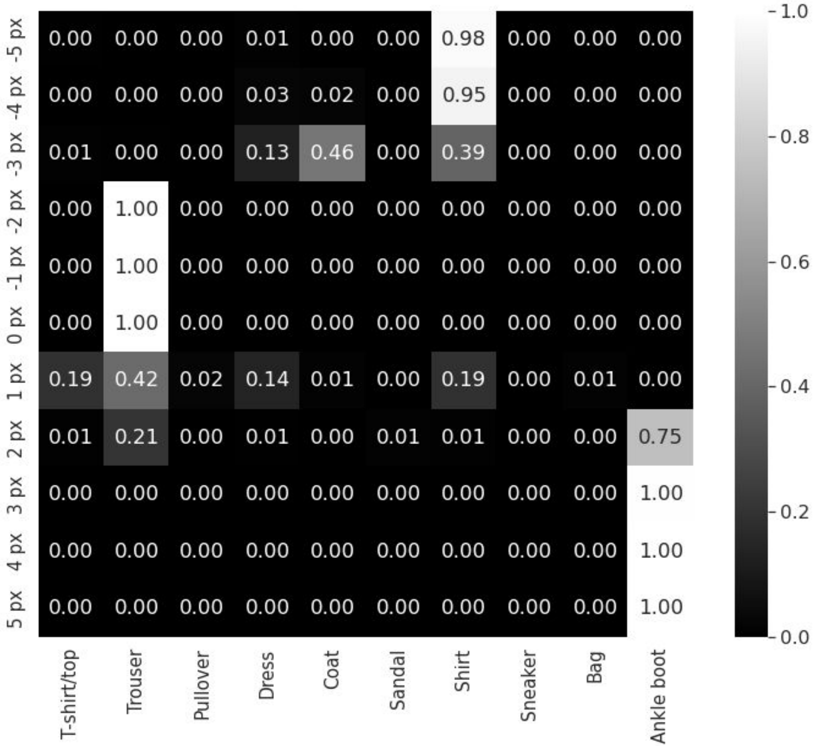
(*A digression*) CNNs and their receptive field

- To be fair with the CNN model, it does **quite a good work** when compared to the Multilayer Perceptron model (on the right).
- The improvement CNN adds to the pure fully connected MLP is related to the **receptive field** of convolutional and pooling layers.



- This means that later individual units have information about greater areas of the original image.
- This enables capturing of some shifting.

Prob. of each class for various shifts (MLP)



Data augmentation as a solution

- **But back to CNNs**! We noticed that these issues with image shifting can have on a model's prediction accuracy.
- However, in the real world, we might encounter various scenarios, such as the following:
  - Images are rotated slightly,
  - Images are zoomed in/out (scaled),
  - Some amount of noise is present in the image,
  - Images have low brightness,
  - Images have been flipped,
  - Images have been sheared (one side of the image is more twisted).
- A neural network that does not take the preceding scenarios into consideration won't provide accurate results.
- One solution to that issue is to **artificially change the data** in the dataset in a way to consider the above settings. This is called **Data Augmentation***.

* In other contexts, augmentation can also mean “make the dataset larger”, but in the end of the day, it is the same as we are doing.

Data augmentation via transformations

- The strategy we'll take consists in making random changes in each of our datapoints before they enter in our `train_batch` function.
- We'll use the very handy `transforms` from `torchvision`, usually imported as:

```
from torchvision.transforms import transforms
```

- A useful tool found in there is the **affine transformation** using `RandomAffine`:

```
transforms.RandomAffine(degrees, translate=None, scale=None, shear=None)
```

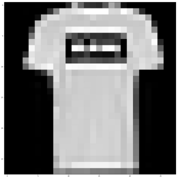
whose objects are python functions (`nn.Modules`, in fact) that perform either a random rotation, translation, scaling, shearing or any subset of them. Its parameters are:

- `degrees` (a number): Range of degrees to select from
- `translate` (a tuple): Maximum image fraction for horizontal and vertical shifts.
- `scale` (a number or a tuple): Scaling factor interval
- `shear` (a number): Range of pixels in the image will be sheared horizontally.

