Object Detection Pipelines

Andrei Constantin Apostol

Coordinator: Lect. Dr. Anca Ignat

Faculty of Computer Science
Alexandru Ioan Cuza University
July, 2018

Abstract

The problem of detecting and classifying objects has seen a boost in accuracy as well as performance speed in recent years, due to the development of deep learning techniques such as YOLO, Single-Shot-Detectors and Faster R-CNN. These methods, however, are notorious for their ravenous appetite for data and compute power; they also require datasets with labeled bounding boxes, as well as the class represented in the box, which are still scarce and are difficult to build manually.

In this paper, we explore a way of building a system that performs detection without having access to a dataset equipped with labeled bounding boxes, and using a relatively smaller amount of computing power. This method involves building a convolutional neural network to discern between object classes, and a binary classifier to discriminate between background and objects of interest.

In the $2nd$ and $3rd$ chapters, we describe how we have built these systems, justifying and explaining how the hyperparameters have been chosen along the way. We have used well established techniques, as well as new experimental insights that have been generated in the training process. The $1st$ chapter is reserved for how we have brought the traffic sign dataset into a form from which the neural network might learn best, as well as how we generated a background dataset, which is required for the detection phase.

While we have framed this task as a general object detection problem, we have chosen a particular dataset consisiting of road signs due to its relevance in the automotive industry, with driver assistance or autopilot technologies emerging into the consumer market.

Personal Contributions

In the development of this project, I have had several contributions, one of them being the preparation of the datasets. In this phase, several different operations have been applied to the data in order to bring it to a state that can yield better results. Some of these operations are data augmentation, format conversion, resizing and adding artificial variance in order to improve the model's ability to generalize. Class balancing has been applied to the dataset by adding a weight to each class and rewarding/penalising the model proportional to the class' respective label.

The main responsibility has been implementing and training the machine-learning algorithms over the aforementioned datasets after they have been passed through the preparation phase. For this step, I have researched and chosen the models that would best fit the data, and ran hyperparameter optimization to maximize the performance of the models. Most of the parameter choices have solid justification, but, due to the open-research nature of this field, some have been determined experimentally. In doing so, some new and previously undocumented insights have been generated, particularly with the combination of optimizer and learning schedule, allowing an increase in the number of training epochs before the model overfits, thus reducing the error rate by a considerable amount. This also allows us to use a more aggressive learning rate without having the risk of the model diverging.

I have also researched and implemented various detection techniques that can be used to create an object detection and classification pipeline using the trained neural network, and compared their performance. This has been done without access to a dataset labeled with bounding boxes. By extending our neural network into such a system it can then be used into an autonomous vehicle's driver assistance module (e.g. a car's dashboard camera).

Contents

Chapter 1

Dataset Preparation

The dataset used for training the convolutional neural network is the German Traffic Sign Recognition Benchmark.[10] This dataset consists of 43 classes of traffic signs, with one directory per class. It contains 26,640 training images and 12,569 testing images. Each image contains one traffic sign each with approximately 10% border around it. The traffic sign is not necessarily centered within the image. Sizes of each image may vary, from 15x15 up to 250x250.
Each sign has a class index, ranging from 0 to 42. Training images are grouped by tracks. Each track contains 30 images of one single physical traffic sign in various lighting conditions, angles and distances. We will need to take this fact into account when building our validation set. Here are a few samples from the dataset:

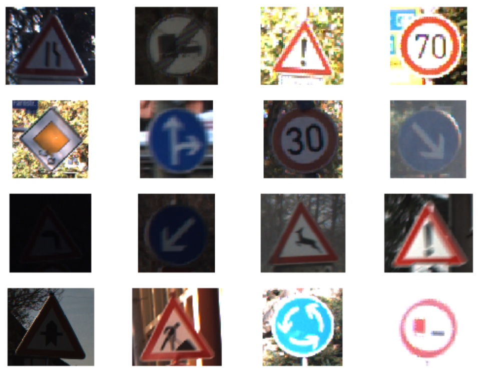


Figure 1.1: Samples from the training set.

The dataset is already partitioned with approximately 70% of the total images in the training set and the remainder of 30% into the test set. The images come in the PPM (Portable PixMap) format and are stored in memory in the form of $H * W * 3$ arrays, where H is the height of the picture, and W is the width, each with 3 channels of colors in the RGB colorspace.

Balancing

A balanced dataset is one where each class has roughly the same number of points. While this is ideal, in practice it is rarely the case. Many machine learning models, including neural networks, tend to learn more from overrepresented classes. As such, a skewed or unbalanced distribution can affect the model's performance and ability to generalize. By plotting the number of samples in each class as a bar chart (class index on x-axis, number of samples on y-axis), we obtain the following figure:

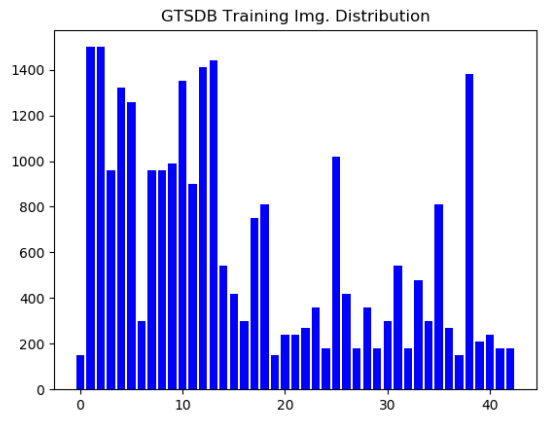


Figure 1.2: Distribution of samples per class

The training set seems to be severely unbalanced, with some classes having 10 times more samples than the most underrepresented class. This can cause our model to learn more about the overrepresented classes since more gradient descent updates are performed, tilting its parameters in a direction that favours those particular signs, thus being unable to create an internal representation of those with a low number of samples. Unbalanced datasets are a relatively common problem in machine learning, so several solutions have been proposed in official literature. We have chosen a method called class weighting. The first step is to compute each class' inverse ratio relative to the class with the highest number of samples (also called reference class):

$$m = \max_{0 \leq i \leq 42} (|s_i|)$$
$$r_i = \frac{m}{|s_i|},$$

where s_i is the set representing the data points in class i.

We can then use each class' ratio as defined above to weight the neural network's loss function during training, effectively causing the model to perform gradient descent updates proportional to the ratio. This means that for classes with a small number of data points, updates are larger. For the reference class, the values of the updates do not change, since the ratio is 1. This is mathematically equivalent to feeding duplicates from the smaller classes into the network.
Note that there is no need to balance the test and validation sets since they play no part in the learning process itself, and are only used for benchmarking.

