EURO-BANKNOTES DETECTION

Using Convolutional Neural Network

Marco Morsiani Cassani - marco.morsiani4@studio.unibo.it

Introducing Convolutional Neural Networks

A breakthrough in building models for image classification came with the discovery that a convolutional neural network (CNN) could be used to progressively extract higher- and higher-level representations of the image content.
A CNN takes just the image's raw pixel data as input and "learns" how to extract these features, and ultimately infer what object they constitute.

To start, the CNN receives an input feature map: a three-dimensional matrix where the size of the first two dimensions corresponds to the length and width of the images in pixels. The size of the third dimension is 3 (corresponding to the 3 channels of a color image: red, green, and blue). The CNN comprises a stack of modules, each of which performs three operations.

1. **Convolution**

```
# Our input feature map is 150x150x3: 150x150 for the image pixels, and 3 for
# the three color channels: R, G, and B
img_input = layers.Input(shape=(150, 150, 3))

# First convolution extracts 16 filters that are 3x3
# Convolution is followed by max-pooling layer with a 2x2 window
x = layers.Conv2D(16, 3, activation='relu')(img_input)
x = layers.MaxPooling2D(2)(x)

# Second convolution extracts 32 filters that are 3x3
# Convolution is followed by max-pooling layer with a 2x2 window
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(2)(x)

# Third convolution extracts 64 filters that are 3x3
# Convolution is followed by max-pooling layer with a 2x2 window
x = layers.Conv2D(64, 3, activation='relu')(x)
x = layers.MaxPooling2D(2)(x)
```

The images that will go into our convnet are 150x150 color images (in the next section on Data Preprocessing, we'll add handling to resize all the images to 150x150 before feeding them into the neural network).
A *convolution* extracts tiles of the input feature map, and applies filters to them to compute new features, producing an output feature map, or *convolved feature* (which may have a different size and depth than the input feature map). Convolutions are defined by two parameters:

- **Size of the tiles that are extracted** (In my case I chose 3x3).
- **The depth of the output feature map**, which corresponds to the number of filters that are applied.

During a convolution, the filters (matrices the same size as the tile size) effectively slide over the input feature map's grid horizontally and vertically, one pixel at a time, extracting each corresponding tile.
During training, the CNN "learns" the optimal values for the filter matrices that enable it to extract meaningful features (textures, edges, shapes) from the input feature map. As the number of filters (output feature map depth) applied to the input increases, so does the number of features the CNN can extract. However, the tradeoff is that filters compose the majority of resources expended by

the CNN, so training time also increases as more filters are added. Additionally, each filter added to the network provides less incremental value than the previous one, so I aim to construct a network that uses the minimum number of filters needed to extract the features necessary for accurate image classification.

2. **ReLU**

Following each convolution operation, the CNN applies a Rectified Linear Unit (ReLU) transformation to the convolved feature, in order to introduce nonlinearity into the model. The ReLU function, F(x)=max(0,x), returns *x* for all values of $x > 0$, and returns 0 for all values of $x \leq 0$.

3. **Pooling**

After ReLU comes a pooling step, in which the CNN downsamples the convolved feature (to save on processing time), reducing the number of dimensions of the feature map, while still preserving the most critical feature information.

Max pooling operates in a similar fashion to convolution. We slide over the feature map and extract tiles of a specified size. For each tile, the maximum value is output to a new feature map, and all other values are discarded. Max pooling operations take two parameters:

- **Size** of the max-pooling filter (as typically, I used 2x2 pixels)
- **Stride**: the distance, in pixels, separating each extracted tile. Unlike with convolution, where filters slide over the feature map pixel by pixel, in max pooling, the stride determines the locations where each tile is extracted. For a 2x2 filter, a stride of 2 specifies that the max pooling operation will extract all nonoverlapping 2x2 tiles from the feature map.

Fully connected input layer (**flatten**) takes the output of the previous layers, “flattens” them and turns them into a single vector that can be an input for the next stage.
At the end of a convolutional neural network are one or more **fully connected layers** (when two layers are "fully connected," every node in the first layer is connected to every node in the second layer). Their job is to perform classification based on the features extracted by the convolutions. Typically, the final fully connected layer contains a ‘softmax’ activation function, which outputs a probability value from 0 to 1 for each of the classification labels the model is trying to predict.
In our case we have 2 fully connected layers:

1. The first fully connected layer takes the inputs from the feature analysis and applies weights to predict the correct label.
2. Fully connected output layer gives the final probabilities for each label, so for each banknote denomination (5,10,20,50).

We will stack 3 {convolution + relu + maxpooling} modules. Our convolutions operate on 3x3 windows and our maxpooling layers operate on 2x2 windows. Our first convolution extracts 16 filters, the following one extracts 32 filters, and the last one extracts 64 filters.