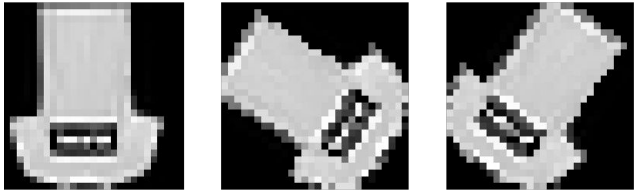
Examples of affine transformations

- Here a some examples of random affine transformations on an image from Fashion MNIST using `transforms.RandomAffine`*:

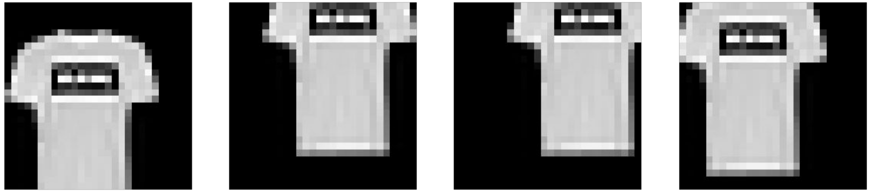
Original



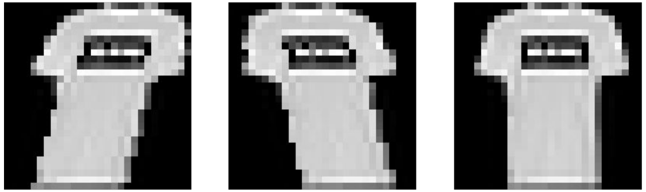
`transforms.RandomAffine(180)`



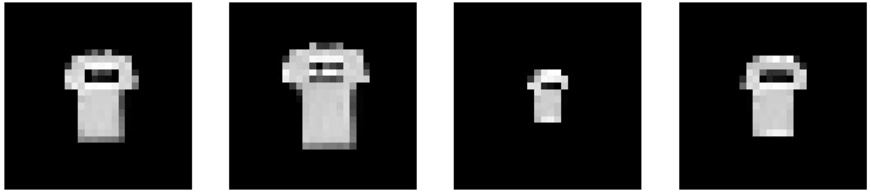
`transforms.RandomAffine(0, translate=(0.3, 0.3))`



`transforms.RandomAffine(0,shear=20)`



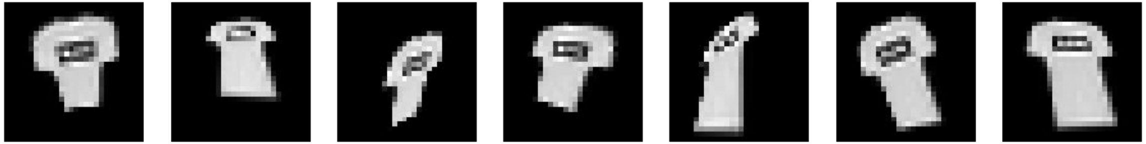
`transforms.RandomAffine(0, scale=(0.2, 0.8))`



*Check the documentation here for more details on the layer and on other possible parameters.

Other Transformations

- The `transforms` library also provides more options of transformations*. For example:
  - Change the perspective (`transforms.RandomPerspective`):



  - Cropping a part of the image out (`transforms.RandomCrop`):

  - Add Gaussian noise (`transforms.GaussianBlur`):

*Here you can find a list of all possible transforms available in PyTorch.

