Traffic Signs Classifier

Goal:

Tune and train a notorious deep neural network model (LeNet) . This model originally designed to classified numbers (0 to 9) is customized in this study to recognize images of 49 different traffic signs.

Data overview :

Training, validation and tests images are downloaded from German Institute of Neuro Computer Science " *the Institüt fur NeuroInformatik*".

RUHR-UNIVERSITÄT BOCHUM
INSTITUT FÜR NEUROINFORMATIK

Zipped data sets can be downloaded on the *Institüt for NeuroInformatik* website.

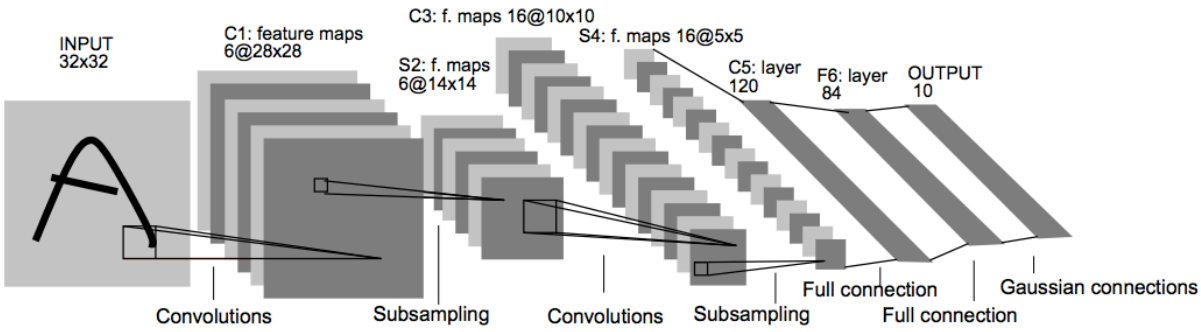
The German Traffic Sign Recognition Benchmark GTSRB offers :

- 3 channels (RGB) 32x32 pixels images
- 55,000 training samples
- 4,410 validation samples
- 12,630 testing samples

Neural network architecture:

The project builds onto the iconic " LeNet" CNN based deep neural network architecture.

This model was originally designed to classify numerals.



- convolutional layers, kernel size is (5x5) , stride 1, no zero padding (i.e "valid")
- subsampling layers are MaxPooling, kernel (2x2) , stride 2

(**C**onvolution / **P**ooling)*/ (**D**ense) *

(28x28x1) -> C:5 / MxPool:2 / C:5 / MxPool:2 / D:120 / D:84 / D:10

We'll modify this model's architecture and hyper-parameters to improve its accuracy .

Programming environment :

Model design and training:

We'll use the high level API Keras (build on tensor flow backend) to focus on fine tuning the model rather than on mastering TensorFlow programming.

Data pre-processing and augmentation :

- pre-processing data set generation :

  To even out the number of different samples per class we used the open source python package Augmentator .

- on-the-fly data augmentation :

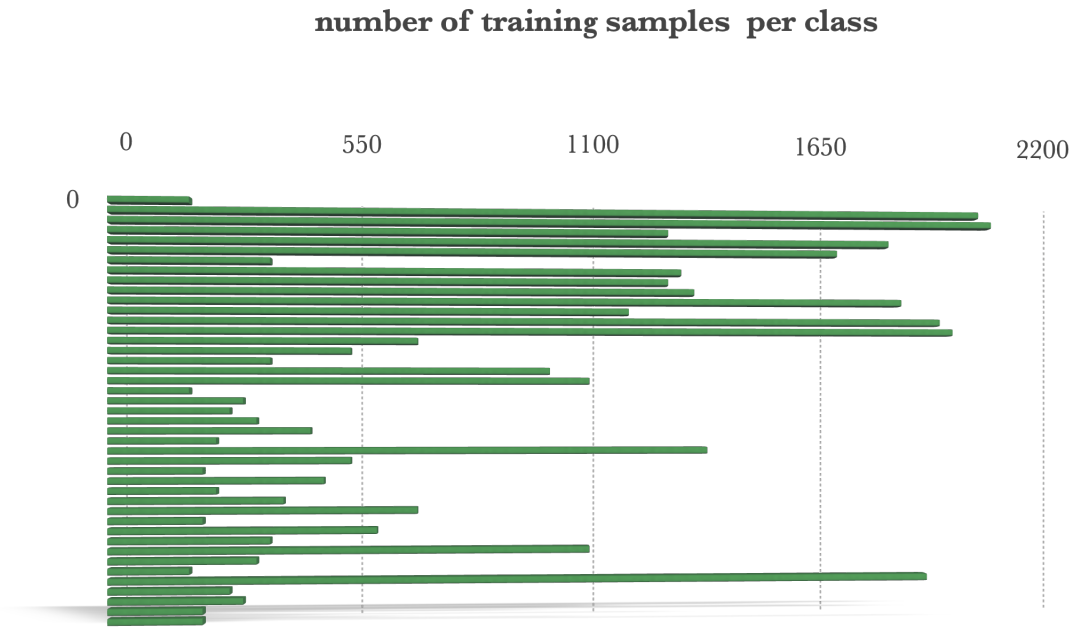
  We use a tf.keras ImageDataGenerator class and its transformations toolbox as a part of others regularization techniques .

