Vandalized Traffic Sign Image Classification Using CNN

Zhen Liu

Lawrence Technological University

Computer Science

Spring 2018

Faculty advisor: Prof. CJ Chung

Introduction

CNN/ConvNet, Convolutional Neural Network, is a category of Neural Networks that have proven to be very efficient in areas such as image recognition and classification. The architecture of a CNN is designed to allow convnets to effectively learn increasingly complex and abstract visual concepts by learning spatial hierarchies of patterns of an input image. Also, patterns learned by a CNN are translation invariant, which means convnets can process images more efficiently since they need fewer training samples to learn representations that have generalization power. Due to these benefits, ConvNets are used in my project to test whether a powerful image classifier can be built to successfully identify traffic signs using very few training examples, only a few hundred images for each class.

Image Data

The dataset used for this project is collected by Dr. CJ Chung. It comprises of more than 1,200 images in total, divided into 4 classes: One_Way, No_Turns, Road_Closed and Stop, taken by three different cameras and in different formats. To test whether gradual, incremental learning, that is from easy to hard, from simple to complex, is a way to improve learning accuracy, the images are taken in three different forms: simple close-ups, vandalized images, and difficult images taken from various angles and under different lighting conditions. The idea is to allow the network to easily extract features from close-up images and apply these features to train more challenging images.

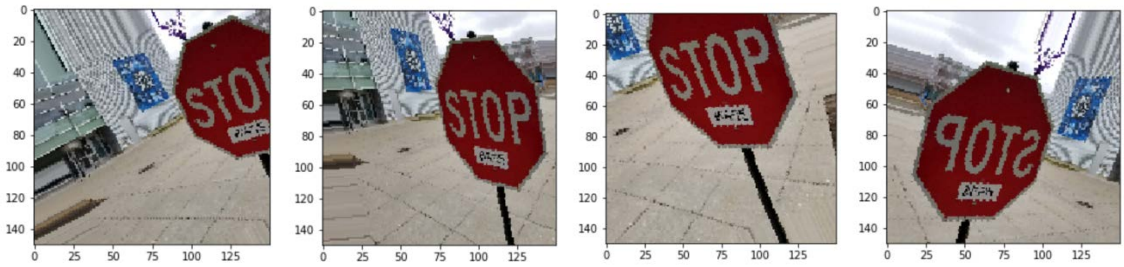
Training Image Examples

Images in each class are further split into two sets: one for training, one for validation to evaluate the models. Since the dataset is unbalanced: there is not exactly equal number of instances in each class, the training and validation sets were split into the ratio of 2: 1. There is an average of 200 images in training set for each class, an average of 100 images in validation set for each class. This is a very small dataset to learn from for classification problems. So this is a challenging deep learning problem, but also a most common use case in real world.

In order to make the most of the small training set, data augmentation is used via a number of random transformations that yield believable-looking images so that at training time, the model would never see the exact same image twice. This helps expose the model to more aspects of the data and generalize better. In Keras, this can be done via using ImageDataGenerator class.

```
from keras.preprocessing.image import ImageDataGenerator

# Augmentation configuration used for training
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')

# Validation data should not be augmented, only rescaling.
test_datagen = ImageDataGenerator(rescale=1./255)
```

