

Report for project: System of linear equations solver with deep learning

Background and motivation

Over the course of the last months e-learning gets more and more to the focus of the broad population. The Covid-19 pandemic forces nearly every country in the world to apply restrictions on the number of people allowed to gather in public spaces including schools. One of the major effects of these lockdowns is the breakdown of public education as we know it. Students, teachers and parents have to turn to modern and innovative approaches for teaching on a remote basis.

This trend results in a rising demand for digital solutions in the education sector enabling students to learn at home and at their individual pace. Furthermore these solutions enable children in less developed countries to catch up with their peers in the so-called First and Second World without having access to a widespread and sophisticated public education infrastructure.

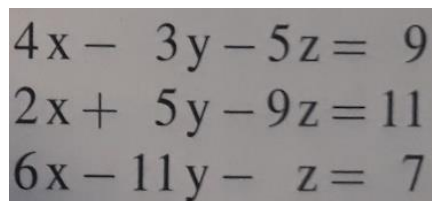
Mathematics is one of the essential domains in the course of every child's formal education providing the necessary skills to thrive in the modern technical and digitalized world. Systems of linear equations (SLE) are a central subject of every Mathematics curriculum. Enabling to solve problems ranging from flows in complex systems to analytical geometry.

There are already different projects out there tackling these problems. Google DeepMind released a paper in 2019 (Saxton et. al, 2019) stating a poor performance with establishing a neural net answering high-school math problems. Especially with problems of higher complexity like solving a 2D system of linear equations the three trained models output the right answer with a probability of 55% - 90%. System of linear equations in 3D were not examined at all. The DeepMind project used questions in free text format that have not been parsed before being used as input for the model. For Example:

What is $g(h(f(x)))$, where $f(x) = 2x + 3$, $g(x) = 7x - 4$, and $h(x) = -5x - 8$?

Additionally models without any mathematical knowledge implemented were used.

Other industry approaches are already in use like the photomath Mobile App (photomath, 2020). These programs reduce some of the complexity by allowing only non-free-text examples like a well formatted equation or a SLE. But on the other hand they add some complexity by allowing users to input scanned and cropped handwritten or printed mathematical tasks.


$$\begin{array}{rcl} 4x - 3y - 5z & = & 9 \\ 2x + 5y - 9z & = & 11 \\ 6x - 11y - z & = & 7 \end{array}$$

Officially released accuracy scores for those apps are not available but self-performed test show scores of nearly 100%. Most probably these apps parse the given information to lines and even single mathematical symbols and solve it with the given mathematical tools. So deep learning is only used for image recognition and parsing.

This project tries to combine both worlds and implement a program that allows to input a scanned pre-cropped image of either handwritten or printed systems of linear equations and retain the solution by parsing each single symbol from the equation, passing it to an image classifier and calculating the solution with the help of the inverse matrix representing the parameters of the equations.

Accuracy on correct found solutions for scanned SLEs should be used as an overall metric for the success of the project. But already at this stage there should be some emphasis on the difficulty of achieving a good performance. A SLE would be solved correctly if it would be parsed (p_{parsed}) completely correct and each single symbol would be classified correctly ($p_{classified_i}$). Given a SLE in the image above with 31 symbols the probability of success is:

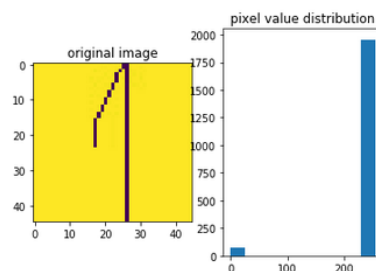
$$p_{success} = p_{parsed} * p_{classified_i}^{31}$$

If only an accuracy score of 50% should be achieved and the probability of a correct parsed image is 80% the accuracy on correct classified images must be greater than 98,5%. Increasing significantly with each additional symbol $\left(\sqrt[n]{\frac{50\%}{80\%}}\right)$.

So this project should aim on finding a procedure for parsing and classifying SLEs in general enabling future projects to concentrate on each of these steps separately trying to maximize performance measured by accuracy scores. Additional implementation of these technics will be performed as a command line tool. This part of the project could also be the center of a future project moving algorithms to a server-based service, adding a frontend and enabling users to receive feedback via the Internet in near real time.

Data

Two major datasets are required. The first one consists of handwritten mathematical symbols used as inputs for training of the image classifier. These images should represent a wide range of possible variants of each symbol used by humans or in printed books. A dataset provided by Xai Nano on Kaggle is used in this project containing images of various symbols ranging from basic mathematical operators like '-' and '+' to more specific symbols like Greek symbols α and γ each stored in a folder given the name of the symbol. The number of images in this dataset is heavily unbalanced among the symbols for example 33,997 images are included for the symbol '-' and only 5,870 images for the variable 'z'. The distribution of images among the symbols seems to reflect the statistical distribution of symbols used in mathematical tasks. As a first approach in reducing complexity we restrict these symbols to the common mathematical variable identifiers x, y and z as well as integer numbers consisting of the literals 0, 1, ...9 as operands and -, + and = as operators. A subset is created manually by using only the required folders. A closer examination reveals that the images have the shape of 45x45 pixels containing a single channel with either pixel values of 0 or 255 (binary image).



