BST 261: Data Science II
Lecture 7
Convolutional Neural Networks (CNNs): Data Augmentation and Pretrained networks

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2020

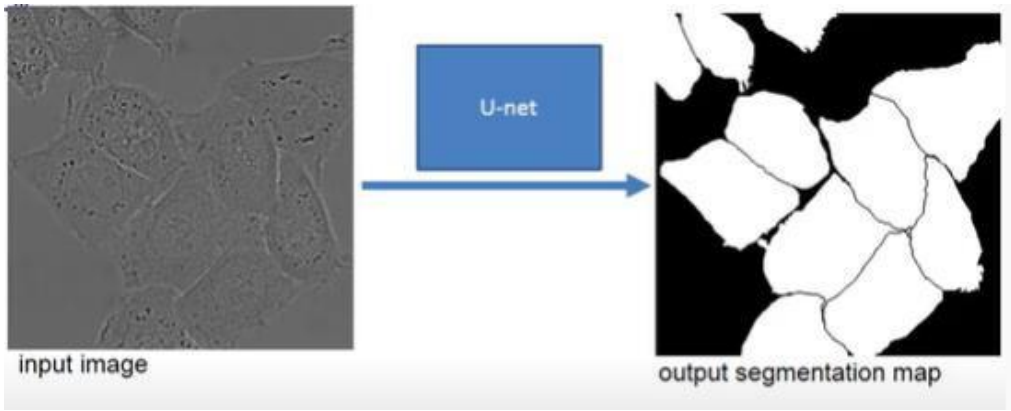
Recipe of the Day!

Rosemary crackers

Paper Presentations

Biomedical image segmentation using U-net

Olaf Ronneberger, Philipp Fischer, and Thomas Brox,2015



Xiaoxuan (Jennifer) Liu

M.S in Health Data Science

- Motivation
- Architecture
- Experiment
- Conclusion

Motivation

- Previous work:
  - Ciresan et al. trained a network in a sliding-window setup to predict the class label of each pixel by providing a local region (patch) around that pixel in biomedical image processing
  - Two drawbacks.
    - Slow: redundancy due to overlapping patches.
    - Trade-off between localization accuracy and the use of context
- Challenge in many cell segmentation tasks:
  - **Very few annotated** images(about 30 per application)--augmentation
  - **Touching objects** of the same class—weighted loss
- “fully convolutional network” : works with very few training images and

  yields more precise segmentations

Biomedical image segmentation using U-net Architecture

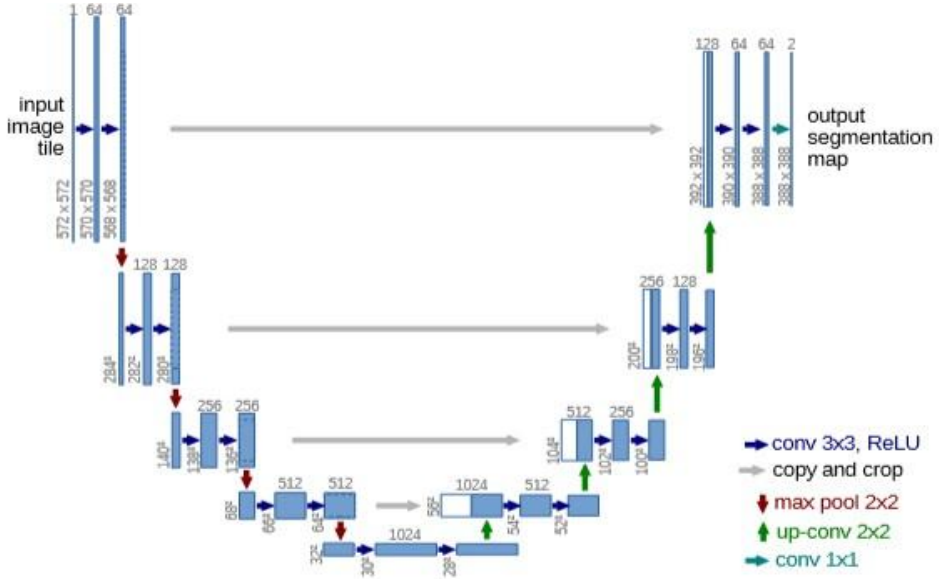


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

The advantage of this structure

- contracting path + expanding path
- trained end-to-end from very few images
  - use excessive data augmentation by applying elastic deformations to the available training images. This allows the network to learn invariance to such deformations, without the need to see these transformations in the annotated image corpus. This is particularly important in biomedical segmentation, since deformation used to be the most common variation in tissue and realistic deformations can be simulated efficiently.
- Fast. Segmentation of a 512x512 image takes less than a second on a recent GPU

3 different segmentation tasks.

- The first task is the segmentation of neuronal structures in electron microscopic recordings. (Figure 2) The training data is a set of 30 images (512x512 pixels). The u-net achieves without any further pre- or postprocessing a warping error of 0.000352[9] (the new best score)

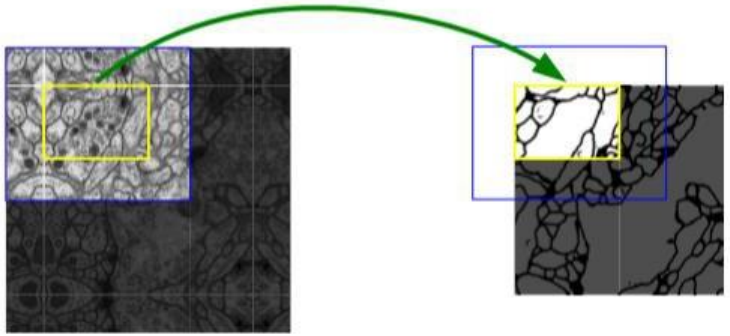


Fig. 2. Overlap-tile strategy for seamless segmentation of arbitrary large images (here segmentation of neuronal structures in EM stacks). Prediction of the segmentation in the yellow area, requires image data within the blue area as input. Missing input data is extrapolated by mirroring

2nd: a cell segmentation task in light microscopic images (see Figure 4a,b),35 partially annotated training
images. Here we achieve best average IOU (“intersection over union”) of 92%.

- 3rd: “DIC-HeLa”3: HeLa cells on a flat glass recorded by differential interference contrast (DIC) microscopy IOU of 77.5%

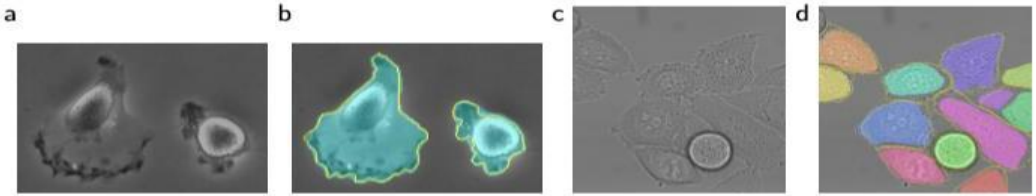


Fig. 4. Result on the ISBI cell tracking challenge. (**a**) part of an input image of the “PhC-U373” data set. (**b**) Segmentation result (cyan mask) with manual ground truth (yellow border) (**c**) input image of the “DIC-HeLa” data set. (**d**) Segmentation result (random colored masks) with manual ground truth (yellow border).

Table 2. Segmentation results (IOU) on the ISBI cell tracking challenge 2015.

Name	PhC-U373	DIC-HeLa
IMCB-SG (2014)	0.2669	0.2935
KTH-SE (2014)	0.7953	0.4607
HOUS-US (2014)	0.5323	-
second-best 2015	0.83	0.46
u-net (2015)	**0.9203**	**0.7756**

Biomedical image segmentation using U-net Conclusion

The u-net architecture achieves very good performance on very different biomedical segmentation applications.