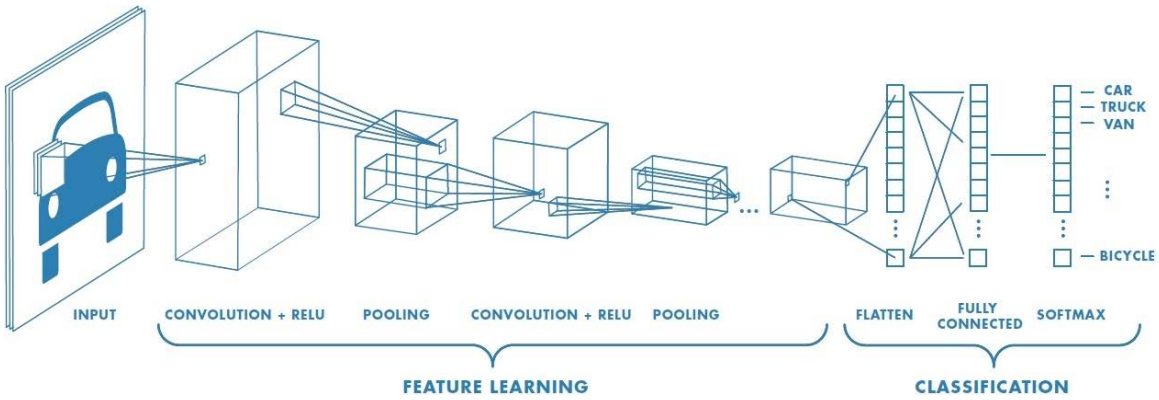
Here is how randomly augmented training images look like:



To test the trained model, 10 unseen images are given by Dr. Chung, our professor, to tell how well the model performs on data it has never seen before. The goal is to get good generalization. Of course we cannot control generalization, but we can adjust the model based on its training and validation data.

Model Architecture

Convolutional Neural Networks (CNNs) consist of multiple layers designed to require relatively little pre-processing compared to other image classification algorithms. They learn by extracting patches, for example 3 × 3, from its inputs and sliding these windows of size 3 × 3 over the input image and then by using filters to allow the network to identify certain patterns in the image. The deeper we go into the layers of the Neural Network, the more complex patterns the network will continually learn.



An example of a CNN Layer Architecture for Image Classification

Train from scratch

Since only few training examples are available in my project, my number one concern should be overfitting. Overfitting happens when a model that is exposed to too few training data begins to learn patterns that are specific to the training data but that are misleading or irrelevant when it comes to new data. Data augmentation is one way to fight overfitting, but it's not enough since the augmented images are still correlated to original ones. So the main focus to fight overfitting is the capacity of the model: how much information to store in the model. If too many information, the model has the potential to store irrelevant features although it may be more accurate by leveraging more features; if just a few information, the model may focus on the most significant feature in the data and therefore generalize better. This requires me to find a compromise between too much capacity and not enough capacity. To be more exact, to choose the right number of parameter in my model. After evaluating an array of different architectures, I found the appropriate model size for my data. The following is my first model.

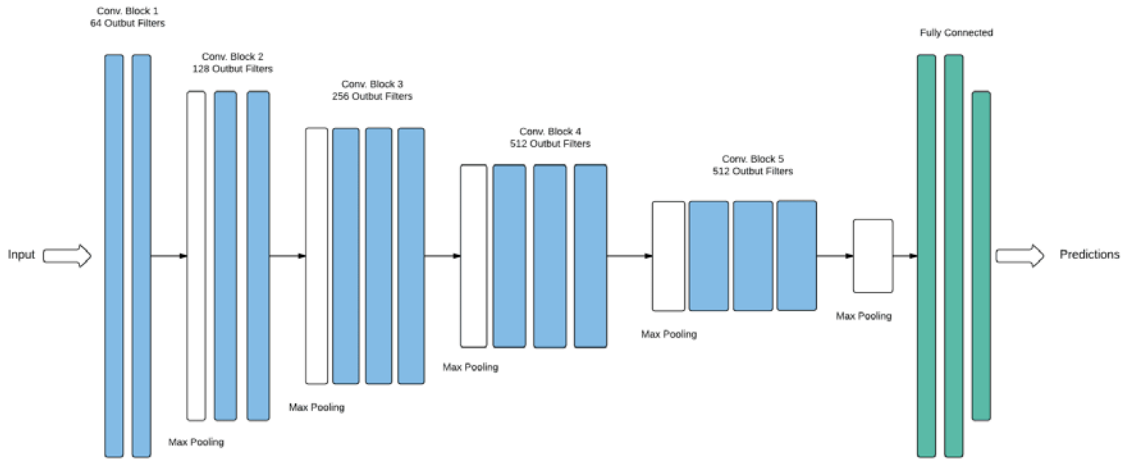
```
from keras import layers
from keras import models
from keras import regularizers

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=input_shape))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.5))
model.add(layers.Flatten())
model.add(layers.Dense(64,kernel_regularizer=regularizers.l2(0.001),activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(4, activation='softmax'))
```