Training the model and evaluating the model "AS-IS" :

We trained an evaluated "LeNet digit classifier" on the traffic sign dataset without modifying any of its original hidden layers and hyper-parameters .

G.I.N.C.S sets :

Each is composed of (32x32x3)(R,G,B) images:



- training set: 34,799 images
- validation set : 4,410 images
- test set : 12,630 images

Quality of the training set :

The training set is unbalanced : the number of samples per class varies from 180 to 2010.

Performance comparison:

On **standardized** images , the accuracy of the LeNet falls from 98 % on classifying digits to 89 % on classifying traffic signs

Causes for this regression could be intuitively asserted as :

- more classes : 49 different traffic signs (i.e classes) instead of the 10 for numbers
- input channels number raising from 1 (grey images) to 3 (RGB) images
- a smaller training set relatively to the larger number of classes (55,000 numbers images from the MNIST set vs 34,799 signs images from the GTSRB set).
- the digit classifier was trained on white background images. On the contrary traffic signs training pictures are taken in-situ. Each sign is surrounded by visual context that can could be interpreted as " noise".

Assessment conclusions :

From the previous observations we'll work on two domains to improve the model :

- hyper-parameters and architecture tuning
- data augmentation, preprocessing and initialization

LeNet customization for traffic signs classification:

We use the original LeNet model for numbers classification:

(**C**onvolution / **P**ooling)*/ (**D**ense) *

(28x28x1) -> C:5 / MxPool:2 / C:5 / MxPool:2 / D:120 / D:84 / D:10

We modified the **input** and **output** layers to accommodate the traffic sign RGB images and its 43 classes as:

(32x32x1) -> C:5 / MxPool:2 / C:5 / MxPool:2 / D:120 / D:84 / D:43

After a few dry runs we settled on a the batch size of 100 and 20 epochs to focus on tuning the model.

1) Hyper-parameters and architecture tuning:

We modified the model parameters one step at a time and train these modified models on the original traffic signs data set.

Modifications and their influence on the **validation set** accuracy are listed below.

All positives changes will be retained to design our final model.

		validation accuracy	
	tested	**result**	**comments**
	data standardization	gain + 45%	a prerequisite in all neural networks (X-mean)/std), See Appendix 4: data preprocessing
	activation Leaky Relu (instead of Relu)	**regression -2%**	will stick to the conventional Relu
CNN layers	**CNN padding :** switching from "valid" to "same "	**gain +1%**	on a on a (32x32) image, a (5,5) kernel with"valid"padding shrinks to (28x28) filters: 4 boarder pixels (1/6) were not fully accounted for.
	CNN kernel size: (C5 to C3 or C7)	**regression -1%**	due to the small size of the images (5x5) kernel size remains the best choice
	number of filters: increase from (C5:6)(C5:16) to (C5:24)(C5:32)	**gain +2%**	more variance in traffic signs features than on numbers: more filters were needed to map these features
	pooling type: switching from MaxPooling to AveragePooling	**regression -2%**	
	changing the pooling size	**regression -1%**	will stick to (2x2) pools
	adding CNN layers	**no gain**	different kernel sizes and number of filters : no effect
	size and the number of dense layers : from (120, 84) to (242, 121, 84)	gain **+0.5%**	
regularization	**Batch Normalization**	**no gain**	training converges faster, moot point as overfitting remains unaddressed
	dropout 0,25	**gain : +2%**	one efficient way to help with overfitting
	dropout 0,12 or 0,50	**regression -1%**	regression really fluctuates significantly from training to training
	learning rate decay :1 e-4	**no gain**	sturdier convergence at the end of the training (accuracy stops oscillating)

