Deep Learning Applied

Handling Images + Transfer Learning

Daniel Yukimura

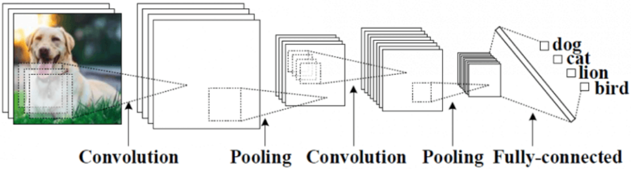
yukimura@impa.br

September 26, 2018

Working with images

Convolutional Neural Networks (CNN):

Convolutional layers are essential for processing image data.

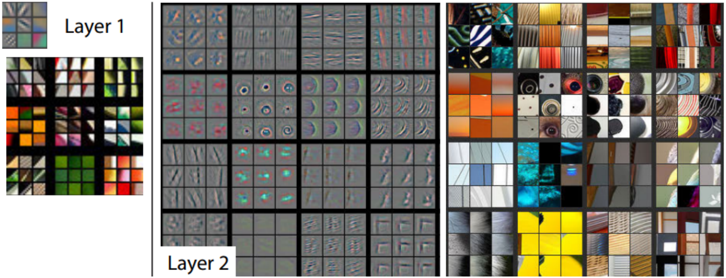


To convert a 256×256 RGB image to another one, a typical linear layer would require about $(256 \times 256 \times 3)^2 \approx 3.87e + 10$ ($\approx 150Gb$), an extreme excess of parametrization. Instead this transformation is replaced by a convolution with learnable filters.

Working with images

Convolutional Neural Networks (CNN):

This layers work essentially as feature maps, and can usually specialize on simple tasks like finding lines, corner, edges, or more refined ones, like textures, face parts, forms, etc.

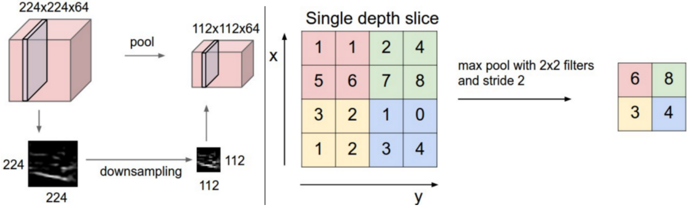


Matthew Zeiler & Rob Fergus

Working with images

Pooling layers

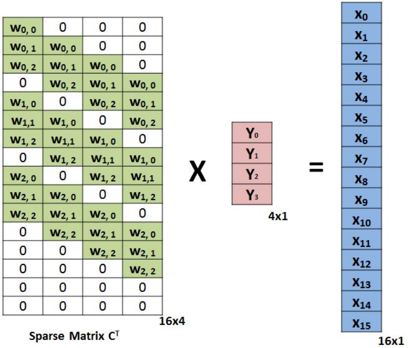
On the processing pipe line we can have downsampling layers, where we reduce dimension while preserving the information significance. The most common operation used is **maxpooling** where we carry the highest activation value forward in each cell.



Transpose convolution

This operation maps spatial shapes in the opposite direction, maintaining the connections of a regular convolutional layer. Is usually applied when the target variable Y is an image for example.

This can be done by: Rearrange into vectors, and transpose the operator



Working with images

Transpose convolution

This operation maps spatial shapes in the opposite direction, maintaining the connections of a regular convolutional layer. Is usually applied when the target variable Y is an image for example.

This can be done by: Rewrite as a padded/fractional convolution

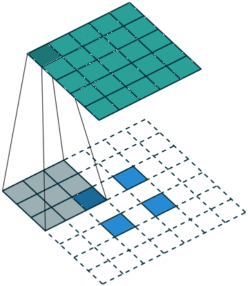


Image Classification

Goal: To predict the class of an image, which often refers to the "main object" in the image.

Measuring performance:

The standard formats are

- The **error rate** $\hat{P}(f(X, \theta) \neq Y)$, or conversely the **accuracy** $\hat{P}(f(X, \theta) = Y)$
- The **balance error rate** (BER)

$$\frac{1}{C} \sum_{y=1}^{C} \hat{P}(f(X, \theta) = y \mid Y = y)$$

Object Detection

Goal: Predicting **classes and locations** of targets in images. The standard setting outputs a collection of bounding boxes, with classes associated to each.

To quantify performance the standard metric is using **intersections over unions** (IoU). A predicted bounding box $\hat{B}$ is correct if there is some annotated bounding box B for that class, such that the IoU is big enough

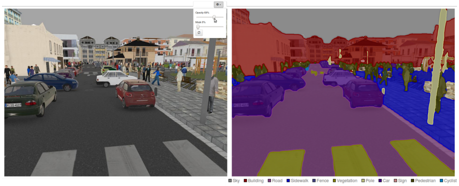
$$\text{IoU} = \frac{\text{area}(B \cap \hat{B})}{\text{area}(B \cup \hat{B})} > \frac{1}{2}$$

Semantic Segmentation

Goal: Consists in labeling individual pixels with the class of the object it refers to.
A standard performance metric is **segmentation accuracy** (SA) given as

$$\mathrm{SA} = \frac{n}{n+e}$$

where n is the number of pixels on the true class, predicted correctly, and e the number of pixels erroneously labeled.