Other Transformations

- Invert the grayscale values / colors (`transforms.RandomInvert`):

- We can compose many different transformations using `transforms.Compose` that receives a list of `transforms` modules and processes them sequentially on the data.
- For example, the following code generates a transformation that first randomly rotates an image and then randomly inverts its colors.

```
transforms.Compose([transforms.RandomAffine(180), transforms.RandomInvert()])
```

Adding a transformation to the dataset

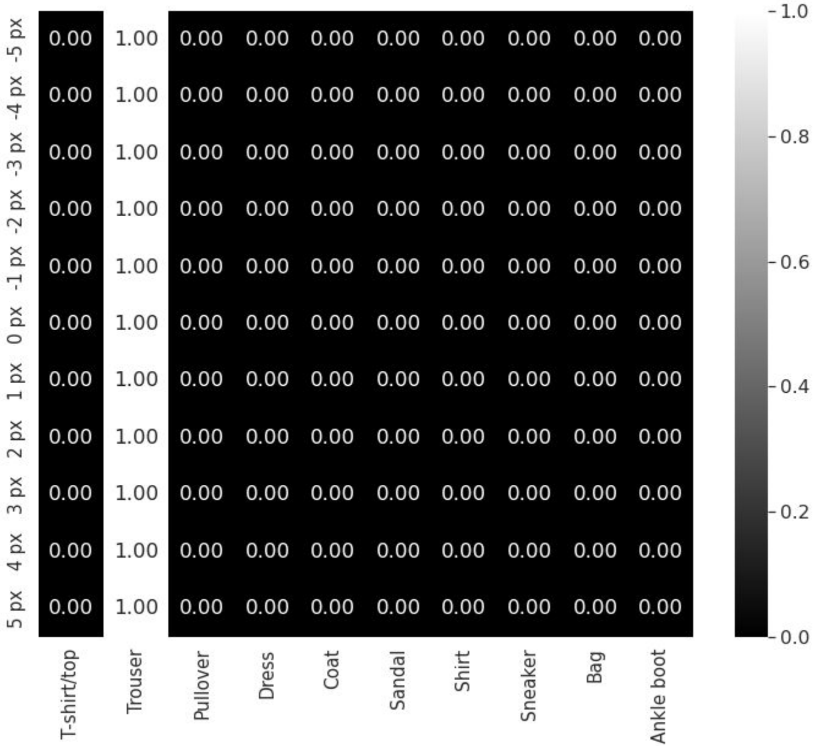
- The simplest way to add a transformation to the dataset is to apply it in the `__getitem__` function to the image being gotten.
- This way, this random transformation will happen whenever the DataLoader is fetching the data to compose the mini-batch.
- In our example, we wish the network to learn that horizontal shifts shouldn't change the object's class.
- Therefore we can augment the dataset by applying random horizontal shifts to the images.

```
from torchvision.transforms import transforms

class FMNISTDataset(Dataset):
  def __init__(self, x, y):
      x = x.view(-1, 1, 28, 28).float()/255
      self.x, self.y = x, y
      self.shift = transforms.RandomAffine(0, translate=(0.5, 0))
  def __getitem__(self, ix):
      x = self.x[ix]
      x = self.shift(x)
      return x.to(device), self.y[ix].to(device)
  def __len__(self):
      return len(self.x)
```

Result of the augmentation

- By making just that change, we are able to achieve the result for the same trouser image from before.
- Notice that the network became more “invariant” to horizontal shifts, as it makes the right prediction with certitude despite the shifts.
- This, however, came at a price:
  a. Adding the random shifting operation at each `__getitem__` made the overall 5 epoch learning take 6 min (from 53s).
  b. The new test accuracy is at around 88% (from 91% from before)

Prob. of each class for various shifts (Augmented)



Problems with augmentation

- The problem a is easy to fix, as the purpose of the previous code was only to serve as an illustration of the augmentation process.
- In PyTorch there are ways to make the transformation application more efficient, by, for example, using them right when you load the data.

```
transform_train = transforms.Compose([transforms.RandomAffine(0, translate=(0.5, 0)),
                                      transforms.ToTensor()])
fmnist_train = datasets.FashionMNIST('~/FMNIST', download=True, train=True, transform=transform_train)
```

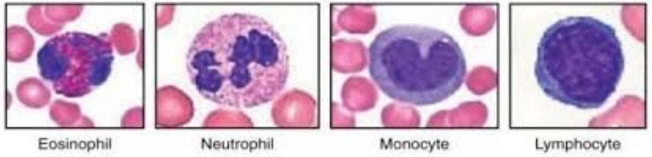
and them changing other parts of the code so we don't need to instantiate our own `Dataset` object, which is inefficient (*these details go beyond the scope of our course*).