Our final model is:

Layers: (**C**onvolution / **P**ooling)*/ (**D**ense) *

C:5 / MxPool:2 / C:5 / MxPool:2 / D:242 / D:121 / D:84 / D:43

Optimizer : Adam

Learning rate : 1e-3 , exponential decay of 1 e-4.

Initialization:

For weights initialization we used the default Keras "glorot_uniform'.

Kaiming would have been a better choice for Relu activation.

(Kaiming not available with Keras yet : for more info on Kaiming initializer: https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79 **courtesy of James Dillinger**)

2)data augmentation:

To further address overfitting we used two techniques back to back to augment the data set :

- preprocessing augmentation adding more samples to the given set before training
- on-the-fly data augmentation editing the training set samples at each epoch

preprocessing augmentation :

Each class is not evenly represented within the training set : population per class ranges from 180 to 2010 individuals

([180, 1980, 2010, 1260, 1770, 1650, 360, 1290, 1260, 1320, 1800, 1170, 1890, 1920, 690, 540, 360, 990, 1080, 180, 300, 270, 330, 450, 240, 1350, 540, 210, 480, 240, 390, 690, 210, 599, 360, 1080, 330, 180, 1860, 270, 300, 210, 210]) .

We designed a Python function built on the image processing class Augmentator to even out the training set. This function adds images to each class so the new training set is balanced , each class counting now 2010 individuals .

We applied a series of random transformations (skew_tilt, zoom, shear, random_color, random_brightness, random_contrast) to the original training data set to expand it.

For more on Augmentator. courtesy of Marcus D. Bloice, MIT

See Appendix 5: data preprocessing , data set augmentation

on-the-fly data augmentation:

I conventional trainings, sets remain immutable throughout all epochs.

ImageDataGenerator allows a different approach : the training set is transformed at each epoch through random operations (width_shift_range, height_shift_range, shear_range, zoom_range the training data.

The training set virtually grows one fold at each epoch. Understandably the model requires more epochs to converge (as weights have to be fitted on a new set at each epoch) . The model generalizes much better as it gets to back propagate on a "fresh set" at each epoch.

	data augmentation	
	result	**comments**
preprocessing augmentation	**gain +2%**	necessary in this study case to balance the training set population
on the fly augmentation	**gain +2%**	increases accuracy and lower overfitting.

Results:

Some structural modifications (layers additions) moderately contributed in the overall model improvement . Although the training accuracy improved, the model kept overfitting.

Adding drop-outs at each layer raised the validation accuracy significantly (+2%). Balancing the training set population, augmenting it statically and on the fly improved the validation accuracy by 4%. Evolution of the training accuracy and validation set on our last final model can be found @ Appendix 3: Training and validation accuracy.
These architectural choices and training set augmentations ultimately helped our model to reach a 95% accuracy on the testing set .
Test set Precision broken down by classes can be found @ Appendix 1, precision , recall
A test set confusion matrix details expected vs predicted @ Appendix 2: Test set confusion matrix :

Prediction on unknown samples:

We found 6 images of traffic signs on the web. Once cropped and resized, we'll run them through the prediction function and associated the class result to the sign description (found in signnames.csv). Images and prediction output description are below :

Conclusions:

Due to the nature of the problem, data augmentation helped addressing overfitting: within a class every signs are identical, they will appear different according to the lighting condition, the viewing angle or the distance. These different viewing conditions are addressed in training by basic transformations in data augmentation (Augmentator, ImageDataGenerator).