Datasets

Available in torchvision.datasets:

- **MNIST** and **Fashion-MNIST**: 50k train images, 10k test images, 28×28 grayscale, labeled on 10 classes.

MNIST

Fashion MNIST

- **CIFAR10** and **CIFAR100** (10 classes and 5×20 "super classes"),: 50k train images, 10k test images, 32×32 RGB.

Datasets

- **ImageNet:** http://www.image-net.org/
  - ≈ 14M images ("Large scale")
  - ≈ 1M images with bounding box annotations

  ImageNet Large Scale Visual Recognition Challenge 2012:

  - 1k classes
  - 1.2M training images and 50k validation images.

Datasets

- **CelebFaces Attributes Dataset (CelebA):** $\approx$ 200K celebrity images, each with 40 attribute annotations

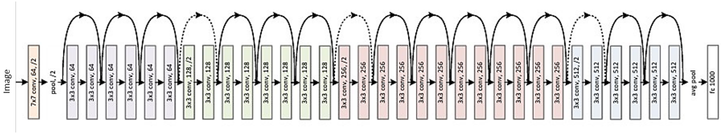
ConvNets

- **Standard models for Image Classification**: The **LeNet** family (leCun et al., 1998) and modern extensions, like the **AlexNet**(Krizhevsky et al., 2012) and **VGGNet** (Simonyan and Zisserman, 2014).

In PyTorch:

`torchvision.models.alexnet`, `torchvision.models.vgg16`

- **Residual Networks(ResNet)**: Uses skip (or short-cut) connections, creating a better gradient flow, it avoids the **vanishing gradient** problem which is critical in networks with large depth.



In PyTorch:

```
torchvision.models.resnet34
```

Transfer Learning

- Is the practice of exploiting what has been learned for some task A to improve generalization on a task B.
- Using a model trained for a task A on a large dataset, we exploit the learned features for learning a task B where data is scarce, but of the same type as task A.
- The idea is to repurpose the learned feature maps of a well trained model, to give a good head start on the training of a new task that doesn't has as many data points.

Transfer Learning

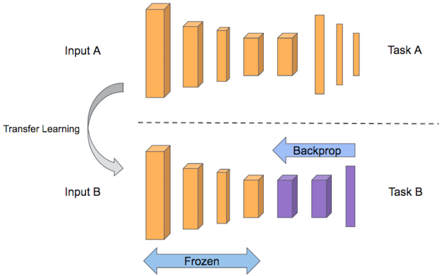
- Is the practice of exploiting what has been learned for some task A to improve generalization on a task B.
- Using a model trained for a task A on a large dataset, we exploit the learned features for learning a task B where data is scarce, but of the same type as task A.
- The idea is to repurpose the learned feature maps of a well trained model, to give a good head start on the training of a new task that doesn't has as many data points.

Transfer Learning

- Is the practice of exploiting what has been learned for some task A to improve generalization on a task B.
- Using a model trained for a task A on a large dataset, we exploit the learned features for learning a task B where data is scarce, but of the same type as task A.
- The idea is to repurpose the learned feature maps of a well trained model, to give a good head start on the training of a new task that doesn't has as many data points.

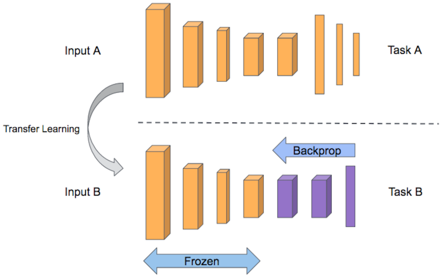
Transfer Learning on Neural Networks

- Change the architecture and reinitialize the weights on the last layers (one or more).
- To train on a new task we can opt from retraining all parameters, or only the ones on the remodelled layers.



Transfer Learning on Neural Networks

- When we are training the whole network in a new task, the initial faze is usually called **pre-training**.
- We can also freeze the original layers and only retrain on the new ones, we call this process **fine-tuning**.



Transfer Learning

Example: Dogs vs Cats

http://files.fast.ai/data/dogscats.zip

- 25k images of dogs and cats.
- In 2013 the Kaggle competition on this dataset had an accuracy of about 80%. (link)

Transfer Learning

Example: Dogs vs Cats

Using the fast.ai library we can easily set a transfer learning setting.