- Problem b, however, is harder to solve, since an augmented dataset **is intrinsically richer and more complex** than the original data.
- Typically, it'd require at least **going through more training epochs** or **changing the network** to more complex ones.

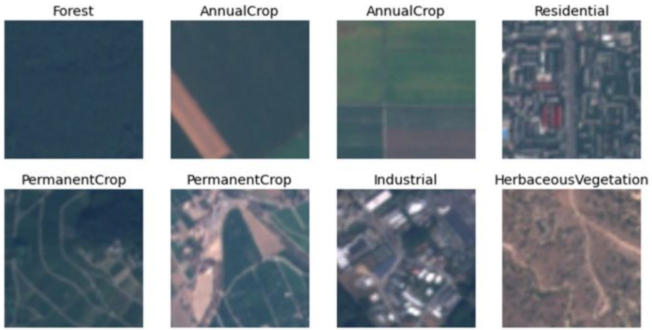
Exercise (*In pairs*)

- For each image classification problem below, list which augmentations make sense and which are risky (not ideal) or unnecessary:

Cell (various types of white blood cell)



Satellite Images (for what type of region)

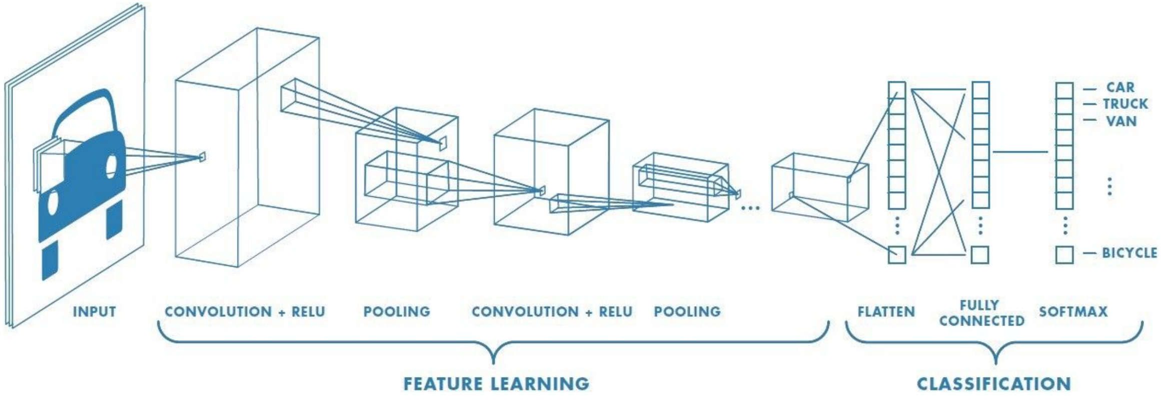


Handwritten Digits

Traffic Sign Classification

Making the model more complex

- We just saw that it is possible to learn a better classification model by presenting a richer variety of data, **even if that data is artificially augmented.**
- Another way to come up with a better model is by training a network whose feature learning phase can capture **more nuanced and representative visual features**.
- With such these more complex features, we hope that the final densely connected layers will be able to output good classifications.



Making the network deeper

- How to come up with better feature learners?
- Over the recent years, researchers have noticed that simply adding more ConvLayers before the dense classifier usually bring improvements.
- This pursuit of more layered nets gave rise to what is know as **Deep Learning**, which is, simply put, **the feature learning process that uses multilayered neural networks.**
- In other words, deep learning is, in many ways, just **representation learning**
- *Later in the course*, we'll see why going deeper helps learning.