On the contrary partially hidden or partially shaded signs won't benefit from this type of augmentation. Hide and seek occlusions packages are commercially available to integrate these kinds of augmentation in the training pipeline.

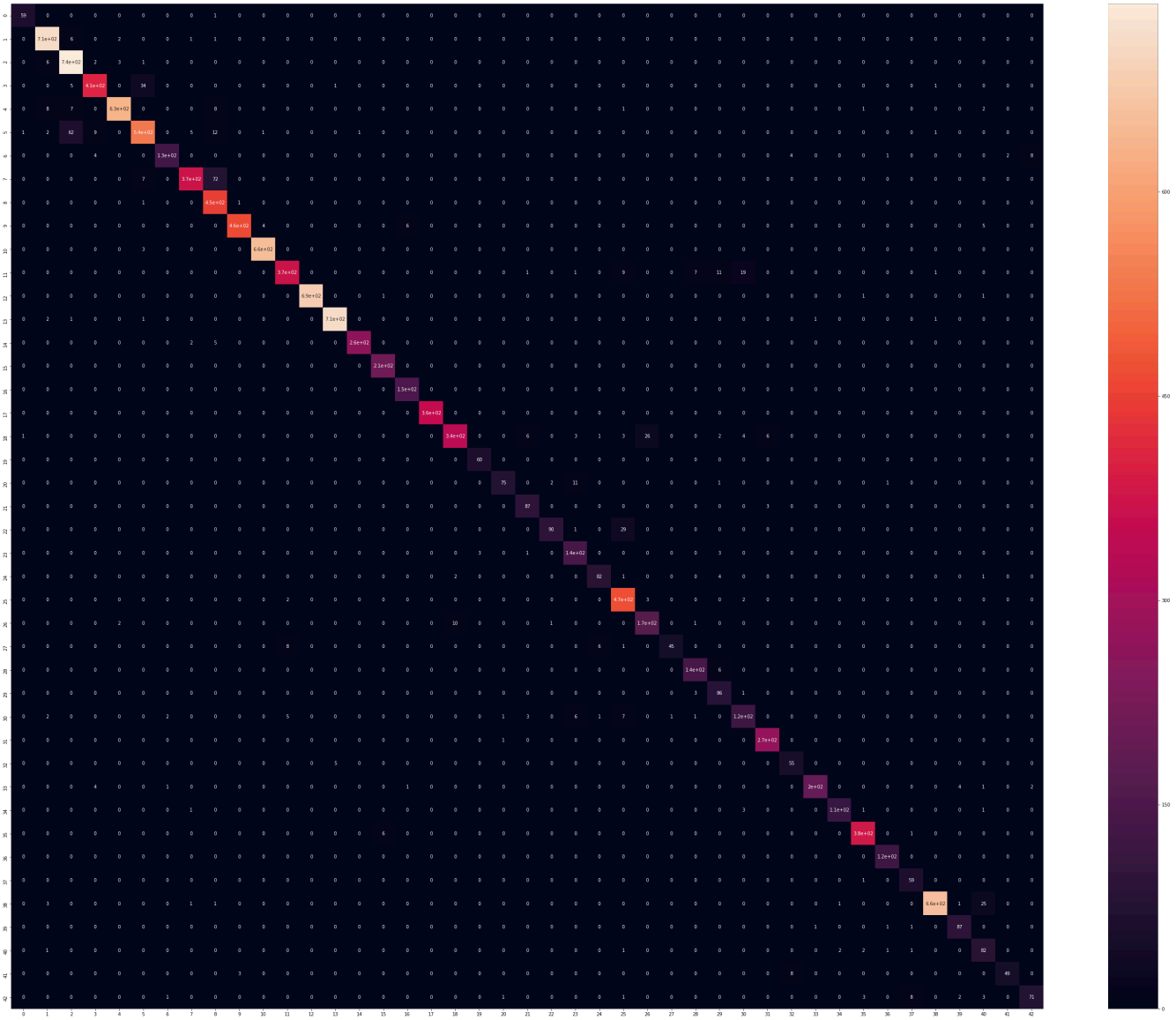
Solutions like Slov4 from Roboflow offer the whole training pipe line setup, including the data augmentation capability.