Building Validation Set

Validation sets are used to estimate how well a model has been trained. Specifically, we can use it to train several models using different hyperparameters and/or learning strategies and compare their results. Once we have chosen the best performing model, we can check it against the test set. This is done to ensure that the choice of parameters is as unbiased as possible; choosing a model for its results on the test set does not necessarily imply its ability to generalize well.
Since the images in our dataset are grouped by tracks, we cannot include pictures from the same track into the training and the validation set, as it would damage the quality of our key performance indexes (our model would be tested on images that are too similar to the ones it has been trained on). Therefore, two full tracks have been randomly selected from each class and have been moved into the validation set. This number has been chosen as to keep the quality of our initial dataset (some classes only have five tracks) and build a sufficiently large validation set for the purpose of testing.
By doing this, we build a set with 1,290 images, which we have used when exploring how the choice of some hyperparameters affects our model's performance, and also as a metric for creating checkpoints. This process is explained in-depth in the 2^{nd} chapter.

Augmentation

Although the pictures in our dataset are taken at varying distances, lighting conditions and/or angles, it is not enough for our system to generalise. Also, the border around the signs is rather small (around 10% of the image's dimension), allowing only for a thin margin of error when used in conjunction with a detection system. As such, more variance is needed in our pictures. For this task, two main methods have been proven to work best: getting more samples (in our case, collecting more pictures of traffics signs), and artificially inflating our existing set (also known as augmentation).
Increasing the size of our dataset by collecting more images causes our distribution to be more representative, and variation occurs naturally. However, this process is very costly and, in some cases, not even possible. As such, we have opted for the second option, augmentation.
We augment our images by using an ImageDataGenerator object provided through the Keras API. We define it as follows:

```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
            rescale=1./255,
            rotation_range=10,
            width_shift_range=0.2,
            height_shift_range=0.2,
            shear_range=0.2,
            zoom_range=0.2,
            fill_mode='nearest')
```

The ImageDataGenerator object defines a set of transformations to be applied to each image during the training process. A random float is chosen for each category defined by us, in the $[-x, +x]$ range, where x is the maximum transformation intensity set by us.
In the particular object defined above, images are shifted by height and width, zoomed into, and sheared by up to 20% each (values chosen independently). The rotation range is ± 10 degrees. Rescaling by 1./255 means dividing the intensity of each pixel by 255, squashing their values in the $[0, 1]$ range, effectively turning the image to grayscale. During runtime, each parameter is chosen randomly and independently, and the transformation is applied to it before being passed on, so the neural network does not see the same image twice. Here is a preview of a set of transformations applied to a single image:

Figure 1.3: Original image.

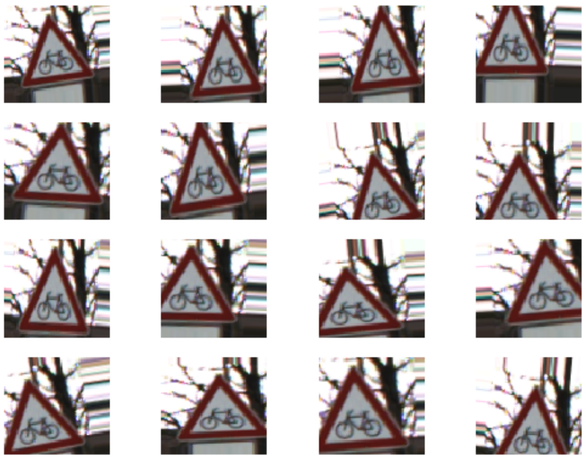


Figure 1.4: Several possible transformations applied to the image.(rescaling not included)

The process of augmentation can be seen as a form of regularization (the network's internal representation has to account for a higher degree of variance in the samples). The exact values when defining the generator have been chosen as a middle ground between adding a negligible amount of variance and regularizing the model to the point where its performance is affected. Rescaling (converting to grayscale) has shown, counter to intuition, an improvement in overall accuracy, and is also useful in keeping the numbers (e.g. variables in the backpropagation algorithm) on the low side.
Note that augmentation has to be turned off when performing hyperparameter optimization since we want to have as little randomness as possible when running our experiments, and only use it once we have established a good baseline for our model.

Building the background dataset

Some of the object detection pipelines require having a labeled set for binary classification (sign/background). As such, we built a dataset of images containing background (which have no sign in them). We have done so by using the GTSDB[9] set. It contains 900 images taken from a car's dash camera, each containing at least one traffic sign.

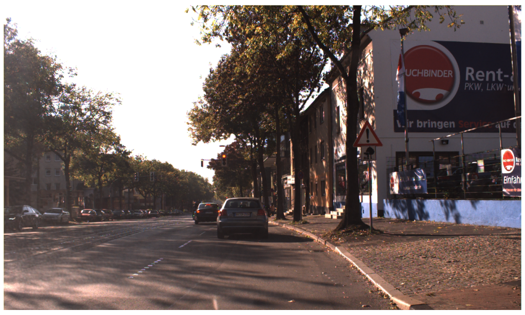


Figure 1.5: An image from the GTSDB dataset containing the Danger sign

For each image, we run a sliding window (size chosen randomly from a predefined set) and choose a number of samples from the total number of windows generated. The sizes as well as the number of images in the set have to roughly match those in the traffic sign set to ensure balance and scale when doing classification.

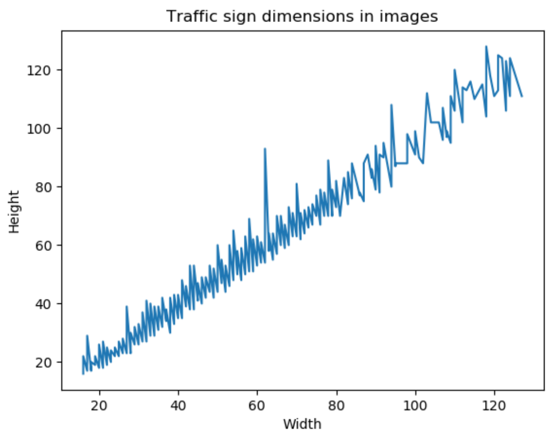


Figure 1.6: Plotting the sizes of the traffic sign images.