This is a configuration that is widely used and known to work well for image classification. Also, since we have relatively few training examples (1,250 for each class), using just three convolutional modules keeps the model small, which lowers the risk of overfitting.

Next, we'll configure the specifications for model training. We will train our model with the 'categorical_crossentropy' loss, because it's a categorical classification problem and our final activation is a '*softmax*'. We will use the rmsprop optimizer with a learning rate of 0.001. During training, we will want to monitor classification accuracy.

Data Preprocessing

Let's set up data generators that will read pictures in our source folders, convert them to float32 tensors, and feed them (with their labels) to our network. We'll have one generator for the training images and one for the validation images. Our generators will yield batches of 50 and 25 images of size 150x150 and their labels (binary), respectively.

Data that goes into neural networks should usually be normalized in some way to make it more amenable to processing by the network. (It is uncommon to feed raw pixels into a convnet.) In our case, we will preprocess our images by normalizing the pixel values to be in the [0, 1] range (originally all values are in the [0, 255] range).

```
# Flow training images in batches of 50 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
        train_dir,  # This is the source directory for training images
        classes=['5-euro','10-euro','20-euro','50-euro'],
        target_size=(150, 150),  # All images will be resized to 150x150
        batch_size=50,
        # Since we use categorical_crossentropy loss, we need categorical labels
        class_mode='categorical')

# Flow validation images in batches of 25 using val_datagen generator
validation_generator = val_datagen.flow_from_directory(
        validation_dir,
        classes=['5-euro','10-euro','20-euro','50-euro'],
        target_size=(150, 150),
        batch_size=25,
        class_mode='categorical')
```

Preventing Overfitting

A key concern when training a convolutional neural network is overfitting: overfitting happens when a model exposed to too few examples is so tuned to the specifics of the training data that it is unable to generalize to new examples. Two techniques to prevent overfitting when building a CNN are:

- **Data augmentation**: artificially boosting the diversity and number of training examples by performing random transformations to existing images to create a set of new variants. Data augmentation is especially useful when the original training data set is relatively small.
  In order to make the most of our few training examples, we will "augment" them via a number of random transformations, so that at training time, our model will never see the exact same picture twice. Let's add our data-augmentation to the preprocessing step.

```
# Add our data-augmentation parameters to 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=False) #we don't need to flip the images in our case.
```

(In our case we don't need to flip the images horizontally because during the testing we are not going to have the banknote flipped.
The validation data should not be augmented.

- **Dropout** regularization: Randomly removing units from the neural network during a training gradient step.
  Let's reconfigure our convnet architecture to add some dropout, right before the final classification layer. Dropout is usually between 0.2-0.5 values.

```
#Add a dropout rate of 0.5
x = layers.Dropout(0.5)(x)
```

Training

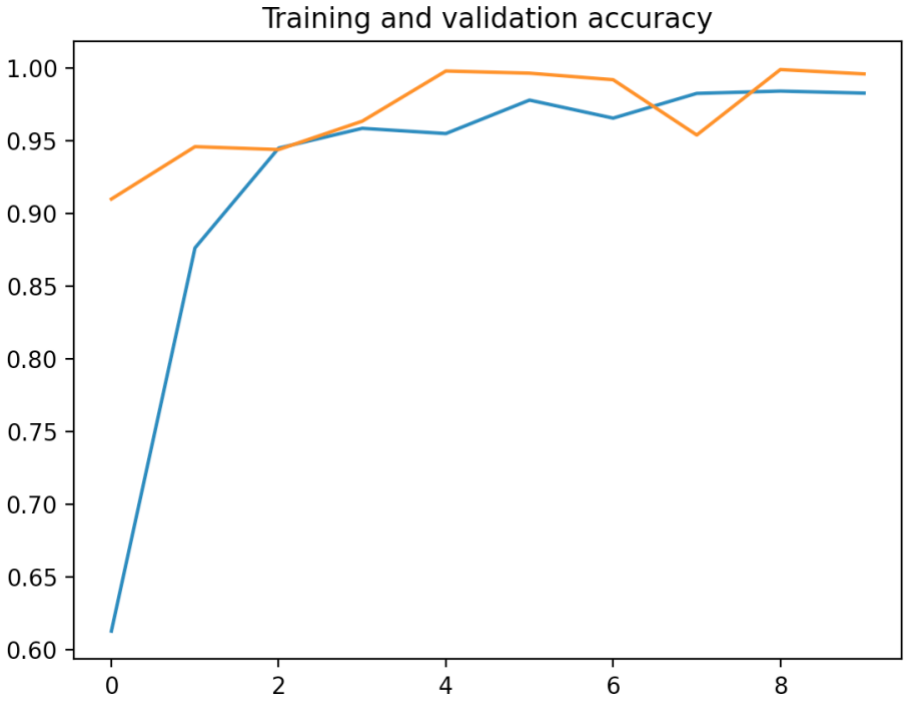
For each banknote, 1250 images were captured for the training dataset and 500 for the validation dataset, using a black background, trying to capture the photos with variable light to cover as many cases as possible: ambient light in the home, ambient light outside, directional light towards the banknote. Obviously I tried to capture images from different angles, positions and distances for greater accuracy in real time testing.
Let's train on all 5,000 images available, for 10 epochs, and validate on all 2,000 validation images. The training of the model took about 30 minutes.

```
# Create model:
# input = input feature map
# output = input feature map + stacked convolution/maxpooling layers + fully
# connected layer + softmax output layer
model = Model(img_input, output)

# or categorical_crossentropy
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(lr=0.001),
              metrics=['acc'])

history = model.fit(
      train_generator,
      steps_per_epoch=100,  # 5000 images = batch_size * steps
      epochs=10,
      validation_data=validation_generator,
      validation_steps=80,  # 2000 images = batch_size * steps
      verbose=2)

model.save('euro-CNN.h5')
```