A stack of four convolution layers with a Relu activation is used, followed by max-pooling layers. On top of it two fully-connected layers are added. At the end of the model is a 4-unit layer and a softmax activation, which is good for a categorical classification. Meanwhile, dropout is added to introduce noise in the output values of a layer, in other words, to disrupt the random correlations in the data. Weight regularization is also used to force the model to take smaller weights. This model gives a validation accuracy of 89% after 40 epochs.

Transfer Learning

To further improve accuracy on my image classification problem, I try to use a pertained model. Such a model acts as a generic model of the visual world, since it has already learned features that prove to be useful for most of computer vision problems, and leveraging such features would allow me to reach a better accuracy. I have tried the VGG16 and ResNet50 architecture, and it turns out that VGG16 works better on my traffic sign image classification problem.

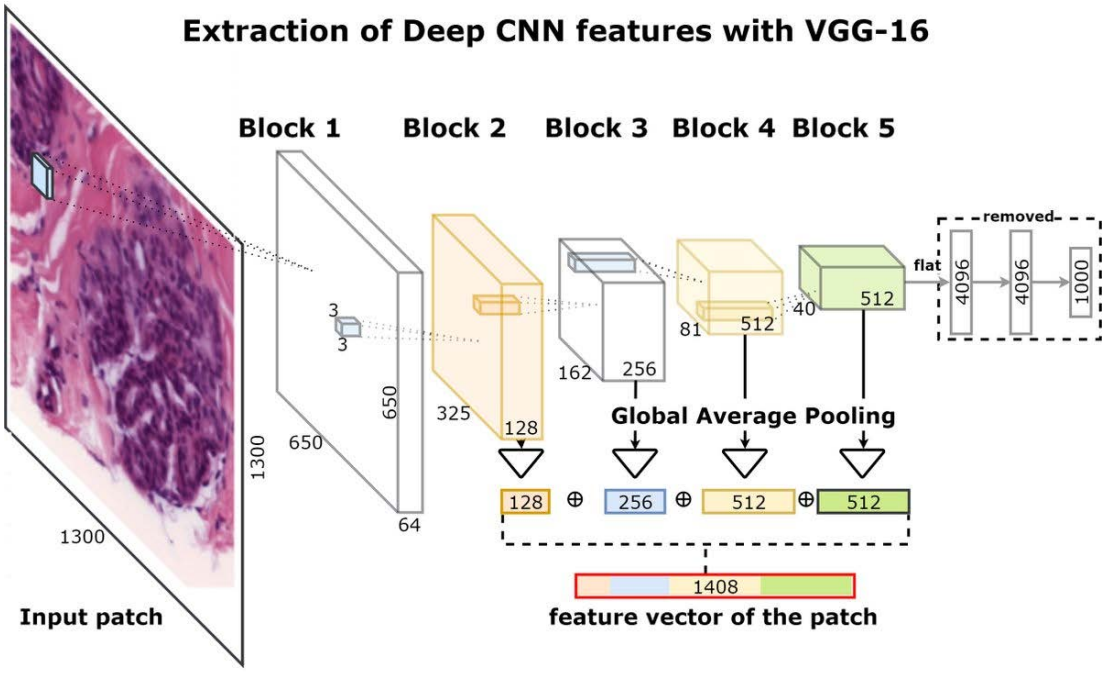
Here is how the VGG16 architecture looks like:



VGG16 Net Architecture

Feature Extraction

My strategy to use the pertained network is first to do feature extraction, and then fine-tuning. Feature extraction is to use the presentations learned by a previous network to exact features from new data samples, and then run these features through a new classifier. In other words, I will only instantiate the convolutional base of the model, everything up to the fully-connected layer, and then run this model on my new data to extract interesting features from traffic signs, and train a new classifier on top of these features.