- Thanks to data augmentation with elastic deformations, it only needs very few annotated images and has a very reasonable training time of only 10 hours. This is important in biomedical segmentation since deformation is the most common variation in tissue and realistic deformations can be simulated efficiently.
- Thanks to the use of weighted loss, the network learn the small separation borders between touching cells and successfully separate touching objects of the same class

Fully convolutional networks for semantics segmentation

Long J, Shelhammer E, and Darell T. 2015

Presented by Ninon Becquart

Introduction

- Convolutional networks improve whole-image classification & progress on local tasks with structured output
- As we progress from coarse to fine inference, the next step is to make a prediction at every pixel
- Convolutional networks are used in semantic segmentation
  - Each pixel is labeled with the class of its enclosing object or region
  - But this has shortcomings

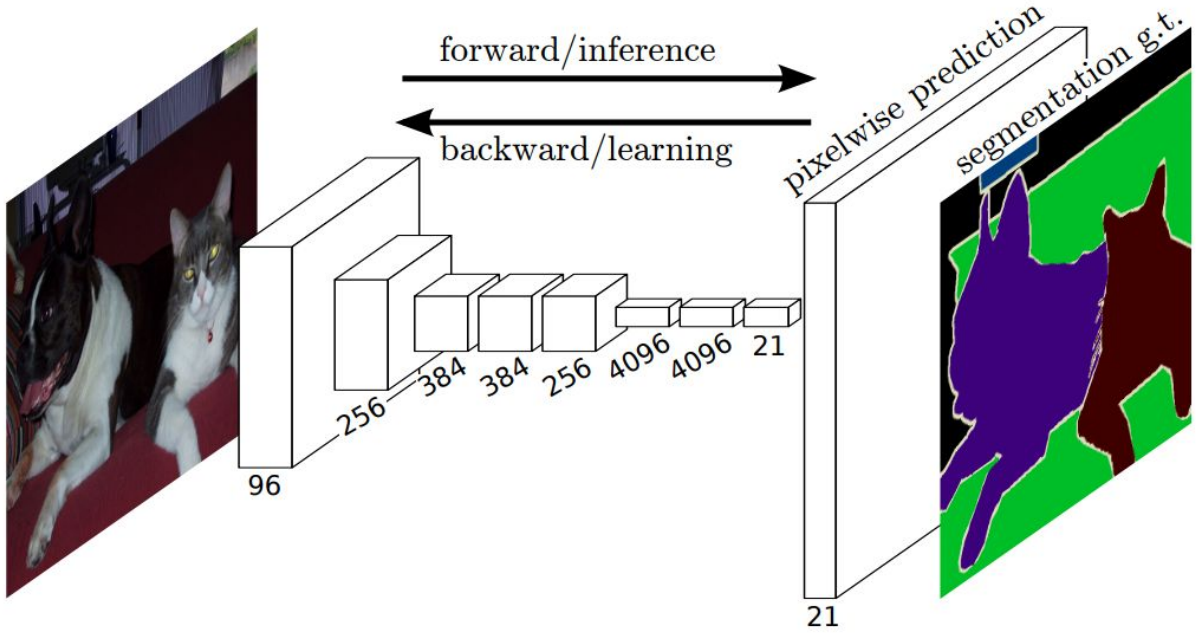


Figure 1. Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation.

Hypotheses

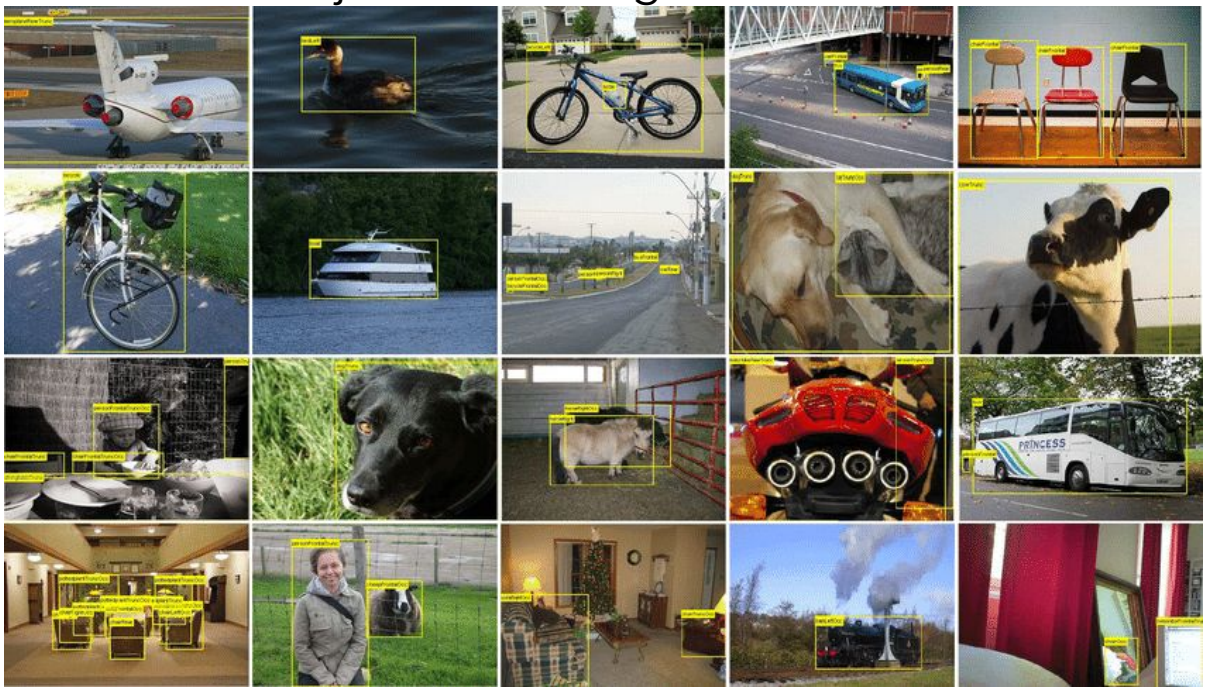
- These researchers wanted to show that a fully convolutional network trained end-to-end, pixel-to-pixel on semantic segmentation exceeds state-of-the-art without further machinery
- Further machinery meaning pre- and post- processing complications

Methods

- They tested their fully convolutional network (FCN) on semantic segmentation and scene parsing, exploring PASCAL VOC, NYUDv2, and SIFT Flow