The images seem to be almost of square shape, judging by the plot above. As such, we define seven possible scales that are chosen from when doing sliding window: $(32, 32), (48, 48), (64, 64), (80, 80), (96, 96), (112, 112), (128, 128)$

To match the size of the traffic sign set, we should choose only 45 samples per image (since $\frac{40000}{900} \approx 44,44$). We have chosen a slightly higher number (65) since the dataset has to be curated manually, removing any traffic signs that happen to appear and images that are too similar (pictures of the sky from the same patch, walls, branches of a tree, etc.). Images 0-700 have been used to generate the training set, and images 701-900 for the test set. After curating the dataset, we end up with aprox. 29000 images in the train foleder and 15000 in the test folder. We generate the validation set by choosing 3000 images from each one and moving it into the validation folder. Here are a few samples of the generated images:

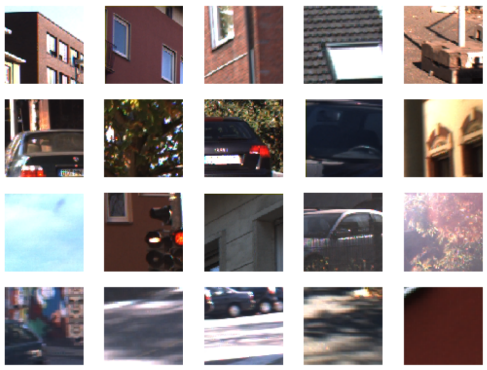


Figure 1.7: Samples taken from the generated background dataset.

This dataset will be useful when building detection systems such as binary classifiers. In other words, background pictures will be labeled as 0, and images of traffic signs will be labeled as 1. Such a system would be used by running a sliding window over a picture and, for each window (or patch), run the classifier to first determine whether or not it is a sign. If the classifier outputs 1, we run the patch through the convolutional neural network to perform classification as well. Finally, a bounding box is added to the coordinates of the patch containing the sign, annotated with its type. Non-max suppresion and bounding box regression may also be applied.

Chapter 2

Training the Convolutional Neural Network

For our pipeline to predict what type a particular sign is, it requires a classifier. For this task, we have chosen to implement and train a convolutional neural network (also known as CNN or convnet). In this chapter, we will briefly discuss what is a convolutional neural network and then present our way of applying it to pictures of traffic signs, detailing and explaining the choices in hyperparameters and learning strategies.

What is a CNN?

Convolutional Neural Networks, as defined by LeCun et al.[1] is a type of feed-forward neural network inspired from biology in that the connection between neurons closely resembles that of the animal visual cortex. They are mostly used in machine learning for visual recognition tasks or natural language processing, and offer a suite of benefits over their counterparts, such as shift and translation invariance. They are similar to regular feed-forward neural networks in that they have a fixed input size, multiple hidden layers and an output layer which performs classification. However, the connections between neurons are local (a neuron in a layer is only connected to a small region of the previous layer), allowing for greater scalability when dealing with images. Next we will present the layers specific to this type of neural network.

Convolutional Layers

Images can be represented as 3-dimensional arrays defined as $height * width * depth$ (where depth is the number of color channels). A convolutional layer simply receives an input volume and passes on an output volume. It is defined by four parameters: width, height, number of filters and stride. During the forward pass, each filter is slided across the input volume (over width and height) and a dot product is computed between the values of the filter and the input volume's patch. Each dot product generates a two-dimensional activation map representing the response of that filter for that particular region. By stacking each resulted activation map along the *depth* dimension, we produce an output volume. Intuitively, this process increases the depth of the volume, but reduces the height and width.

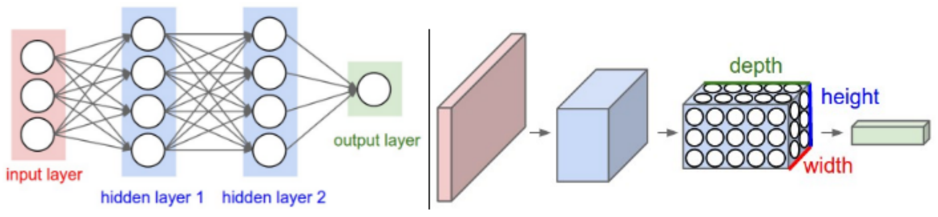


Figure 2.1: Left: Regular neural network. Right: Convolutional Neural Network with several convolutional layers stacked together.
Each filter has its own set of weights that are updated (learned) when running backpropagation. Note that a fully-connected neural network would require several orders of magnitude more connections between layers causing it to not scale well (e.g. for a $30*30*3$ image, each neuron in the first hidden layer would have to have 2700 weights. Adding up the weights for each neuron in the hidden layer, this number gets out of control rapidly).

Pooling Layers

Pooling layers have a similar purpose to that of the convolutional layers, in that they reduce the spatial size of the input's representation. They are defined by three parameters: height, width and stride. They work by sliding a $h * w$ window over the input volume, independently for every depth slice, and applying a single-output operation over the window. More generally, they produce an output volume $W_2 * H_2 * D_2$ defined by

$$W_2 = \frac{W_1 - w}{S+1}$$
$$H_2 = \frac{H_1 - h}{S+1}$$
$$D_2 = D_1$$

where $W_1 * H_1 * D_1$ is the input volume, S is the stride and w, h are the width and height of the sliding window.
Note that the depth width and height of the resulting volume are reduced, and the depth remains unchanged.
Most common operations used when pooling are MAX, MIN and AVG. In our final architecture, we have chosen the MAX function, hence the layer's name Max Pooling. A layer of pooling is typically added after every convolution.

Architecture

We defined a network with three sequences of convolution and max pooling layers, followed by a flattening layer (so that the output is compatible with the following layers), dropout and two fully connected layers, the last one being responsible for outputing the classification. The network architecture is summarised in the table below (generated by calling the *model.summary()* method). Before being passed to the network, images are resized to $32 * 32$ pixels each. Note that activation, dropout and flattening layers have not been included for the sake of brevity.

Layer (type)	Output Shape	No. of params
Conv2D-1	(30,30,32)	896
MaxPool2D-1	(15,15,32)	0
Conv2D-2	(13,13,64)	18496
MaxPool2D-2	(6,6,64)	0
Conv2D-3	(4,4,128)	73856
MaxPool2D-3	(2,2,128)	0
Dense-1	(1024)	525312
Dense-2	(43)	44075

The final dense layer has 43 outputs representing class probabilities for each type of sign. *argmax* is performed over them, and the class with the highest probability is chosen as the classification.
Each convolutional layer uses $1 * 1$ strides, while the max pooling layers use a $3 * 3$ stride. The number of layers depends on the input resolution; in our case, resizing images to a lower resolution would cause a considerable loss of quality, and increasing the resolution only adds to training time without gaining any boost in accuracy (in some cases, even hurting it), so a three layer approach was chosen, with the output volumes being condensed to $(2, 2, 128)$ in the final layer.
Total trainable parameters for this network is 662.635, which is manageable,

perhaps even on the low side considering current trends of deep networks. Another gain of having a relatively shallow network is better inference time.