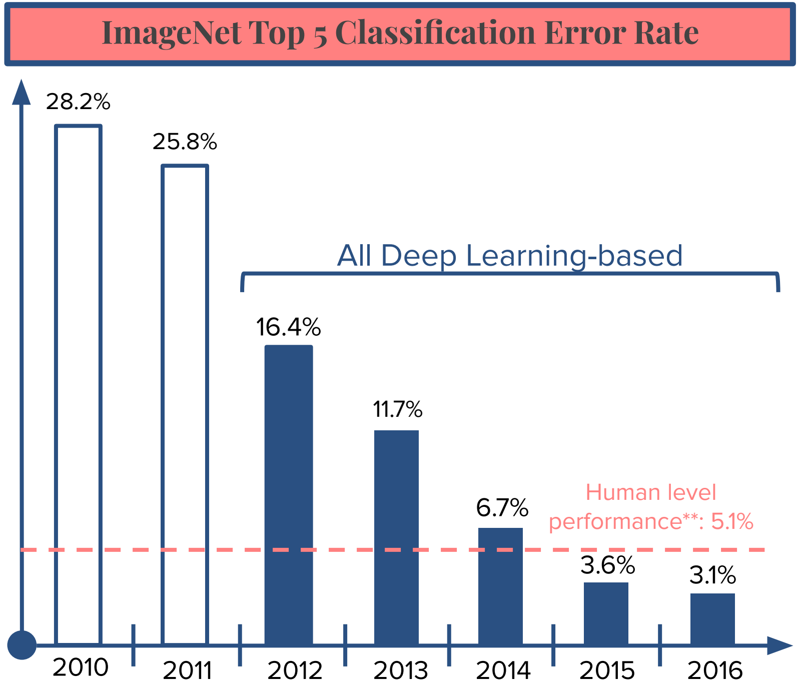
The ImageNet Dataset

- Historically, Deep Learning started to impress the world in 2012, when a deep net called AlexNet broke the classification record on the ImageNet dataset.
- This dataset spans 1000 classes and contains $1{,}281{,}167$ training and $100{,}000$ test images* of various sizes.
- The **images are very realistic**, all hierarchically annotated by humans.

*In fact, this is just a subset of +14 million images spanning more than 20k classes called the ImageNet project. More info on it here.

The ImageNet Challenge

- Since 2012, Deep Learning has outperformed every other method in the ImageNet's Top 5* Classification competition.
- Starting from 2014, it also **overcame humans**** when submitted to the same challenge.
- One common feature of all these winning networks is that they were **getting deeper and deeper**.
- Today we'll focus on one of the runner-ups from the 2014 edition: the **VGG16 network**.

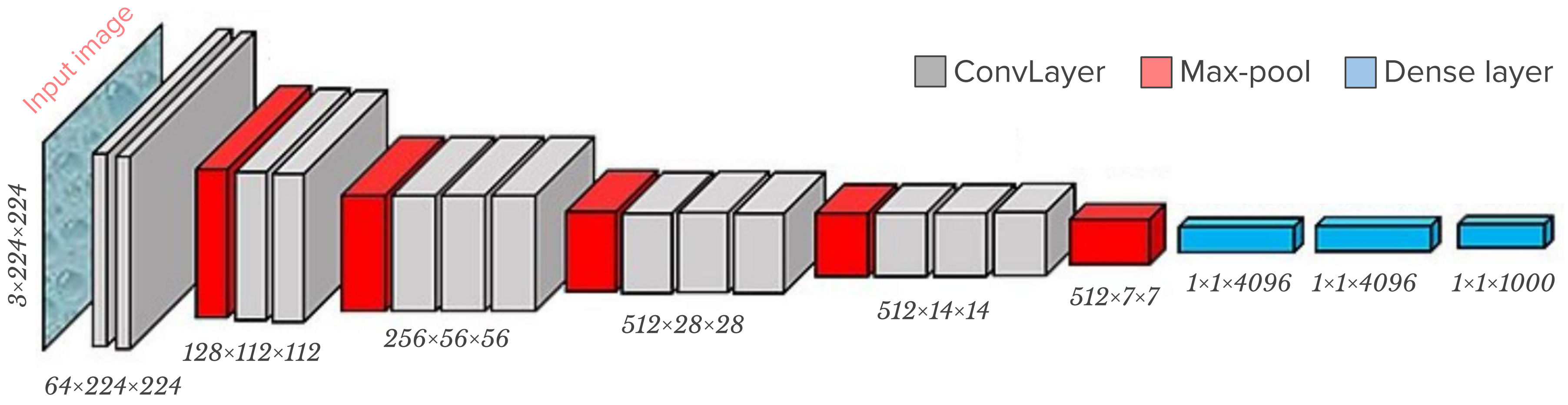


* The true class only need to be among the top 5 predicted classes to be considered a successful prediction.