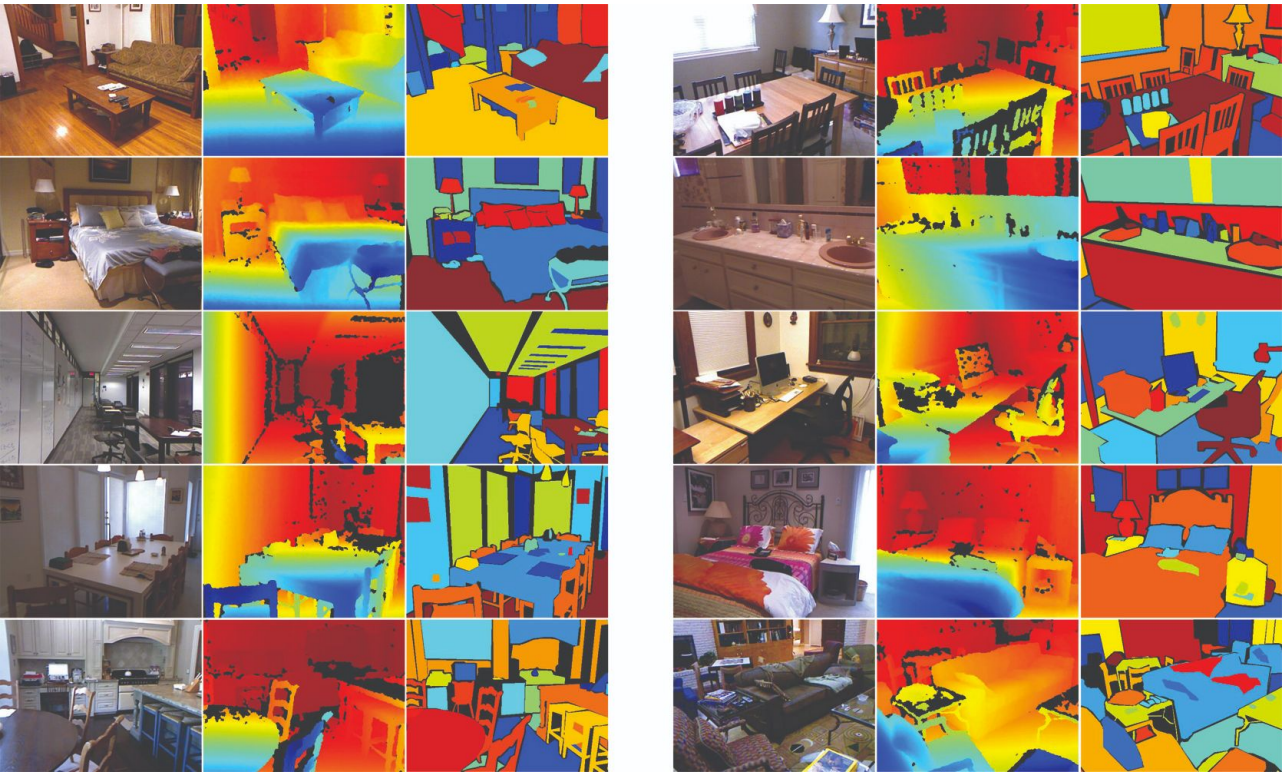
PASCAL VOC?

Standardized image datasets for object class recognition



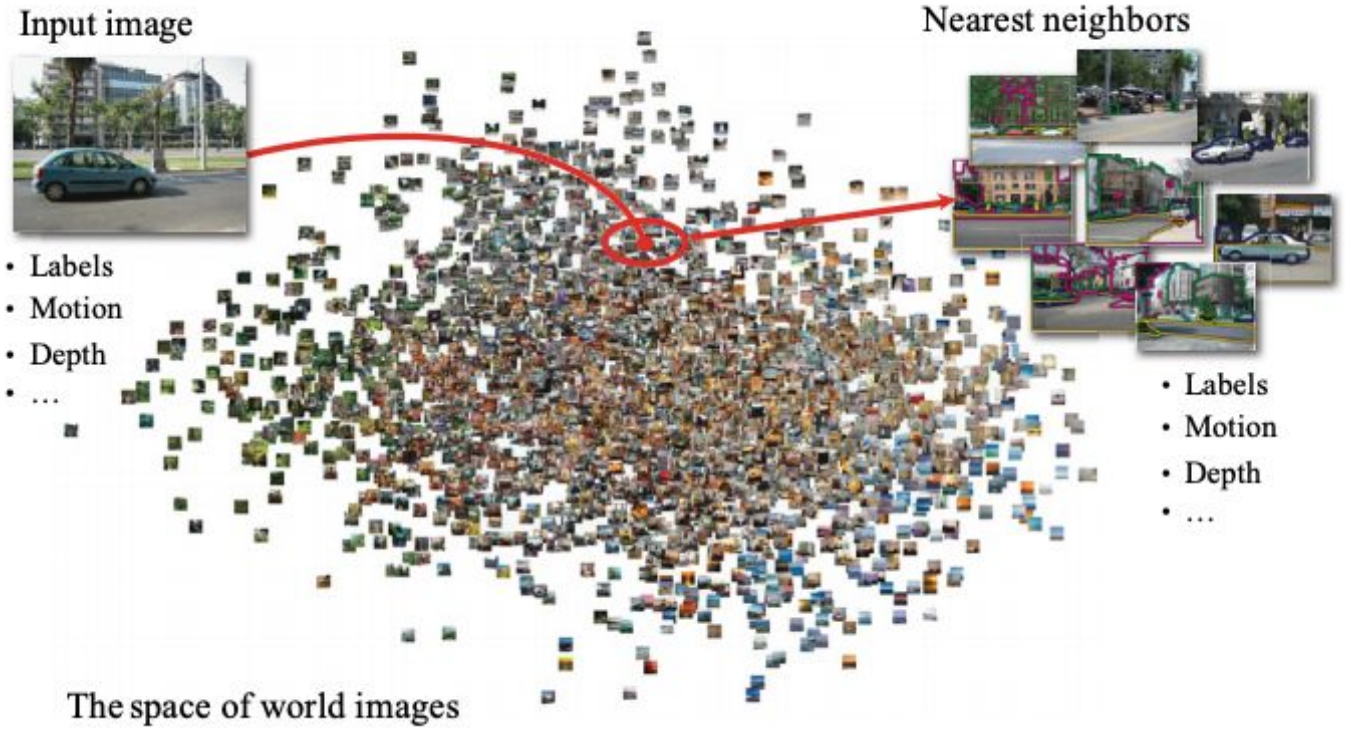
NYUD v2 ??

1,449 RGB-D images with pixel-wise labels



SIFT Flow ???

A dataset of 2,688 images with pixel labels for 33 semantic categories & 3 geometric categories



Methods

- These tasks have historically distinguished between objects and regions
  - But they treat both universally as pixel prediction
- They evaluated their FCN ship architecture on each of these datasets, and then extended it to multi-modal input for NYUD v2 and multi-task prediction for semantic and geometric labels of SIFT Flow

Results

- Metrics used to evaluate how well their FCN did:
  - Pixel accuracy
  - Mean accuracy
  - Mean IU (region intersection over union)
  - Frequency IU