Initialization Strategy

By initializing all the weights and biases of the neural network to zero, we will cause them to not be updated in future iterations. This is explained by how the backpropagation algorithm works.

$$\frac{\delta}{\delta\theta_{ij}^{l}} = \alpha_j^l \delta_i^{l+1} \tag{2.1}$$

$$\alpha_j^l = g(\theta^{l-1}\alpha^{l-1}) \tag{2.2}$$

$$\delta^l = (\theta^l)^T \delta^{l+1} * g'(\theta^{l-1}\alpha^{l-1}) \tag{2.3}$$

where
(2.1) is the calculation of the partial derivatives for weights in layer l
(2.2) the application of non-linearity g to the neuron's output
(2.3) the difference propagated backwards from the succesive layer
and $*$ stands for element-wise multiplication.
By looking at how the weights are computed, zero-valued weiths would prevent the activation from changing. For δ, zero-valued weights multiplied by the delta from the succesive layer's delta lead to zero delta, effectively blocking the errors from propagating past them. Note that choosing another constant value for initialization would lead to the network getting stuck in a local minima in the loss function's hyperspace.
Therefore, an initialization strategy based on random sampling is required. For this we have chosen the "Glorot Uniform" method, as described in the paper by Xavier Glorot[14]. It works by drawing a sample from a uniform distribution within $[-a, +a]$ where $a = \sqrt{\frac{6}{n_{inputs}+n_{outputs}}}$ with n_{inputs} the number of inputs in the weight tensor and n_{output} the number of output units in the weight tensor.
This initialization strategy is also a good way of preventing the vanishing/exploding gradients problem. It ensures that the signal flows properly in both directions, without it dying out or exploding and saturating. The authors of this method argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs, and the gradients have to have equal variance before and after flowing through a layer in the reverse direction. It is actually not possible to guarantee both unless the layer has an equal number of input and output connection, but the initialization scheme described above is a good compromise.

Note that we have mentioned earlier in this paper that randomization should be turned off when validating for different hyperparameter values. However, when dealing with weight initialization, it is simply unavoidable and some margin of error has to be taken into account.

Activation Functions

Four activation functions are compared in this section: logistic (sigmoid), ReLU (rectified linear unit), leaky ReLU and ELU (exponential linear unit). For the final layer, the softmax activation is detailed immediately afterwards.

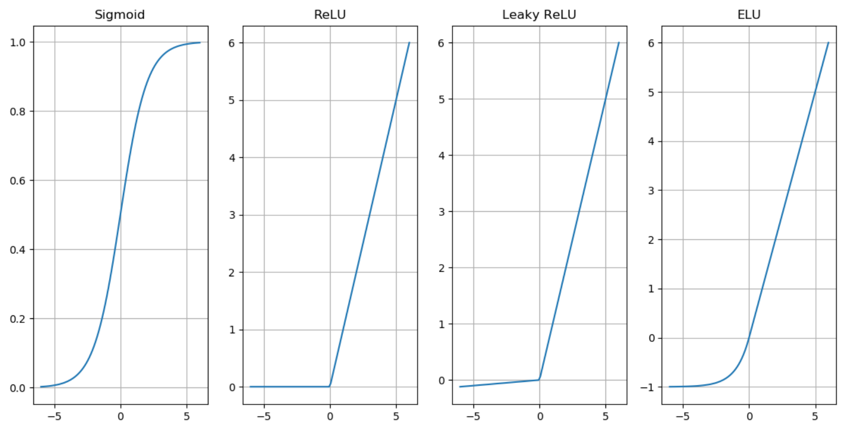


Figure 2.2: Plotting the four activation functions we have tested.

$$sigmoid(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

The **sigmoid function** is the first function we have tested and has caused training to be slower than expected. It is also known to cause other problems such as the vanishing gradients (due to the fact that the function saturates at 0 or 1, its derivative getting extremely close to 0, causing the gradients to progressively get diluted as backpropagation passes them down to the lower layers), particularly for deeper networks. As such, we have chosen to avoid using the sigmoid activation, even though it is standard in many networks.

$$relu(x) = max(0, x)$$

ReLU (rectified linear unit) is a slight improvement in the sense that it does not saturate for positive values, but has shown similar computation speeds and caused us to run into the *dying ReLUs* problem, especially when we used a larger learning rate. During training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs are equal to 0, it will start outputting only 0 (the neuron has "died"). In this event, it is unlikely that the neuron will "come back to life" since the gradient of the ReLU function is 0 when its input is negative. To circumvent this problem, we have used some of the variants of ReLU.

$$leaky\_relu_\alpha(x) = \begin{cases} x & x \geq 0 \\ x\alpha & x < 0 \end{cases}$$

Leaky ReLU allows for a small slope when $x < 0$. It has outperformed the simple ReLU function in terms of performance and has completely eliminated the dying ReLUs problem. We have tested with multiple values for the α parameter, ranging from 0.01 to 0.2, and found comparable performances.

$$elu(x) = \begin{cases} x & x \geq 0 \\ \alpha(\exp(x) - 1) & x < 0 \end{cases}$$

ELU has shown to outperform all other variants of the ReLU function in terms of accuracy. It has also boosted training time by a considerable amount (in the sense that it took less epochs to reach similar performance) due to the fact that the exponential linear units try to make the mean activations closer to zero. Given this result, it has been chosen as the activation function for the convolutional and fully-connected layers.

For the final layer of the neural network (which is responsible for classification) we have used the **softmax** activation, a generalized form of the logistic function, which transforms a real-valued k-dimensional vector z to a k-dimensional vector $\sigma(z)$ of real values, where each entry is in the interval $(0, 1)$ and all the entries add up to 1. It is defined as:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } j = \overline{1, K}$$

with K the total number of classes (in our case, 43) Its properties are very useful since we want our model to output a probability attached to each class.

Optimizers

Trying to improve on the performance of the standard stochastic gradient descent, we have experimented with several other optimization algorithms.

We have tested Nesterov Accelerated Gradient, Root-Mean-Square Propagation (RMSProp) and Adaptive Moment Estimation (Adam). In this section we will briefly look over at how these optimizers work and compare their resuls.