** Note that these methods need to identify 1 of a 1000 possible classes, while humans can recognize a much larger number of categories.

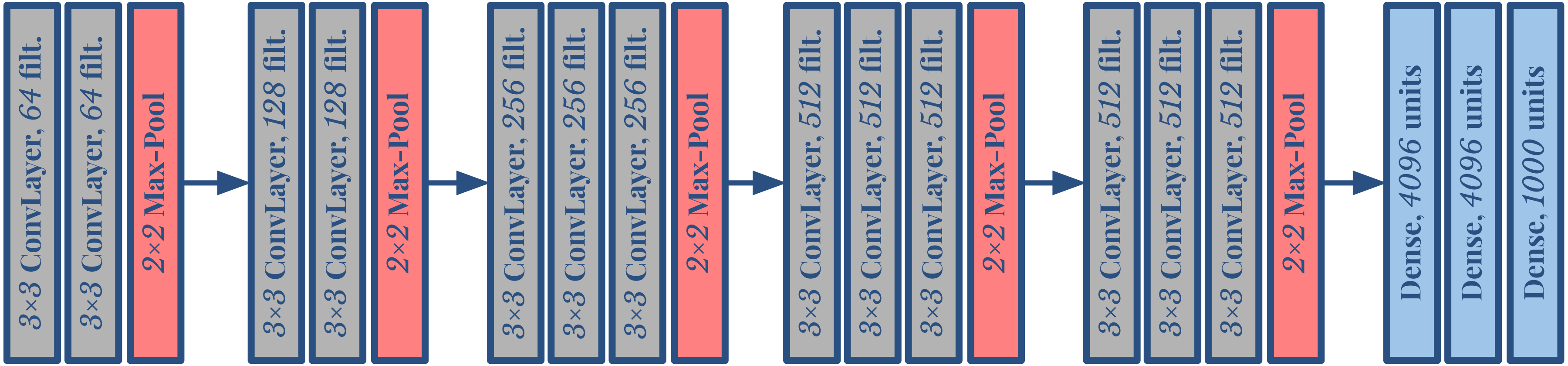
The VGG16 Network

- The VGG16 net, for Visual Geometry Group (VGG) at University of Oxford, who developed the network in 2014, is a simple, by very deep network, with 16 layers!
- While the input RGB image has to be reshaped to 224×224 pixels, it uses many ConvLayers and max-poolings to gradually decrease its size, before the dense layers.



VGG16 in PyTorch

- In a simplified way, the VGG16 can be summarized as follows:



- Although I'm sure you can code that network up from scratch, PyTorch also provides the model as it was conceived via in `tourchvision`:

```python
from torchvision import models
model = models.vgg16()
```