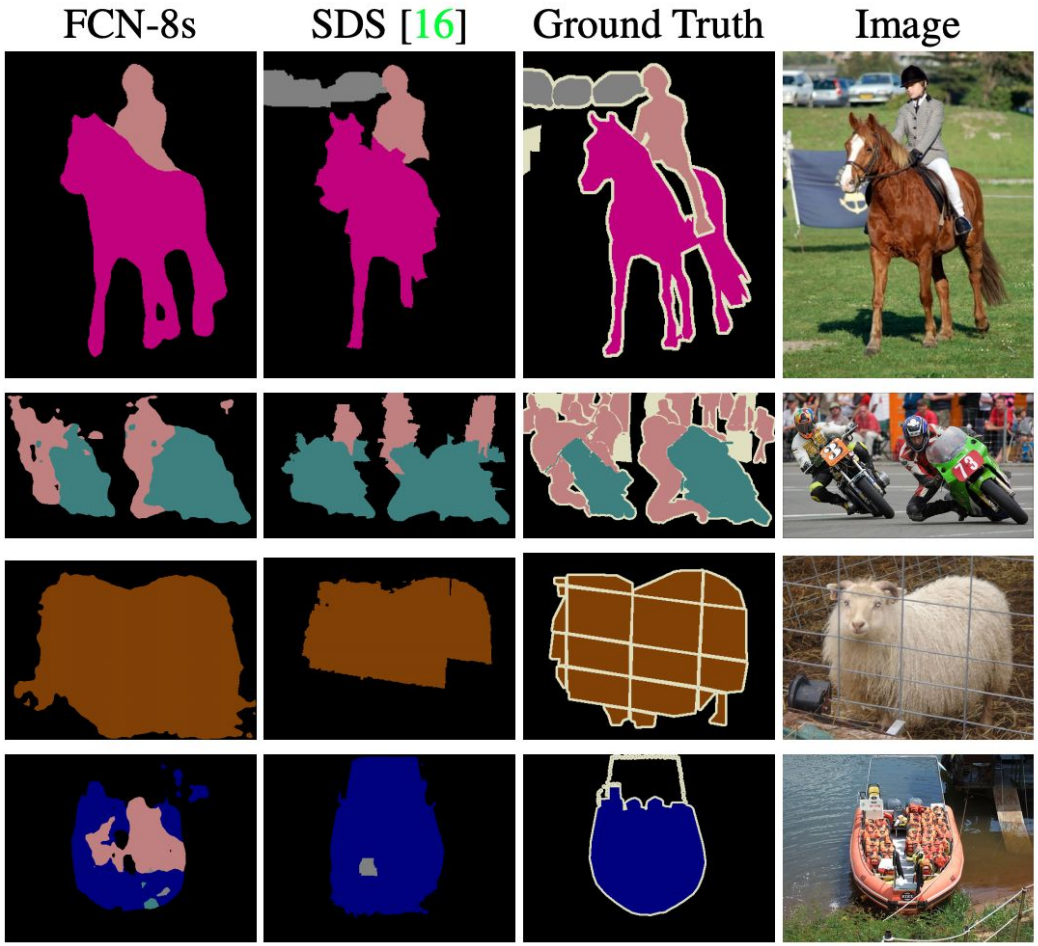


Figure 6. Fully convolutional segmentation nets produce state-of-the-art performance on PASCAL. The left column shows the

Conclusion

"Our fully convolutional network achieves state-of-the-art segmentation of PASCAL VOC (20% relative improvement to 62.2% mean IU on 2012), NYUDv2, and SIFT Flow, while inference takes less than one fifth of a second for a typical image"

Data Augmentation

Data Augmentation

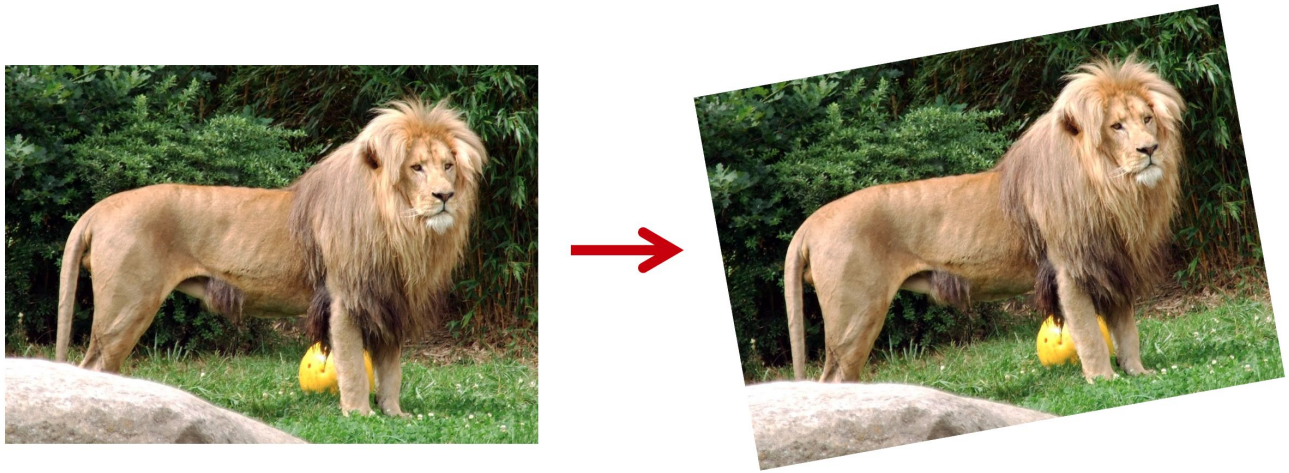
- As we have seen, overfitting is caused by having too few training examples to learn from
- Data augmentation generates more training data from existing training examples by **augmenting** the samples via a number of random transformations
- These transformations should yield believable images

Data Augmentation

- Types of augmentation:
  - Rotation
  - Horizontal/vertical flip
  - Random crops/scales
    - Zoom
    - Width or height shifts
  - Shearing
  - Brightness, contrast, saturation
  - Lens distortions
  - Etc. (see Keras documentation for full list)