Appendix 1: precision recall on the test set

class	precision	recall	f1-score	support	sign description
0	0.97	0.98	0.98	60	Speed limit (20km/h)
1	0.97	0.99	0.98	720	Speed limit (30km/h)
2	0.90	0.98	0.94	750	Speed limit (50km/h)
3	0.96	0.91	0.93	450	Speed limit (60km/h)
4	0.99	0.96	0.97	660	Speed limit (70km/h)
5	0.92	0.85	0.88	630	Speed limit (80km/h)
6	0.97	0.87	0.92	150	End of speed limit (80km/h)
7	0.97	0.82	0.89	450	Speed limit (100km/h)
8	0.82	1.00	0.90	450	Speed limit (120km/h)
9	0.99	0.97	0.98	480	No passing
10	0.99	1.00	0.99	660	No passing for vehicles over 3.5 metric tons
11	0.96	0.88	0.92	420	Right-of-way at the next intersection
12	1.00	1.00	1.00	690	Priority road
13	0.99	0.99	0.99	720	Yield
14	1.00	0.97	0.99	270	Stop
15	0.97	1.00	0.98	210	No vehicles
16	0.96	1.00	0.98	150	Vehicles over 3.5 metric tons prohibited
17	1.00	1.00	1.00	360	No entry
18	0.97	0.87	0.91	390	General caution
19	0.95	1.00	0.98	60	Dangerous curve to the left
20	0.96	0.83	0.89	90	Dangerous curve to the right
21	0.89	0.97	0.93	90	Double curve
22	0.97	0.75	0.85	120	Bumpy road
23	0.87	0.95	0.91	150	Slippery road
24	0.91	0.91	0.91	90	Road narrows on the right
25	0.90	0.99	0.94	480	Road work
26	0.85	0.92	0.89	180	Traffic signals
27	0.98	0.75	0.85	60	Pedestrians
28	0.92	0.96	0.94	150	Children crossing
29	0.76	0.96	0.85	90	Bicycles crossing
30	0.81	0.81	0.81	150	Beware of ice/snow
31	0.97	1.00	0.98	270	Wild animals crossing
32	0.82	0.92	0.87	60	End of all speed and passing limits
33	0.99	0.94	0.96	210	Turn right ahead
34	0.97	0.95	0.96	120	Turn left ahead
35	0.98	0.98	0.98	390	Ahead only
36	0.97	1.00	0.98	120	Go straight or right
37	0.84	0.98	0.91	60	Go straight or left
38	0.99	0.95	0.97	690	Keep right
39	0.93	0.97	0.95	90	Keep left
40	0.68	0.91	0.78	90	Roundabout mandatory
41	0.96	0.82	0.88	60	End of no passing
42	0.88	0.79	0.83	90	End of no passing by vehicles over 3.5 metric tons
avg / total	**0.95**	**0.95**	**0.95**	**12,630**	

Appendix 2: Test set confusion matrix :

horizontal axis: classes expected
vertical axes : classes predicted
matrix contents: sample counts