Nesterov Accelerated Gradient proposed by Yurii Nesterov[8] is a variant of the momentum optimization technique. The idea behind it is to measure the gradient of the cost function not at the local position, but one step ahead in the direction of the momentum.

$$m \leftarrow \beta m + \eta \nabla_\theta J(\theta + \beta m) \tag{2.4}$$

$$\theta \leftarrow \theta - m \tag{2.5}$$

where $\beta \in [0, 1]$ is the momentum, J the cost function, η the learning rate, m the momentum vector, θ the weights, and $\nabla_\theta J(\theta + \beta m)$ the gradient of the cost function measured at the direction ahead of the current gradient.

This tweak works because, in general, the momentum vector will generally be pointing in the right direction, so it will be slightly more accurate to use the gradient measured a bit farther in that direction. As evidence, it has outperformed regular stochastic gradient descent by a large margin. The hyperparameter β can be seen as a friction factor to prevent the momentum from getting out of hand; 0 means high friction, and 1 means no friction. A typical momentum value is 0.9, which we have used in our tests.

Root Mean Square Propagation has outperformed the Nesterov optimizer by approx. 4 percentage points in terms of accuracy when running on 60 epochs, and offered an improved loss value. It works by scaling down the gradient vector along the steepest direction.

$$s \leftarrow \beta s + (1 - \beta) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta) \tag{2.6}$$

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \div \sqrt{s + \epsilon} \tag{2.7}$$

The first step accumulates the square of the gradients from the most recent iterations (by using the exponential decay rate β, typically set to 0.9).

The second step is similar to Gradient Descent, except it scales down the gradient vector by a factor of $\sqrt{s + \epsilon}$, where ϵ is a smoothing term to avoid division by zero, typically set to 10^{-10}.

In short, this optimizer decays the learning rate, but it does so faster for steep dimensions and slower for dimensions with gentler slopes. Its adaptive learning rate is very useful since it means we don't have to tune the η parameter as much.

Adam (Adaptive Moment Estimation) is the final optimizer we have experimented with, and has shown very similar performance with RMSProp ($\pm 1\%$).

However, the training time has been reduced, and, as such, it has been chosen as the optimizer for our neural network. It keeps track of an exponentially decaying average of past gradients, and an exponentially decaying average of past squared gradients. It is described as follows:

$$m \leftarrow \beta_1 m + (1 - \beta_1)\nabla_\theta J(\theta) \tag{2.8}$$

$$s \leftarrow \beta_2 s + (1 - \beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta) \tag{2.9}$$

$$\theta \leftarrow \theta - \eta m \div \sqrt{s + \epsilon} \tag{2.10}$$

The momentum decay parameter β_1 is typically chosen 0.9, and the scaling decay parameter β_2 is initialized to 0.999. As earlier, the smoothing term is chosen as 10^{-10}.

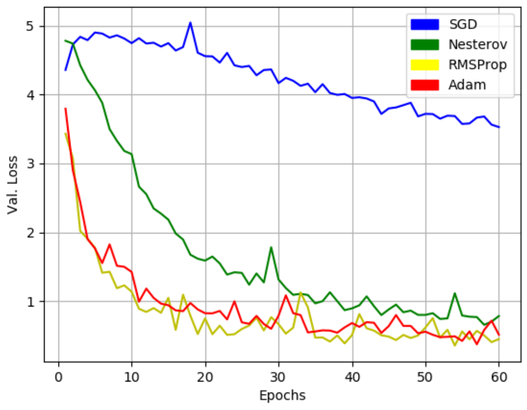


Figure 2.3: Plotting the validation loss of the four optimizers for 60 epochs of training.

Key Performance Indicators

In order to evaluate the performance of our network, we used a loss function and an accuracy metric.
For loss function we have used categorical crossentropy, which models the difference between the ground-truth distribution y and the predicted distribution $\hat{y}$. Mathematically, it is described as follows:

$$H(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i$$

Backpropagation minimizes the loss function with respect to the network's weights. In doing so, it gets as close as possible to the ground-truth in hopes that it can generalize to novel data.
To measure how many of our samples are predicted correctly, we have used the accuracy metric, which is the ratio between the correctly predicted samples and the total number of samples.(true positives and true negatives divided by the number of samples)

$$acc(X, y) = \frac{tp+tn}{tp+tn+fp+fn}$$

Note that when we have discriminated between hyperparameter settings we have used these metrics on the validation and training data. These metrics were used on the test data only when a good baseline was established for the model.

Regularization

We have observed that our network began overfitting the training set without achieving an acceptable performance. As such, we had to employ some methods to regularize it in order to avoid this problem.
Dropout works by assigning a dropout rate p to each neuron in a layer. At every training step, that neuron will be "dropped out" with p probability, effectively being ignored for that respective step. This works due to the fact that the neuron is unable to co-adapt with its neighboring neurons and has to learn by itself, ending up being less sensitive to changes in inputs. A rate of 0.3 has been used for the fully-connected layer. Adding dropout to convolutional layers seemed to decrease performance and is generally not a good practice. Adding a larger degree of dropout would cause the network to underfit the dataset.
L2-regularization works by tweaking the loss function to include the neurons' weights. Specifically, our loss function becomes:

$$H(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i + \lambda \sum_i w_i^2$$

where the term

$$\lambda \sum_i w_i^2$$

is the sum of the squared weights of all neurons to which we applied regularization, and λ is a hyperparameter.

The effect of the regularization is that our loss function will yield worse values for bigger weights. In doing so, the network is forced to keep the weights of the regularized layers on the small side.
We applied regularization to the first fully connected layer (with a standard $\lambda = 1e-4$) and have observed more robustness in our network and resistance to overfitting. Adding it to convolutional layers has incapacitated the feature extraction ability of the network and as a result the performance dropped significantly; therefore, we have chose not to use it on the convolutional layers.

Other Parameters

Learning Rate & Scheduling

Since we are using an adaptive learning rate algorithm (Adam) it is not strictly necessary to fine tune the learning rate. Adam's default learning rate is 1e−3 and, in practice, yields good results. However, we have found out that when coupling it with a learning schedule and using a more aggresive learning rate we obtain several benefits, such as training for a larger number of epochs before overfitting and reaching the performance of the model with no schedule faster (in a lower number of epochs).
The particular schedule that yielded the best results was monitoring the value of the loss function and reducing the learning rate by a factor of 20% once there is no improvement for 3 epochs. A minimum threshold of 1e−4 for the learning rate is applied.
Adding a learning schedule to an already adaptive learning rate optimizer is currently an **undocumented** method, and is one of the novel insights generated through our experiments. The final learning rate set for the Adam optimizer is 3e−3.