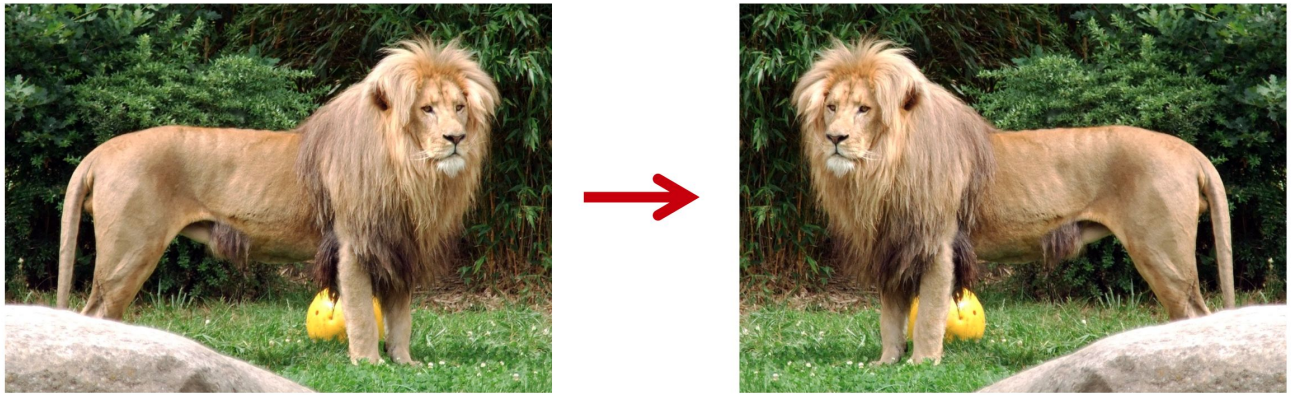
Types of data augmentation

1. Rotations



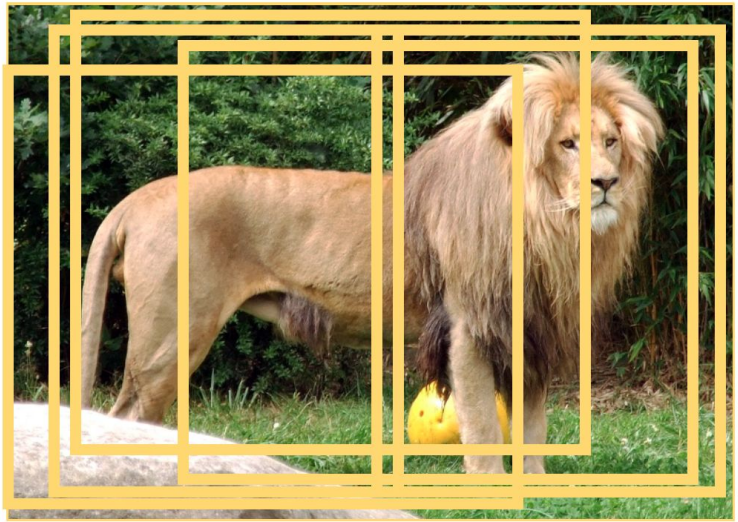
Types of data augmentation

2. Horizontal/Vertical Flips



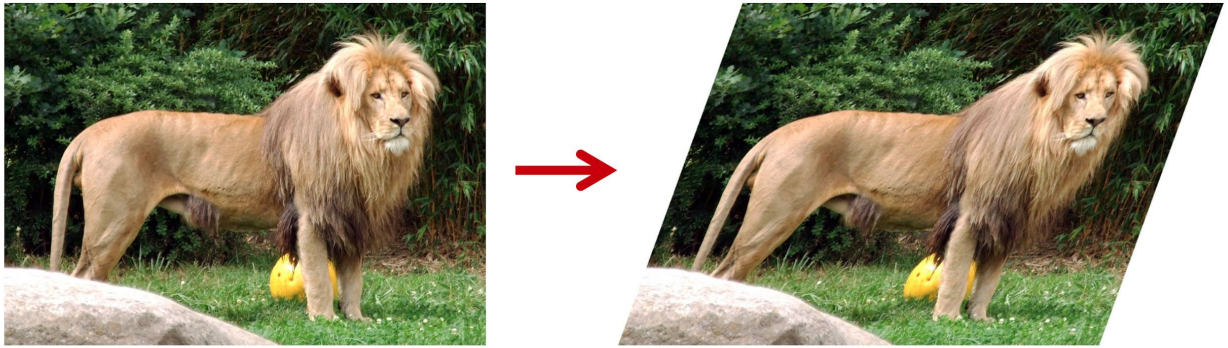
Types of data augmentation

3. Random crops/scales



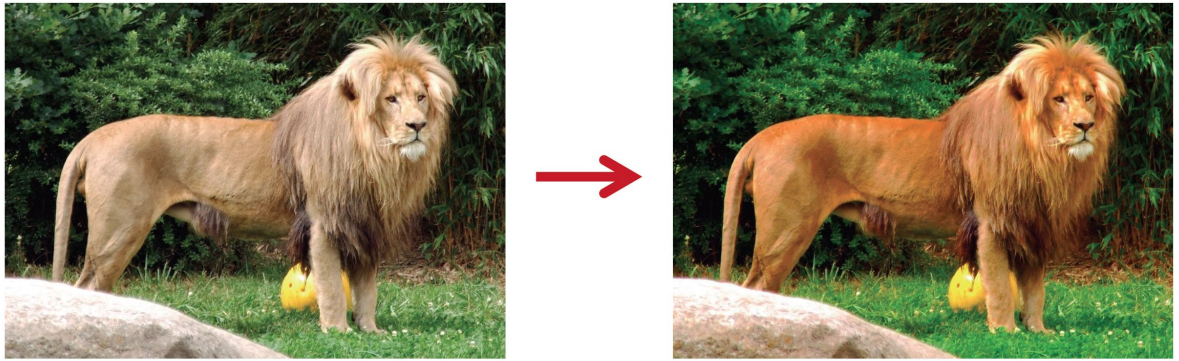
Types of data augmentation

4. Shearing



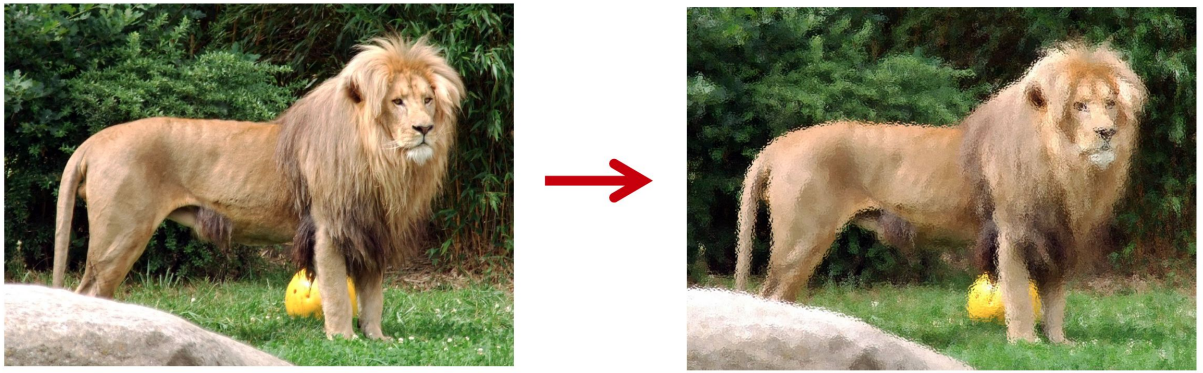
Types of data augmentation

5. Brightness, contrast, saturation



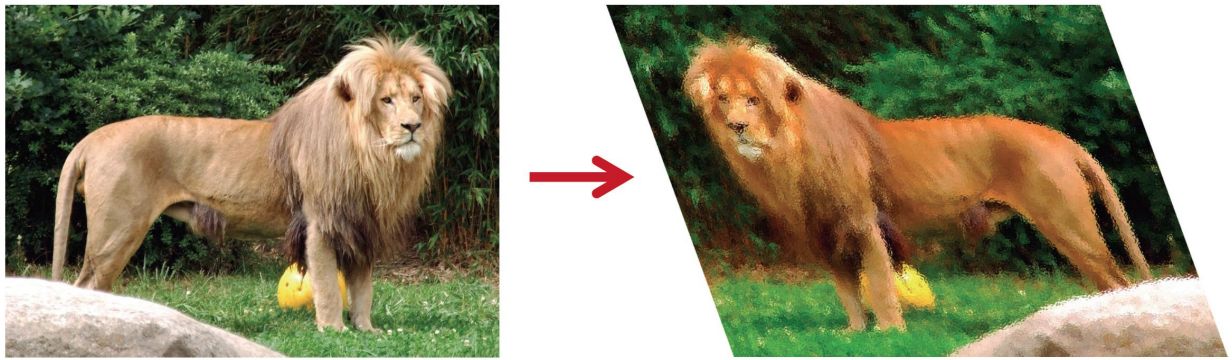
Types of data augmentation

6. Lens distortions



Types of data augmentation

7. Combinations of the above



Data Augmentation

- If you train a network using data augmentation, it will never see the same input twice, but the inputs will still be heavily correlated
  - You're **remixing** known information, **not producing new** information
- May not completely escape overfitting due to this **correlation**
- Adding dropout can also help
- With data augmentation, each epoch uses different (augmented) images, without data augmentation, each epoch uses the same (training set) images

Data Augmentation in Keras

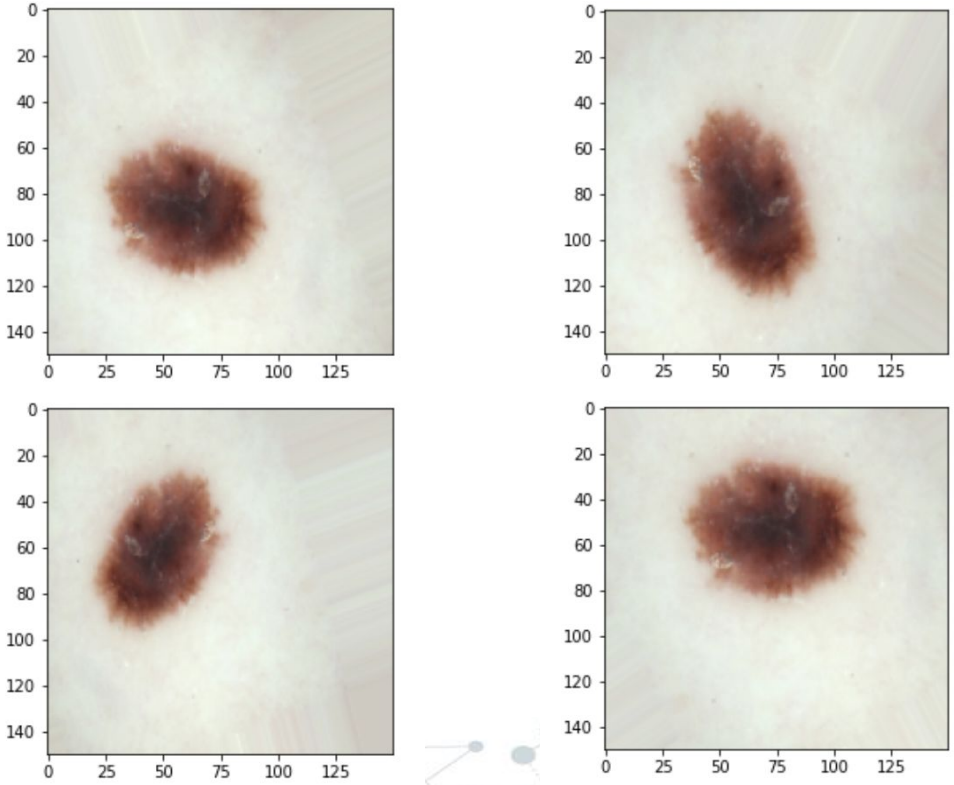
```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
      rotation_range = 40,
      width_shift_range = 0.2,
      height_shift_range = 0.2,
      shear_range = 0.2,
      zoom_range = 0.2,
      horizontal_flip = True,
      fill_mode = 'nearest')
```

You can create your own data generator with any specifications you'd like. The values chosen here are arbitrary.