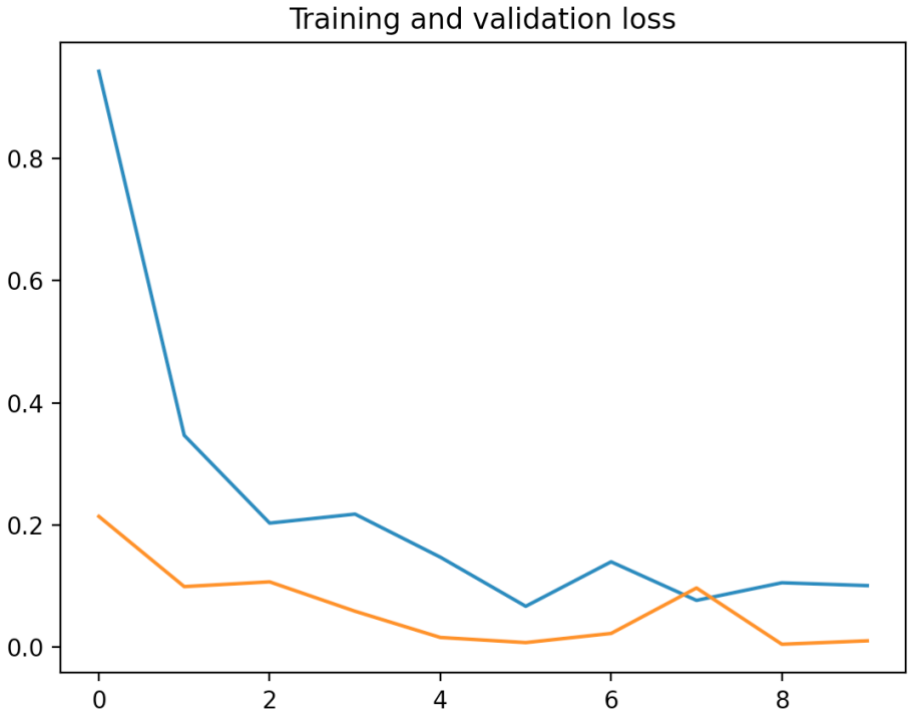
We save our model in 'h5' format so that we don't have to recompile it every time we run the program. We will use another file to load the model and use it to identify the banknotes.

Evaluate the Results

Let's evaluate the results of model training with data augmentation and dropout.



We can see from the graph that the accuracy of the validation dataset (orange) is very high, about 99%. Also the accuracy of the training set (blue) is about 97%.



We can see from the graph that the loss of the validation dataset (orange) is very low, about 1%. We can say also that the loss of the training set (blue) is low: about 12%.

Testing the Model

I created another python file to test our model on real time video via webcam. The program will split the video into frames and make the predict on the latter and then render the video in out with an inscription at the top left indicating the value of the recognized banknote.
Before making the predict on each frame, the images has been modified in scale to make them the same size set in the model (150x150) and normalized the pixel values to be in the [0, 1] range.

Conclusion

One of the difficulties encountered in this project was to create an appropriate training dataset. At the beginning I started with images downloaded from the internet depicting the banknotes, but when I went to test my model in real time from my webcam, the accuracy was quite low because this type of technology depends a lot on the type of images passed to the model in the training phase. And that was when I decided to create a dataset with images taken directly from my webcam, trying to cover various light situations and positions.
I also decided to use a black background for the dataset because my hands, my face and the same background for each image reduced enough the accuracy of the model. Obviously the program also worked on such training, but the prediction of the banknote value was wrong for a good percentage of times.
As regards the hyper-parameters of the model, quite standard values have been used for a type of training that does not contain a large number of images (normally you can have tens of thousands of images on which to do it).