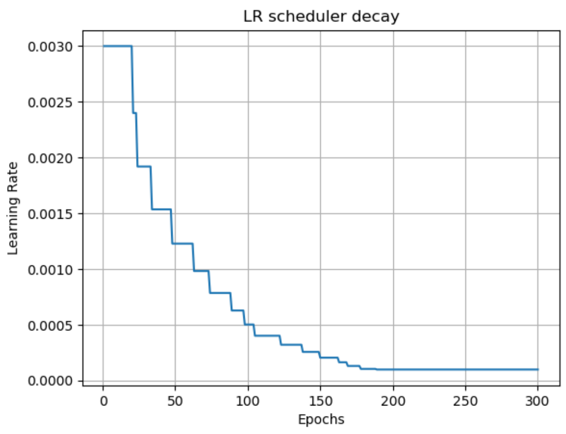


Figure 2.4: Learning rate as decayed by the scheduler.(300 epochs)

Model Checkpoint & Epochs

We have added two model checkpoint callbacks which monitor the value of the loss function as well as the accuracy on the validation data, and save the best performing models, depending on each metric. This was done to ensure that we can use a higher number of training epochs without overfitting, allowing us to save compute time. As such, the total number of epochs trained on was 300, with overiftting observed around epoch 280, where the performance on the training and validation set began to diverge.

Batches

In all of the experiments we have used mini-batch training. The difficulty in choosing the number of samples to include in each mini-batch comes from the fact that using a lower number will, on the one hand, require much less memory and train faster, but on the other hand using a number of samples that is too small might affect the accuracy of the gradient estimates. Standard batch sizes that offer compromises between the two extremes are $128, 256, 512, 1024, 2048$. In our case, a batch larger than 256 seemed to yield no noticeable benefits, but a batch of 128 damaged the model's performance. As such, we have chosen 256 as the size of the batch for our model.

Putting It All Together

Until now, we have detailed the choices in the network's hyperparameters. In this section, we will present our results. Below is a chart for the training and validation values of the performance indicators.

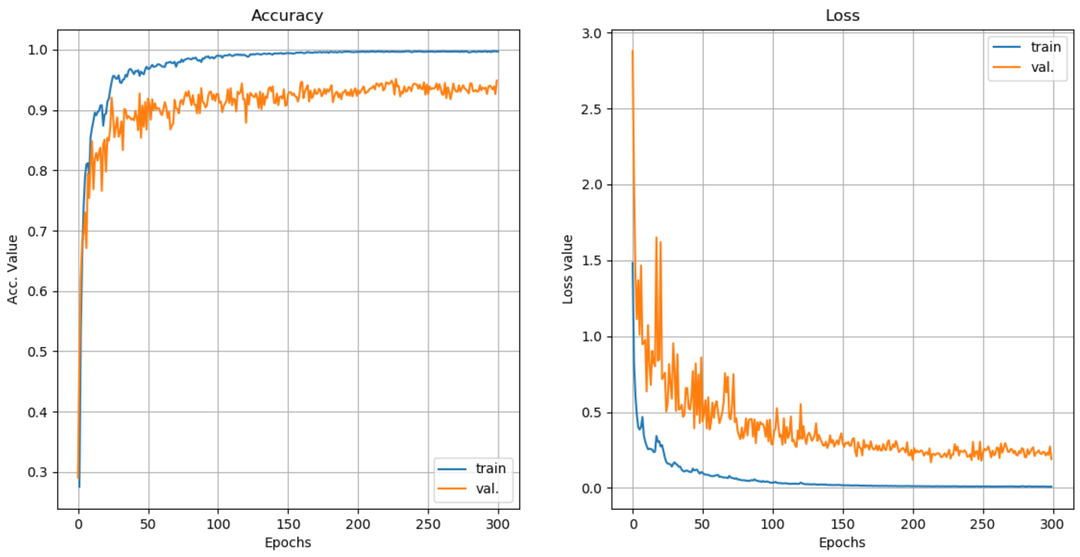


Figure 2.5: Training and validation metrics functions.

The difference in the values of the training and validation sets are due to the fact that the validation set has a much lower number of samples than the training set and does not have enough representation power when compared. However, it is still useful as a metric since their trends should be similar (upward for accuracy, downward for loss) before overfitting. At around epoch 270, barely any change is noticeable, and the network starts to overfit. The model recorded at around that epoch is stored and saved.
Test set results are **98.1%** accuracy and a loss value of approx. **0.116**, which is quite good considering that human-level detection is, according to a recent paper by Yann LeCun[15] approximately the same: 98.8%.

Chapter 3

Integrating into a Detection System

Since we now have a high performing neural network model saved and ready for inference, we have researched several methods in which we might integrate this into a system that also performs detection of the traffic signs. In this chapter, we define a possible pipeline that can be used for this task as well as the methods for detection.

Neural Network as a Feature Extractor

In all of the methods outlined in the following sections, we have used the convolutional neural network as a feature extractor. What this means is that we retrieved the trained model (saved as a file to disk), and removed the layers that performed classification (e.g. the two fully-connected layers). By passing an image into this newer model, we obtain a vector (tensor) that represents the convolutional features extracted so far by the network. It has been shown in a paper by Razvaian et al.[3] that convolutional features work better than Haar, HOG or SIFT style features as they are able to capture more patterns and are also invariant to small shifts in the image. Therefore, they are perfect for training a binary classifier on.
As such, the pipeline that we want to build involves running a sliding window or image segmentation technique on the image, and for each patch of potential object, pass it through the convolutional and pooling layers to extract the features, use those features to classify it as either sign or background, and if it is a sign, pass it through the fully-connected layers of the network to perform classification. Finally, we add a labeled bounding box to the image.

Outlier Detection Techniques

Initially we have experimented with outlier detection techniques. Such a model doesn't require labeled data and attempts to classify novel data as different or similar to the data it was trained on. Some of the outlier detection models we have tried are Isolation Forest, Local Outlier Factor and One-class Support Vector Machine. They have been trained on the traffic sign data and tested on the background pictures. While they have performed well on the training set, the results have been very poor on the testing set and as such we have chosen not to explore this option any further.