You can check out the Keras documentation to see all of the available options and values each type of augmentation type can take.

Note that only your training data should be augmented - not the test or validation sets. The point of augmentation is to "increase" your training set size.

Data Augmentation in Keras

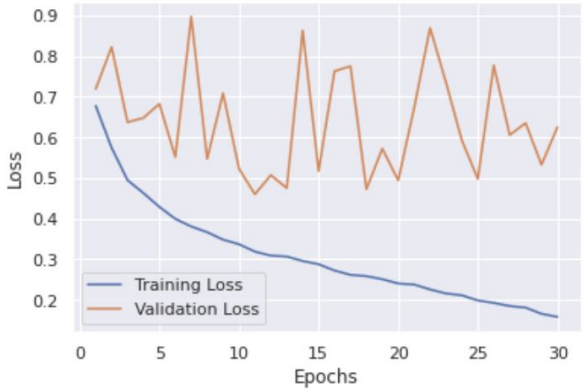


Back to the notebook

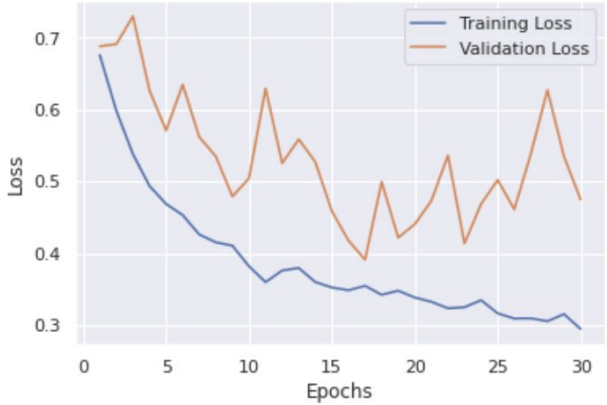
Skin lesions with data augmentation

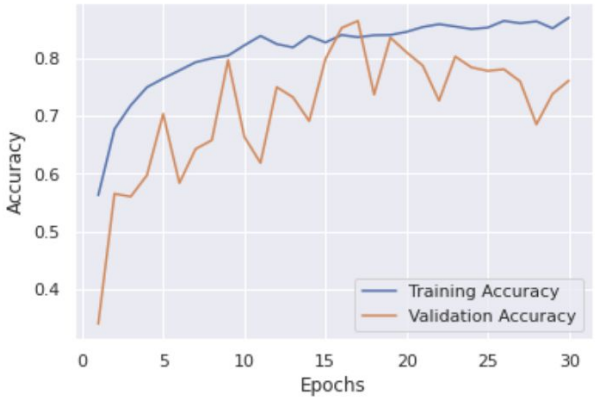
Data Augmentation in Keras

Without augmentation



With augmentation





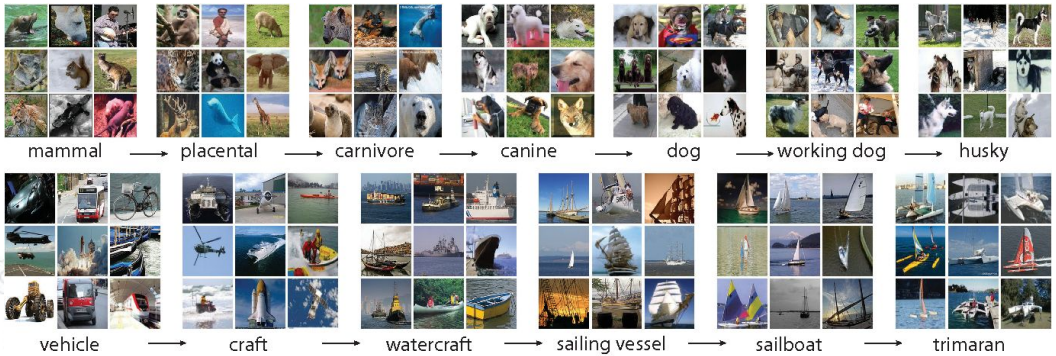
Pretrained Networks

Pretrained Networks

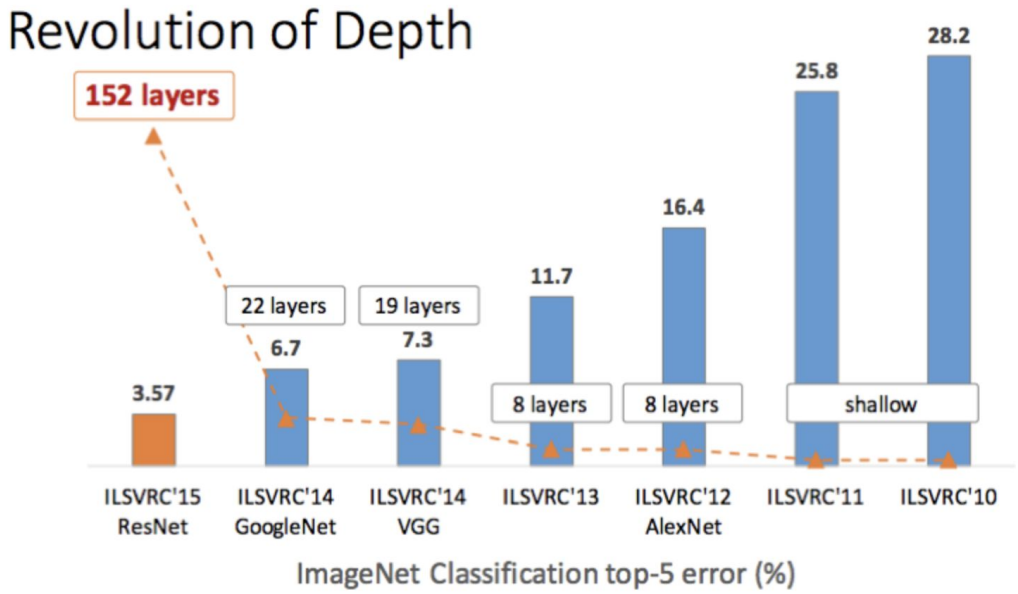
- Another way around having a small number of training examples to learn from is using networks that have been trained on other, bigger datasets similar to the type of data you have
- A **pretrained network** is a saved network that was previously trained on a large dataset
- If the dataset used to train the network is large enough and big enough, the features learned by the pretrained network can act as a generic model to use as a base for your network
- This saves an enormous amount of computing time
- Pretrained networks can be used for **feature extraction** and **fine-tuning**