Appendix 3: Training and validation accuracy

Training accuracy raises much slower due to ImageGenerator. Although the model seems to under-fit, it generalizes much better then after a conventional training : validation accuracy converges at 95% after 20 epochs:

```
training_steps_per_epoch 865
validation_steps 44
Epoch 1/20: - 50s - loss: 3.1149 - acc: 0.1655 - val_loss: 5.1507 - val_acc: 0.3348
Epoch 2/20: - 49s - loss: 2.2259 - acc: 0.3854 - val_loss: 2.2600 - val_acc: 0.5675
Epoch 3/20: - 53s - loss: 1.9101 - acc: 0.4713 - val_loss: 1.2193 - val_acc: 0.7068
Epoch 4/20: - 49s - loss: 1.7890 - acc: 0.5056 - val_loss: 0.5908 - val_acc: 0.8284
Epoch 5/20: - 48s - loss: 1.7053 - acc: 0.5287 - val_loss: 0.5044 - val_acc: 0.8525
Epoch 6/20: - 48s - loss: 1.6527 - acc: 0.5443 - val_loss: 0.4045 - val_acc: 0.8820
Epoch 7/20: - 48s - loss: 1.6101 - acc: 0.5561 - val_loss: 0.2348 - val_acc: 0.9266
Epoch 8/20: - 48s - loss: 1.5696 - acc: 0.5665 - val_loss: 0.2244 - val_acc: 0.9300
Epoch 9/20: - 48s - loss: 1.5404 - acc: 0.5749 - val_loss: 0.2217 - val_acc: 0.9291
Epoch 10/20: - 48s - loss: 1.5236 - acc: 0.5798 - val_loss: 0.1852 - val_acc: 0.9414
Epoch 11/20: - 49s - loss: 1.5057 - acc: 0.5839 - val_loss: 0.1777 - val_acc: 0.9443
Epoch 12/20: - 48s - loss: 1.5001 - acc: 0.5867 - val_loss: 0.1801 - val_acc: 0.9473
Epoch 13/20: - 48s - loss: 1.4851 - acc: 0.5895 - val_loss: 0.1742 - val_acc: 0.9473
Epoch 14/20: - 48s - loss: 1.4593 - acc: 0.5966 - val_loss: 0.1961 - val_acc: 0.9445
Epoch 15/20: - 48s - loss: 1.4546 - acc: 0.5996 - val_loss: 0.1804 - val_acc: 0.9505
Epoch 16/20: - 48s - loss: 1.4329 - acc: 0.6053 - val_loss: 0.1910 - val_acc: 0.9434
Epoch 17/20: - 48s - loss: 1.4354 - acc: 0.6033 - val_loss: 0.1775 - val_acc: 0.9470
Epoch 18/20: - 48s - loss: 1.4209 - acc: 0.6093 - val_loss: 0.1618 - val_acc: 0.9461
Epoch 19/20: - 48s - loss: 1.4255 - acc: 0.6082 - val_loss: 0.1538 - val_acc: 0.9543
Epoch 20/20: - 48s - loss: 1.4082 - acc: 0.6126 - val_loss: 0.1554 - val_acc: 0.9509
```

Appendix 4: data preprocessing, standardization

Standardization/normalization :

Features feeding neural networks should be on the same scale : this way weights won't have to swing widely throughout the training to fit the samples. It also prevents one feature not on the same scale then the others to affect adversely affect the weights calculation. Once data are normalized, weights are more uniform hence convergence is faster.

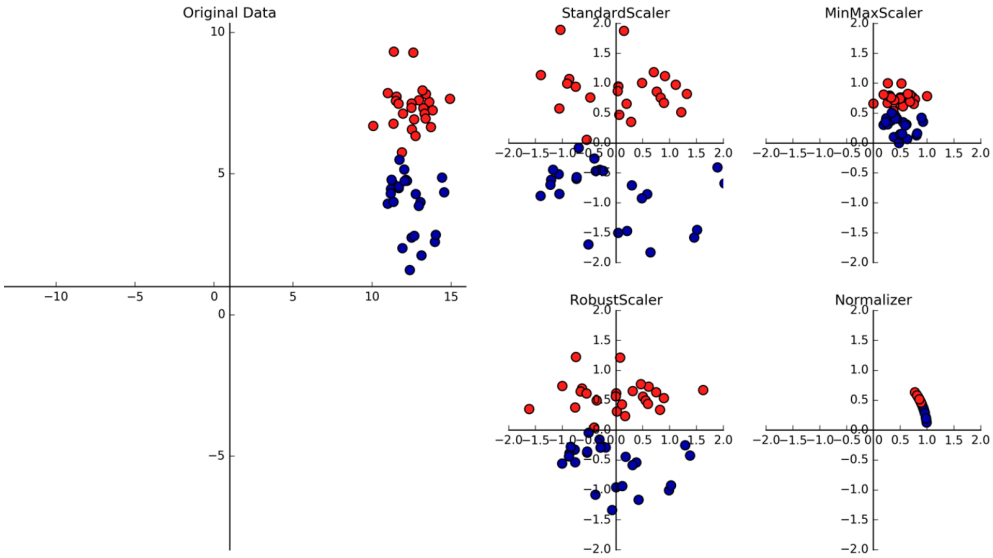
Best practice is to standardize the data.