- **Source Model:** pre-trained ResNet34.

```
s_model = resnet34
data = ImageClassifierData.from_paths(PATH, tfms=tfms_from_model(s_model, s
learn = ConvLearner.pretrained(s_model, data, precompute=True)
learn.fit(0.01, 2)
```

[notebook]

Transfer Learning

Example: Image Colorization

We can also exploit transfer knowledge on tasks apart from classification, in this example we'll see it for a colorization problem.

Image Colorization

Setting:

- Given a grayscale image, which we consider as the lightness component, we want to infer saturation and hue. (We are using LAB colorspace).
- **Data:** We are using the MIT places, a dataset of places, landscapes, and buildings. It contains almost 2.5M images.
- Our input has size 256×256 ($\times 1$), and our outputs are of size $256 \times 256 \times 2$.

Image Colorization

Model:

- The model has a "autoencoder" kind of structure. We begin with a series of convolutional layers **pre-trained**, and then use transpose convolutions to infer the other two color channels.
- The first pre-trained part comes from **ResNet18**, where we modified the input for grayscale images, and we will cut it off after the 6th set of layers.
- The second part has a series of transposed convolutions generating the $256 \times 256 \times 2$ output

Image Colorization

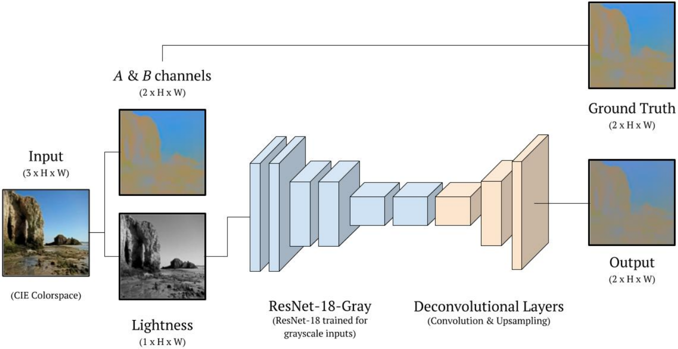


Image Colorization

```
class ColorizationNet(nn.Module):
    def __init__(self, input_size=128):
        super(ColorizationNet, self).__init__()
        MID_FT_SIZE = 128

        ## First half: ResNet
        resnet = models.resnet18(num_classes=365)
        # Grayscale
        resnet.conv1.weight = nn.Parameter(resnet.conv1.weight.
                                      sum(dim=1).unsqueeze(1))
        # Midlevel features
        self.midlevel_resnet = nn.Sequential(*list(resnet.children())[0:6])
```

Image Colorization

```
...## Second half: Upsampling
self.upsample = nn.Sequential(
    nn.Conv2d(MID_FT_SIZE, 128, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(128), nn.ReLU(),
    nn.Upsample(scale_factor=2),
    nn.Conv2d(128, 64, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(64), nn.ReLU(),
    nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(64), nn.ReLU(),
    nn.Upsample(scale_factor=2),
    nn.Conv2d(64, 32, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(32), nn.ReLU(),
    nn.Conv2d(32, 2, kernel_size=3, stride=1, padding=1),
    nn.Upsample(scale_factor=2))
```

Image Colorization

```
...
def forward(self, input):
    midlevel_features = self.midlevel_resnet(input)

    output = self.upsample(midlevel_features)
    return output
```

Image Colorization

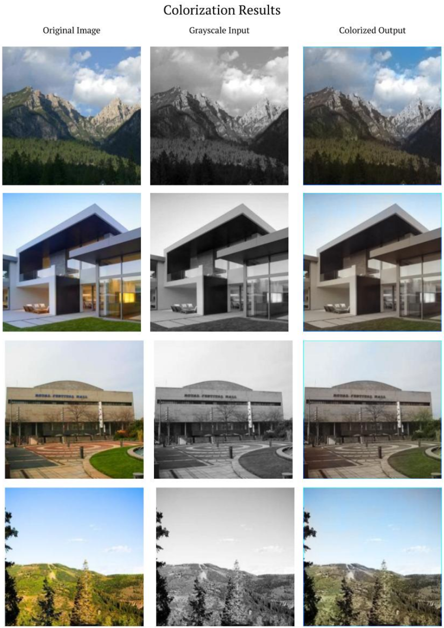
Training the model:

```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-2, weight_decay=0.0)
...
def train(train_loader, model, criterion, optimizer, epoch):
    model.train()
    ...
    for i, (input_gray, input_ab, target) in enumerate(train_loader):
        ...
        loss = criterion(output_ab, input_ab)
        losses.update(loss.item(), input_gray.size(0))
        ...
```

Image Colorization

Training the model:

```
        ...
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    ...
    for epoch in range(epochs):
        # Train for one epoch, then validate
        train(train_loader, model, criterion, optimizer, epoch)
        with torch.no_grad():
            losses = validate(val_loader, model, criterion, save_images,
                              epoch)
```


Colorization Results
Original Image
Grayscale Input
Colorized Output