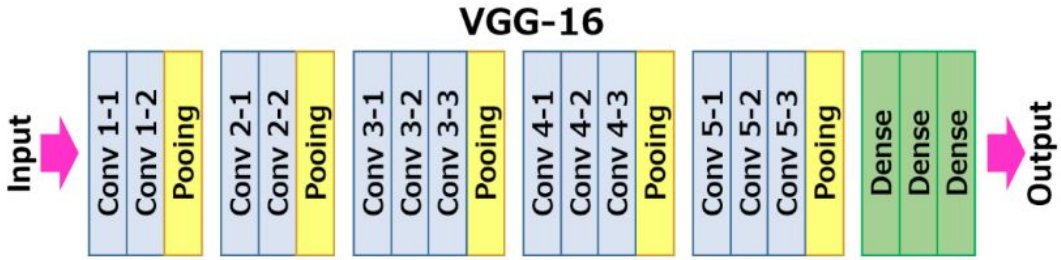
Pretrained Networks

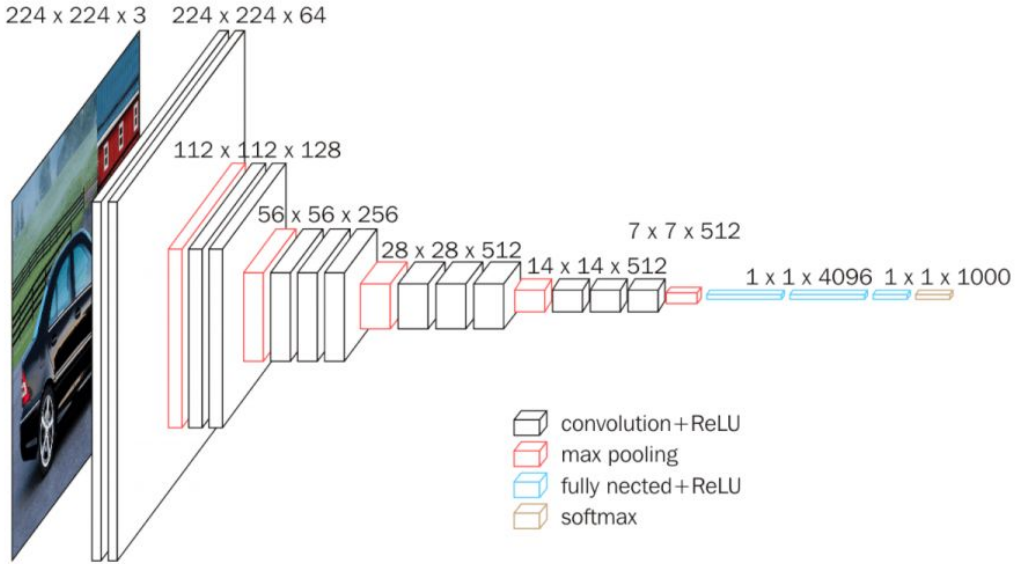
- Commonly used pretrained networks include
  - VGG16
  - ResNet
  - Inception
  - Inception-ResNet
  - Xception
- Commonly used dataset used to train a network is the [ImageNet dataset]
  - 1.4 million labeled images
  - 1,000 different classes
  - Mostly animals and everyday objects



Pretrained Networks

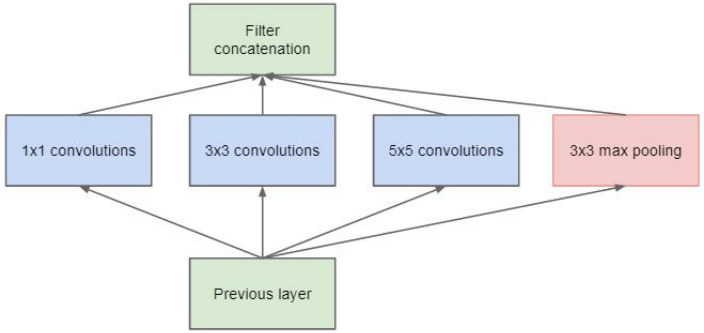




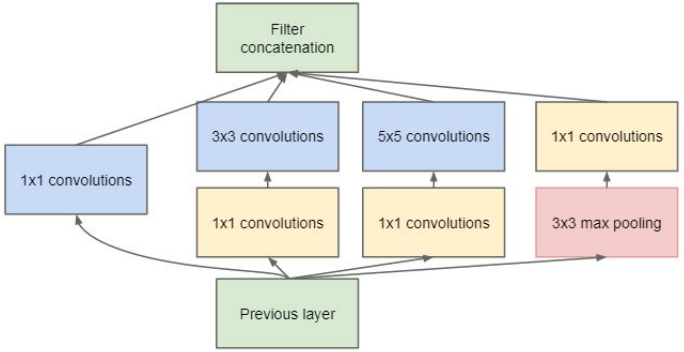


https://neurohive.io/en/popular-networks/vgg16/

Inception Models



(a) Inception module, naïve version

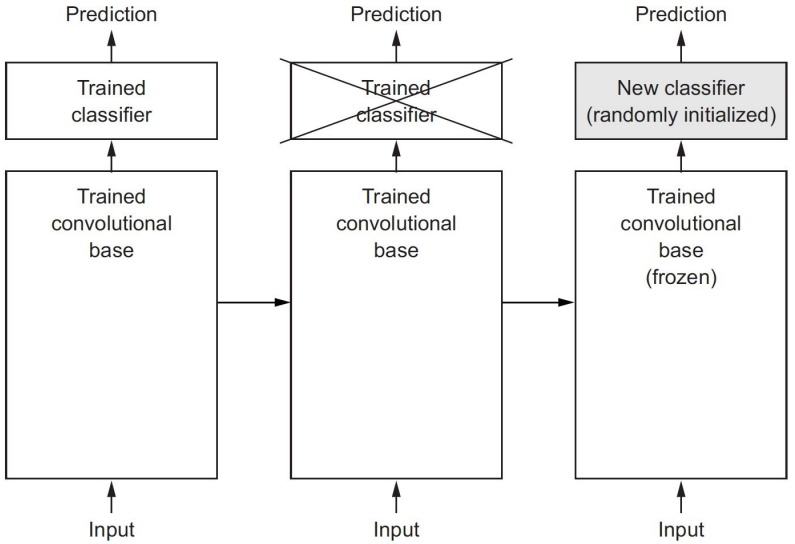


(b) Inception module with dimension reductions

https://becominghuman.ai/understanding-and-coding-inception-module-in-keras-eb56e9056b4b

Feature Extraction

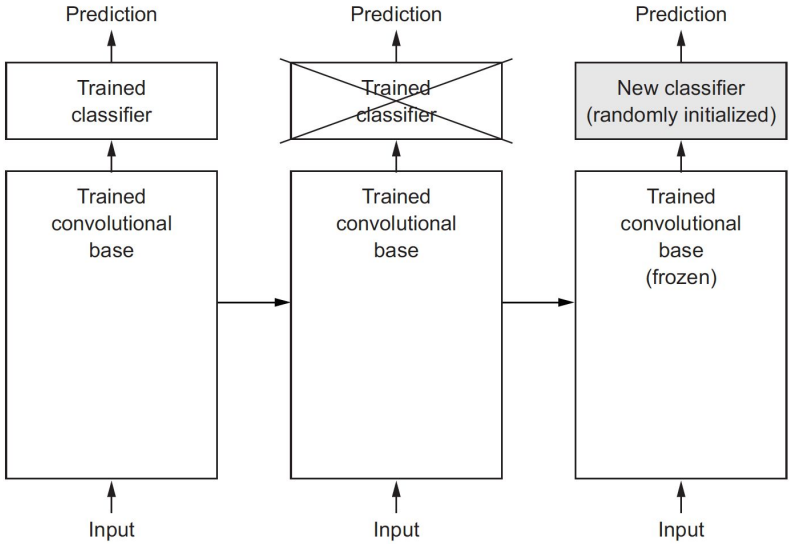
- Consists of using the representations learned by a previous network to extract features from new samples
- These features are then run through a new classifier that is trained from scratch, and predictions are made



Feature Extraction

- For CNNs, the part of the pretrained network you use is called the **convolutional base**, which contains a series of convolution and pooling layers