standardization steps:

- calculate the mean and standard deviation on the training set
- apply (X_train-mean)/standard deviation) on the training set
- apply (X_test-mean)/standard deviation) on the validation set
- pickle (mean, standard deviation) : for later standardization of the test set and other samples before prediction

Once standardized features have a mean of zero and standard deviation of 1

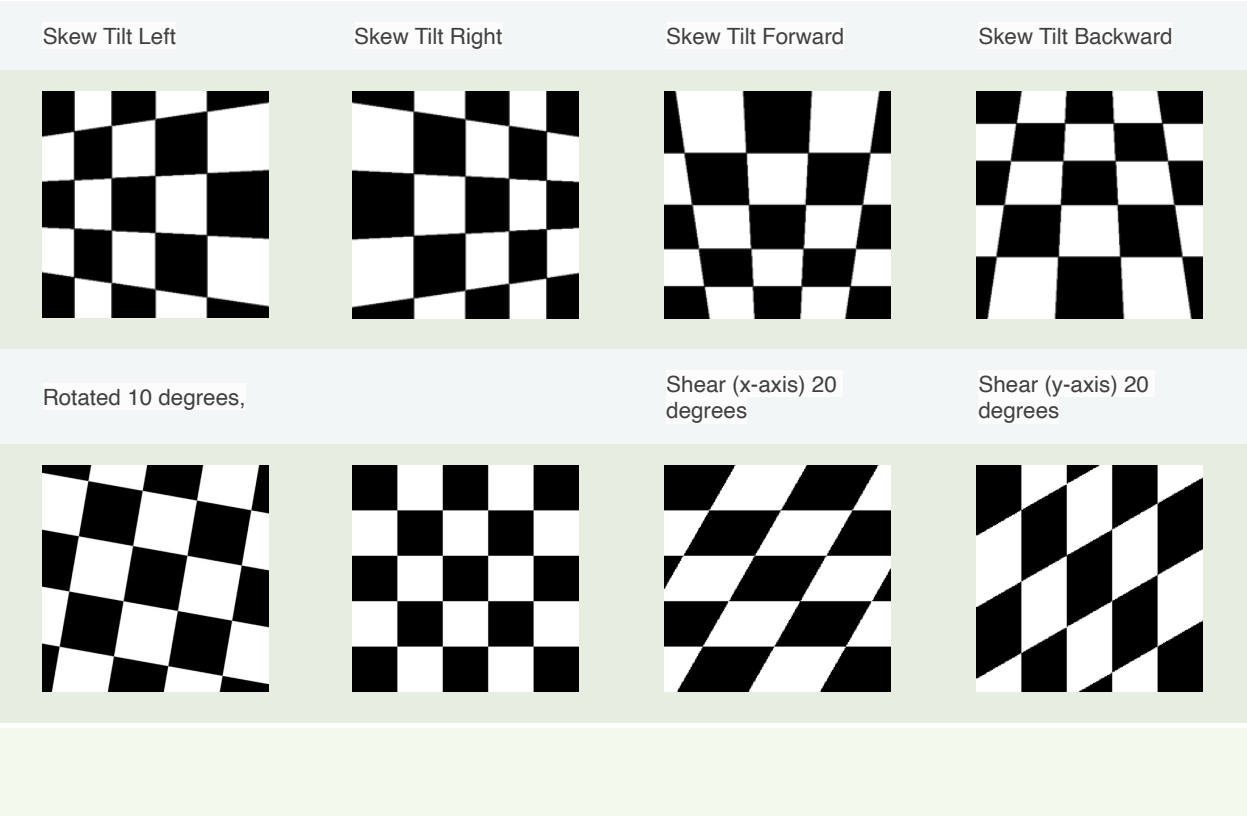
See below different forms of scaling including normalizations (courtesy of "python-data-science")



Appendix 5: data preprocessing , data set augmentation

To get an even number training sample per class, we apply and series transformation on existing image. Each transformation parameters vary randomly within a preset range . These ranges should emulate real live conditions (eg: rotation should not exceed +/- 10 degrees)

geometric transformations:



appearance transformations:

- color
- brightness
- contrast