The summary of VGG16

```python
from torchsummary import summary
summary(model.to(device), (3, 224, 224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
              ReLU-2         [-1, 64, 224, 224]               0
            Conv2d-3         [-1, 64, 224, 224]          36,928
              ReLU-4         [-1, 64, 224, 224]               0
         MaxPool2d-5         [-1, 64, 112, 112]               0
            Conv2d-6        [-1, 128, 112, 112]          73,856
              ReLU-7        [-1, 128, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]         147,584
              ReLU-9        [-1, 128, 112, 112]               0
        MaxPool2d-10          [-1, 128, 56, 56]               0
           Conv2d-11          [-1, 256, 56, 56]         295,168
             ReLU-12          [-1, 256, 56, 56]               0
           Conv2d-13          [-1, 256, 56, 56]         590,080
             ReLU-14          [-1, 256, 56, 56]               0
           Conv2d-15          [-1, 256, 56, 56]         590,080
             ReLU-16          [-1, 256, 56, 56]               0
        MaxPool2d-17          [-1, 256, 28, 28]               0
           Conv2d-18          [-1, 512, 28, 28]       1,180,160
             ReLU-19          [-1, 512, 28, 28]               0
           Conv2d-20          [-1, 512, 28, 28]       2,359,808
             ReLU-21          [-1, 512, 28, 28]               0
           Conv2d-22          [-1, 512, 28, 28]       2,359,808
             ReLU-23          [-1, 512, 28, 28]               0
        MaxPool2d-24          [-1, 512, 14, 14]               0
           Conv2d-25          [-1, 512, 14, 14]       2,359,808
             ReLU-26          [-1, 512, 14, 14]               0
           Conv2d-27          [-1, 512, 14, 14]       2,359,808
             ReLU-28          [-1, 512, 14, 14]               0
           Conv2d-29          [-1, 512, 14, 14]       2,359,808
             ReLU-30          [-1, 512, 14, 14]               0
        MaxPool2d-31            [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-32            [-1, 512, 7, 7]               0
           Linear-33                 [-1, 4096]     102,764,544
             ReLU-34                 [-1, 4096]               0
          Dropout-35                 [-1, 4096]               0
           Linear-36                 [-1, 4096]      16,781,312
             ReLU-37                 [-1, 4096]               0
          Dropout-38                 [-1, 4096]               0
           Linear-39                 [-1, 1000]       4,097,000
================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
----------------------------------------------------------------(...)
```

A new type of layer.

Adaptative Average Pooling and Other VGG's

- As you may have noticed on the previous summary*, VGG16 utilizes a layer we haven't yet learned, the **Adaptive Average Pooling layer**.

```
              (...)                 (...)          (...)
            ReLU-30      [-1, 512, 14, 14]             0
       MaxPool2d-31        [-1, 512, 7, 7]             0
AdaptiveAvgPool2d-32       [-1, 512, 7, 7]             0
          Linear-33             [-1, 4096]   102,764,544
              (...)                 (...)          (...)
```

- It is similar to nn.AvgPool2d, which returns the average of a section instead of the maximum, which nn.MaxPool2d does. In both cases, we set choose the kernel size.
- In nn.AdaptativeAvgPool2d, we instead set the output size, and it automatically computes the kernel size so that the specified size is returned.
- This layer plays an important role **in the transition from the feature learning phase to the classifier** and will be important in our next class.
- This layer is found is other models, such as VGG16's "siblings": VGG13 and VGG19, width 13 and 19 layers, respectively, which can be used via models.vgg13(), and models.vgg19().

* Despite not explicitly showing here, there is a flattening layer in between the AvgPool and the Linear layers, as its official implementation recognizes.

The challenges of Deep Nets

- Note that in VGG16 we have to train **more than 135 million parameters** on **RGB images of size 224×224!**
- Using a simple GPU, we were taking ~1 min to learn 800k weights for just 5 epochs on 60000 grayscale images of size 28×28.
- For most applications, **it is not worth** to retrain these networks, especially if one is running on a low computational/memory budget.
- Also, the dataset VGG16 was trained on (ImageNet) has +1 million images to be trained on.
- Two issues that are very common in most deep learning applications:
    a. The **models are huge** and most companies can't afford the of computational requirement.
    b. These models need to be **trained on very large datasets** so to justify their complexity. In many applications, the datasets are very small (*one could recur to data augmentation in this case*).
- *Next class*, we'll see how we can still leverage the capacities of deep learning models in the applications at a considerably low computational cost.

Exercise (*In pairs*)

- Go back the VGG16's summary and explain how the output sizes change as they do (remember that each ConvLayer uses 3×3 kernels). *Hint*: try to **print** the model and see is it gives you any help:

```
from torchvision import models
model = models.vgg16()
print(model)
```