- For feature extraction, you keep the convolutional base of the pretrained network, remove the dense / trained classifier layers, and append new dense and classifier layers to the convolutional base



Feature Extraction

- We could also reuse the densely connected classifier as well, but this is not advised
- Representations learned by the convolutional base are likely to be more generic and thus more reusable
- The representations learned by the classifier will be specific to the set of classes the model was trained on
- They will also no longer contain information about where objects are located in the input image
  - This makes them especially useless when the object's location is important

Feature Extraction

- The level of generality depends on the depth of the layer in the model
  - Early layers extract local, highly generic features, i.e. edges, colors, textures
  - Later layers extract more abstract concepts i.e. “cat ear” or “dog eye”

- If your new dataset is very different from the dataset that was used to train the model, you should use only the first few layers for feature extraction rather than the entire base

Pretrained Networks in Keras

- Xception
- Inception V3
- ResNet50
- VGG16
- VGG19
- MobileNet

Instantiating the VGG16 Base

```
1 #from keras.applications import VGG16
2
3 conv_base = tf.keras.applications.VGG16(weights='imagenet',
4                                         include_top=False,
5                                         input_shape=(150, 150, 3))
```

```
conv_base.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

```
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Instantiating the VGG16 Base

```
1 #from keras.applications import VGG16
2
3 conv_base = tf.keras.applications.VGG16(weights='imagenet',
4                                         include_top=False,
5                                         input_shape=(150, 150, 3))
```

The final layer is a pooling layer and the final output shape from this base is (4, 4, 512). We need this information when adding layers to the base. This output shape will be the input shape for the densely connected layer we'll add to the base.

```
conv_base.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

```
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Using a Pretrained Network

- The final output has shape (4, 4, 512)
- You have 2 options:
  1. **Feature extraction without augmented data**: you can run the convolutional base over the dataset, record its output to a numpy array, and then use these values as input to a densely connected classifier
     - This is fast and cheap to run
     - It won't allow you to use augmented data
  2. **Feature extraction with augmented data**: you can extend the convolutional base by adding dense layers on top and running the whole model on the input data
     - This allows data augmentation
     - This is very computationally expensive

Feature Extraction without Augmented Data

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
        features_batch = conv_base.predict(inputs_batch)
        features[i * batch_size : (i + 1) * batch_size] = features_batch
        labels[i * batch_size : (i + 1) * batch_size] = labels_batch
        i += 1
        if i * batch_size >= sample_count:
            # Note that since generators yield data indefinitely in a loop,
            # we must `break` after every image has been seen once.
            break
    return features, labels

train_features, train_labels = extract_features(train_dir, 1609)
validation_features, validation_labels = extract_features(validation_dir, 426)
test_features, test_labels = extract_features(test_dir, 392)
```

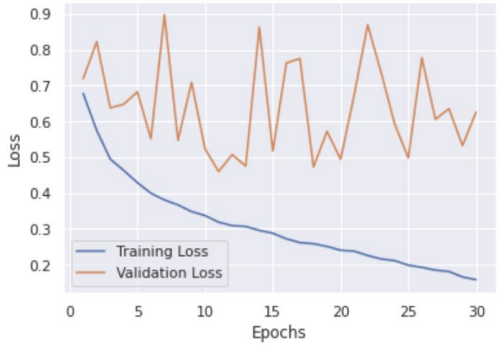
Colab notebook

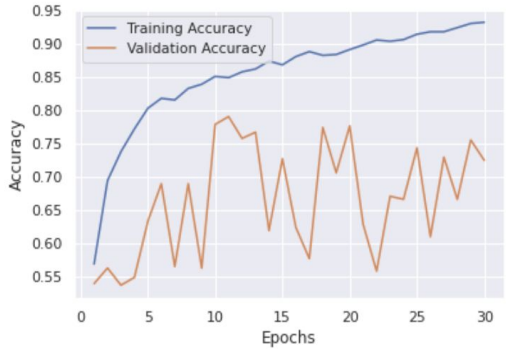
We need to reshape the outputs so we can feed them into a dense layer - recall that dense layers take in vectors.

```
train_features = np.reshape(train_features, (1609, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (426, 4 * 4 * 512))
test_features = np.reshape(test_features, (392, 4 * 4 * 512))
```

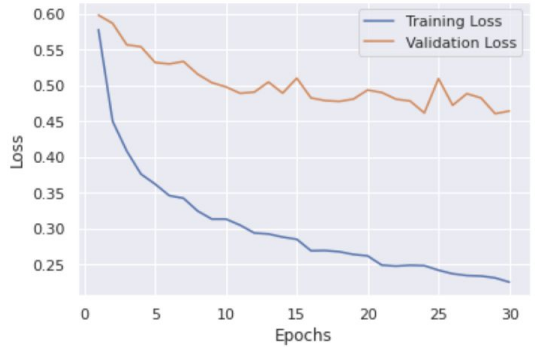
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(256, activation='relu', input_dim=4 * 4 * 512),
  tf.keras.layers.Dropout(0.5),
  tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=2e-5),
              loss='binary_crossentropy',
              metrics=['accuracy'])

history = model.fit(train_features, train_labels,
                    epochs=30,
                    batch_size=20,
                    validation_data=(validation_features, validation_labels))
```

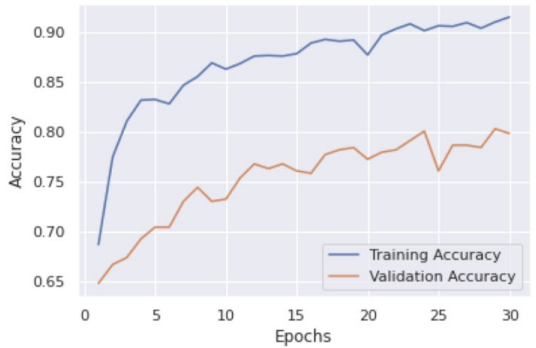
Original CNN made from scratch





CNN using pretrained base





Feature Extraction with Augmented Data

[Colab notebook]

```
model = tf.keras.models.Sequential([
  conv_base,
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(256, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense_2 (Dense)              (None, 256)               2097408
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 257
=================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

We can add the base just like a layer to our network

```
model = tf.keras.models.Sequential([
  conv_base,
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(256, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense_2 (Dense)              (None, 256)               2097408
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 257
=================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

```
print('This is the number of trainable weights '
      'before freezing the conv base:', len(model.trainable_weights))
```

```
This is the number of trainable weights before freezing the conv base: 30
```

```
conv_base.trainable = False
```

```
print('This is the number of trainable weights '
      'after freezing the conv base:', len(model.trainable_weights))
```

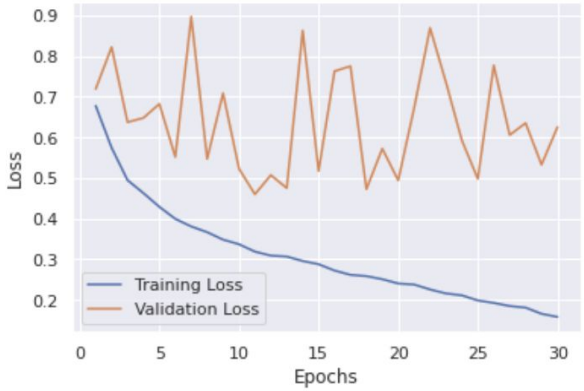
```
This is the number of trainable weights after freezing the conv base: 4
```

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')

# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=2e-5),
              loss='binary_crossentropy',
              metrics=['accuracy'])


history = model.fit(
      train_generator,
      steps_per_epoch=81,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=22)
```

Note: do not run this code without access to a GPU.

Back to [Colab notebook]

Original CNN made from scratch with data augmentation



CNN using pretrained base with data augmentation