Instead of running the convolutional base on my dataset, recording the output to Numpy arrays on disk, and then training a small fully-connected model on top of the stored features, I choose to extend the convolutional base model by adding Dense layers on top, and run the whole thing to end on the input data. Although this approach will not be computational efficacy, far more expensive than the first technique, it allows me to use data augmentation during training.

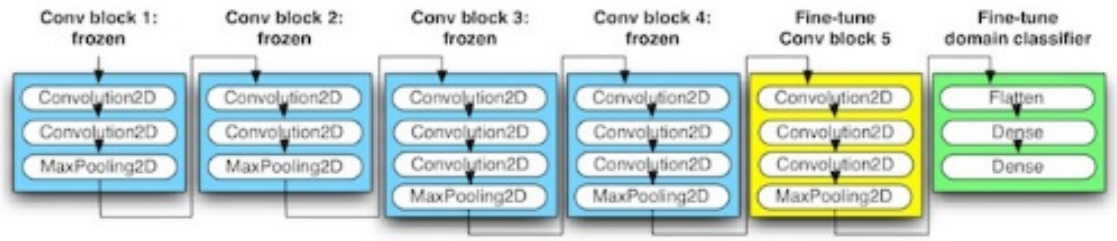
Add a densely connected classifier on top of the convolutional base:

```
from keras import models
from keras import layers

# Build a classifier model on top of the convolutional base
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(4, activation='softmax'))
```

I reach a validation accuracy of 97.8% after 30 epochs, pretty good compared to what I have achieved by training a small convent from scratch.

Fine Tuning

To further improve my accuracy, I try to use fine-tuning, another technique complementary to feature extraction, to the last convolutional block of the VGG16 model alongside the top-level classifier. When doing feature extraction, I have instantiated the already-trained convolutional base of VGG16 and added my custom fully-connected model on top. To do fine-turning is to build upon the previous work, and then to freeze the layers of the VGG16 model up to the last convolutional block or, in other words, the last three convolutional layers, and jointly train both these layers and the part I have added.



Freezing all layers up to conv block 5:

```
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

Fine-tuning the model. I will do this with RMSProp optimizer with a very low learning rate to limit the magnitude of the updates to the previously learned features of the three layers I am fine-tuning, so as not to harm these learned features.

```
from keras import optimizers
from keras.models import load_model
import numpy as np

# fine-tune the model
model = load_model('traffic_sign.h5')

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

history = model.fit_generator(
          train_generator,
          steps_per_epoch=50,
          epochs=10,
          validation_data=validation_generator,
          validation_steps=25)
```

Finally I reach a validation accuracy of 99.2% after 10 epochs.

Prediction on New Images

Now it’s time to evaluate this trained model on the never seen test images.

I a

```
from keras.models import load_model
from keras.preprocessing import image
import numpy as np
from keras import optimizers

# dimensions of our images
img_width, img_height = 150, 150

model = load_model('/model/traffic_sign_finetune.h5')
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])

# predicting images
img_path = '/traffic_sign/test'
img = image.load_img(img_path, target_size=(img_width, img_height))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x /= 255
```

I am given 10 images to predict the classes of each image and probabilities of each class. Finally, I have got 100% accuracy.

Conclusions

From this project, CNN is once again proven to be an extremely powerful tool in computer vision, turning tasks that appear very hard, or even impossible, into reasonable ones. Specifically, model with pre-trained weights and fine-turning proves to be incredibly effective, especially when little training data is available. However, if I train different models, it is possible that they pick up slightly different things, and therefore combining their outputs may further boost accuracy. Or poor models may not make up for a strong model, ensembling models may reduce final accuracy. This is the topic I will explore in my next project.