Support Vector Machines

A support vector machine is a very powerful and versatile machine learning model, able to perform binary classification. It works by drawing a linear decision boundary (if using the linear kernel) as to fit the widest possible street to separate the negative and the positive samples. This is called large margin classification. Note that when training an SVM, adding instances "off the street" does not affect the decision boundary at all: it is fully determined by the instances located on the edge of the street. These instances are called the *support vectors.*
The linear SVM classifier model predicts the class of a new instance x by simply computing the decision function $w^T \cdot x + b$. If the result is positive, the predicted class $\hat{y}$ is the positive class, otherwise it is the negative class.

$$\hat{y} = \begin{cases} 0, if & w^T \cdot x + b < 0, \\ 1, if & w^T \cdot x + b \geq 0 \end{cases}$$

where w is the feature weights vector (coordonates of a vector orthogonal to the hyperplane, which is the decision boundary), and b the bias term

To get a margin as wide as possible, we want to minimize the slope of the decision function, which is equal to the norm of the weights vector $\|w\|$. The smaller the slope, the wider the margin, since reducing the slope will cause the distance from the points where the decision function is ± 1 to increase. We also want to avoid any margin violation, so we define $t^{(i)} = -1$ for negative examples, and $t^{(i)} = 1$ for positive samples to express this constraint as $t^{(i)}(w^T \cdot x^{(i)} + b) \geq 1$ for all instances. We can now express the objective function as follows:

$$\begin{aligned} &\min_{w,b} && \frac{1}{2} w^T \cdot w \\ &\text{subject to} && t^{(i)}(w^T \cdot x^{(i)} + b) \geq 1 \quad \text{for} \quad i = \overline{1, m} \end{aligned} \tag{3.1}$$

If we impose that no instances may violate the margin, however, our model will only function if the data is linearly separable, and be extremely sensitive to outliers. As such, we have used a method called *soft margin classification*, which is a more flexible model. To get the soft margin objective, a slack variable $\xi^{(i)} \geq 0$ has to be introduced for each instance, measuring how much the i^{th} instance is allowed to violate the margin. Reducing the margin violations and making the margin as wide as possible are actually conflicting objectives, so we require a hyperparameter $C \in [0, 1]$ which defines a trade-off between the two objectives. This gives us the constrained optimization problem for the soft margin classification:

$$\begin{aligned} &\min_{w,b,\xi} \quad \frac{1}{2} w^T \cdot w + C \sum_{i=1}^{m} \xi^{(i)} \\ &\text{subject to} \quad t^{(i)}(w^T \cdot x^{(i)} + b) \geq 1 - \xi^{(i)} \quad \text{and} \quad \xi^{(i)} \geq 0 \quad \text{for} \quad i = \overline{1, m} \end{aligned} \tag{3.2}$$

A SVM doesn't necessarily fit a linear function, and can build more complicated decision boundaries by using the *kernel trick* which adds slight modifications to its internal representation so as to switch to a different space. However, we have obtained good metrics when running it with the linear kernel, so this option was skipped.
When applying a support vector machine with linear kernel to convolutional features extracted from the data (both sign and background images) we have obtained the following results:

Train Acc.	Train F-Score	Test Acc.	Test F-Score
99,936%	99,934%	99,088%	99,096%

Confusion matrix for training and testing, respectively:

$$\begin{bmatrix} 26803 & 7 \\ 26 & 25324 \end{bmatrix}$$

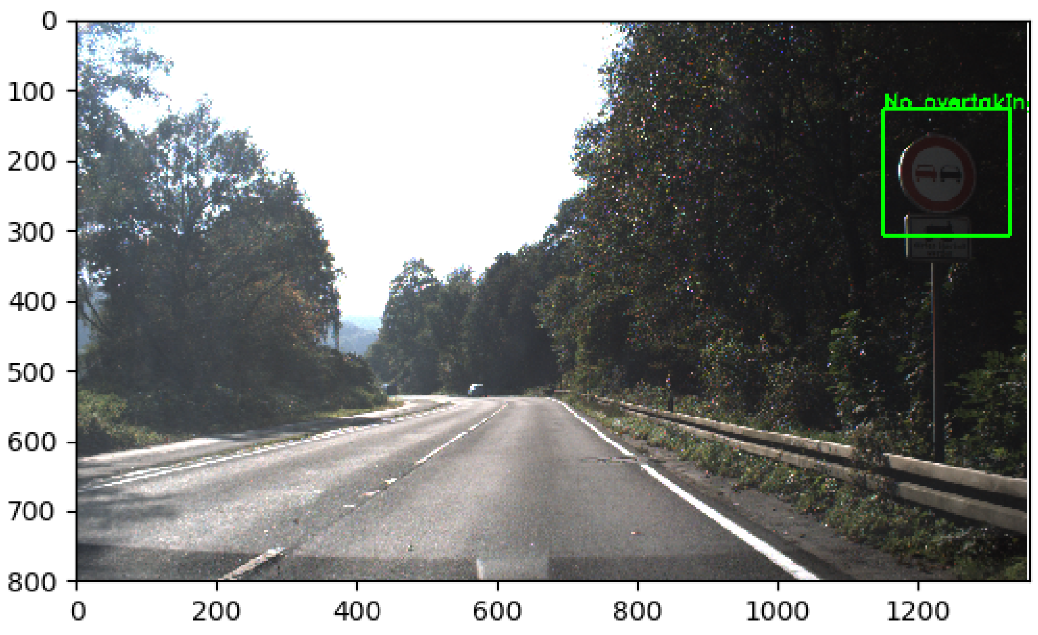
$$\begin{bmatrix} 12214 & 109 \\ 118 & 12451 \end{bmatrix}$$

where $F1score = 2 * \frac{p+r}{p*r} \quad p = \frac{tp}{tp+fp} \quad r = \frac{tp}{tp+fn}$

Note that in order to get the best possible C parameter, we have used grid search over a list of values ranging from $1e-5$ to 1, with a total of 12 runs over the validation set. The C value that offered the best results is 0.8.

The Pipeline

Now we can combine these two models and use them in a pipeline. First, we slide a window over the image and for each patch we pass it through the first layers of our neural network to perform feature extraction. We classify these features using the trained support vector machine. If it yields 1 then it is a sign, and we pass it to the final fully-connected layers of the network. Moreover, we perform thresholding over the neural network's probabilities to eliminate false positives as much as possible. A threshold of 80% seemed to work best in practice. Also, when dealing with multiple bounding boxes in the same region that belong to the same sign, we simply merge them to form a larger rectangle. Here is an example of a labeled picture that our pipeline outputs:



Final Insights

Conclusion

The development of real-time object detection systems, particularly when dealing with traffic sign detection, is still an open-research field, and poses a multitude of challenges. While it is true that with the advent of new deep learning techniques, great strides have been made in terms of runtime and precision alike, they still remain inaccessible to most developers due to the data and hardware requirements.
In this sense, the method that has been developed in this thesis, which is inspired from the Region-Based Convolutional Neural Network model [4] serves as a good first step towards accessible and accurate object detection systems. It can be applied to any type of object detection task given a labeled training set (no bounding-box requirements), a background dataset that can be generated without massive costs and using relatively modest computing power (FloydHub's CPUs clusters were sufficient).
That being said, there are still a number of improvements that can be added to our existing model. In the next section, some of these improvements are briefly detailed.

Future Considerations

Residual modules, as defined by Xiangyu Zhang et al.[5], introduce skip or shortcut connections which allow the network to use multiple scales making it easier for the layers to learn the identity mappings. It has been demonstrated empirically to benefit the accuracy of the networks and smooth the training process.
The OverFeat Method described by Pierre Sermanet, Yann LeCun et al. [2] is a way of increasing performance speed. It works by stripping the final fully connected layers from the network and passing the entire picture as an input (as seen from a car's dashboard camera, in our case) and outputting

a feature map for the entire image. This is mathematically equivalent to running a sliding window with strides equal to the sum of the strides in the network's layers and passing them one by one as inputs. This is useful when wanting to use the network as a feature extractor, like in our case.
Multithreading implementations take advantage of the modern CPU multi-core architectures by running multiple tasks in parallel. In our case, we could adapt our scripts to use multiple threads for different parts of the image independently. Additionally, we can reduce the workload by ignoring the bottom, top and centermost part of the image since signs are highly likely to appear on either the left or the right side (with a few exceptions).

Appendix A

Appendix: Frameworks & Tools Used

All of the source code has been written in Python v3.6 and has multiple dependencies. Some libraries used include NumPy (for linear algebra computation and storing the datasets as tensors), Matplotlib (used to generate plots and figures) as well as various standard Python modules: os (general operating system functionality), pickle (serialization and de-serialization), glob (Unix-style pathname pattern expansion), etc.
In this chapter, we will present the frameworks as well as the cloud service used for training and tweaking the machine learning models involved, which is the bulk of the application.

OpenCV

OpenCV (Open Computer Vision)[13] is a library used mainly for computer vision and image manipulation tasks. It is written in optimized C/C++ and also has interfaces for Python and Java. In the context of this project, it has been used for applying transformations and resizing/reshaping to the image dataset in order to bring it to a state which a neural network might learn best from. The entire dataset preparation process is detailed in the $1st$ chapter.

Tensorflow and Keras

According to the official documentation[7], TensorFlow is an open-source software library for high performance numerical computing. The way that it operates is by graph computation, which means that at runtime, a computational graph is defined via the Python API. When we run the Python interpreter, the graph is then passed to a C parser which handles all of the

computation. This is done to ensure a high degree of efficiency as well as a low memory footprint. While TensorFlow can be used for any type of numerical computation, its main use in this project is the development of the convolutional neural network. More precisely, we have used it as a back-end for a deep learning framework, Keras.
Keras[11] is a high-level neural networks API, capable of running on top of TensorFlow, CNTK or Theano. Its use inside this project is as a wrapper for TensorFlow, allowing us to easily prototype different model architectures as well as hyperparameters. Another advantage of Keras is that it runs seamlessly on CPU/GPU clusters, reaching almost the native speeds of its back-end library.

Sci-kit Learn

Sci-kit Learn (or SKLearn)[12] is a Python library which includes various machine learning algorithms as well as evaluators and functions used to manipulate data. We have used its SVC package (for defining and running support-vector machine models), outlier detection modules (OneClassSVM, IsolaionForest etc.). For debugging, various evalutaion metrics have been used such as confusion matrix, F1-score and accuracy.
Training, testing and validation sets are passed to SKLearn as NumPy tensors and all of the numerical computation is also done using highly efficient linear algebra operations defined in NumPy. Its utility is showcased in the Detection chapter.

FloydHub

FloydHub[6] is a cloud ecosystem which allows us to run our application using high performance hardware. Specifically, we have used an Intel Xeon® dual-core processor and an Nvidia© Tesla K80 GPU with 12 GB VRAM, 61 GB RAM and 100 GB SSD. When running inside the cloud application, we have used a version of TensorFlow compiled specifically for the instruction set used by the CPU and GPU. This has gained us a lower training and test time which allowed for faster experimentation and development.
It has a command-line interface from which an user can upload/update a dataset and run his application, which then gets assigned a job number and is queued for execution. FloydHub also offers a browser interface where a user can visualise training metrics (accuracy and loss function value, in our case) as well as system metrics (CPU/GPU/memory/disk usage).

Bibliography

[1] LeCun et al. *Gradient-Based Learning Applied to Document Recognition.* http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf.

[2] Pierre Sermanet et al. *OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks.* https://arxiv.org/abs/1312.6229.

[3] Razvaian et al. *CNN features off-the-shelf: an Astounding Baseline for Recognition.* https://arxiv.org/pdf/1403.6382.pdf.

[4] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation.* https://arxiv.org/abs/1311.2524.

[5] Xiangyu Zhang et al. *Deep Residual Learning for Image Recognition.* https://arxiv.org/abs/1512.03385.

[6] FloydHub Inc. URL: https://www.floydhub.com/.

[7] Google Inc. *TensorFlow Docs.* URL: https://tensorflow.org/.

[8] Yurii Nesterov. *A method for unconstrained convex minimization problem with the rate of convergence O(1/k2).* https://goo.gl/V011vD.

[9] Institüt fur Neuroinformatik. *German Traffic Sign Detection Benchmark.* URL: http://benchmark.ini.rub.de/?section=gtsdb&subsection=dataset.

[10] Institüt fur Neuroinformatik. *German Traffic Sign Recognition Benchmark.* URL: http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset.

[11] Project ONEIROS. *Keras Docs.* URL: https://keras.io/.

[12] *SKLearn Docs.* URL: https://keras.io/.

[13] OpenCV Team. *OpenCV Official Documentation.* URL: https://opencv.org/.

[14] Yoshua Bengio Xavier Glorot. *Understanding the difficulty of training deep feed-forward neural networks*. `http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf`.

[15] Pierre Sermanet Yann LeCun. *Traffic Sign Recognition with Multi-Scale Convolutional Networks*. `http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf`.