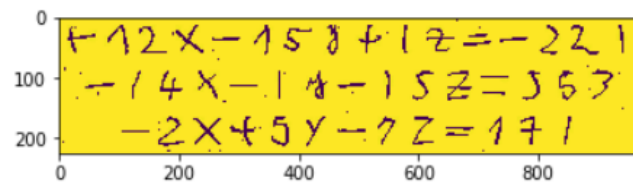
Additionally a very wide range of variants for each symbols is contained. This fact should enable the image classifier to generalize well. But on the other hand there are some similar variants for different symbols that look alike like the images for a '2' and 'z' shown below.



Eventually this fact may lead to a poor performance in image classification and could be mitigated by manually cleaning the dataset.

As a second dataset a set of scanned SLE images used as inputs for the command line tool is needed. These images have to contain multiple variants of each mathematical symbols that can be used in an SLE representing different humans writing these equations or styles that are used in printed books too. Because of the fact that there is no labeled dataset containing images of SLEs on the Internet such a dataset has to be created either manually by scanning handwritten or printed SLEs and labeling them or synthetically by concatenating images of mathematical symbols in such a way that they represent a complete SLE. In this project the second approach is used. Scanned SLE images are created by randomly creating a digital representation of a SLE and then concatenating the images representing the used operators and operands in the SLE. Additionally some random noise has to be added to the images mimicking real world images that mostly contain some kind of fragments

created by dust or other small fragments on camera lenses or sensors. The visualization below shows such a synthetically created SLE image with some random added fragments:



Solution

The solution to the given problem consists of five major tasks:

1. Create synthetical dataset of scanned SLE images.
2. Parse SLE images to single symbol images.
3. Train an image classifier.
4. Present parsed images to classifier and get predictions on symbols in these images, create a digital representation of the SLE and solve it if possible.
5. Combine steps 2 and 4 in a command line tool enabling users to input an image file and receive a solution for their SLE.

Steps 1-4 were implemented in Jupyter notebooks:

- create_synthetic_sle_dataset.ipynb
- parse_sle_images.ipynb
- classification_with_convolutional_net.ipynb

Emphasis was placed on modular code that can be used for the command line tool (solver.py). All code was well documented using comments and self-explaining variable and function identifiers. All data created by the dataset creation and parsing step will be stored locally in a data subfolder. Model parameters will be stored in a model subdirectory. The command line tool and all associated files reside in a designated subfolder.

```
---capstone_project
  ---command_line_tool
    sle_solver.py
    image_classification.py
    image_parsing_helpers.py
    data
  ---data
    --manually_scanned_sles
    ...
    --synthetic_scanned_sles
    ...
    --sle_symbols_parsed
    ...
  ---model
    model_dict.pth
  jupyter_notebooks.ipynb
```

Create Synthetical SLE image dataset

As a first step a SLE has to be created containing randomly chosen parameters for the variables x, y and z and a vector b.

For example a presentation of the SLE:

$$x - y + 8z = 12$$

$$4x - y + 0z = 2$$

$$-x - y + 6z = 4$$

is given by

$$\begin{pmatrix} 1 & -1 & 8 \\ 4 & -1 & 0 \\ -1 & -1 & 6 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 12 \\ 2 \\ 4 \end{pmatrix}$$

or more general

$$A \times x = b$$

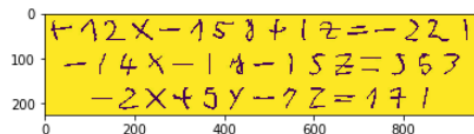
A random vector x could be generated as a solution for a SLE. In the next step random parameters for matrix A are created. A SLE with the solution vector and matrix given is then calculated by adding the vector b as a product of A and x .

One more thing has to be checked to ensure that the SLE has the given solution. A SLE can be undetermined which means that there are less linearly independent row vectors than variables a solution must be found for. In this case we need 3 linearly independent vectors. Mathematically the eigenvectors of a linear independent system of equations are all unequal to zero. This condition is checked and if needed a new set of A , b and d is calculated.

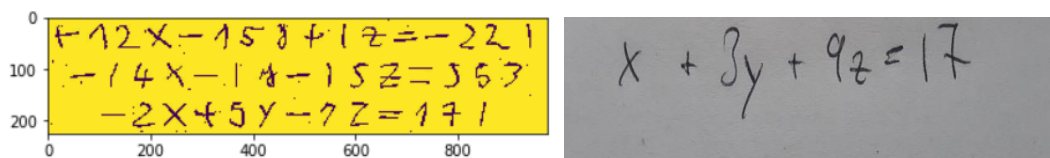
With the given random SLE in hand images are chosen randomly from the Kaggle dataset that show exactly the symbols used in the equations. These images are then concatenated and some padding is added to get one single image showing the whole SLE.



At this step it becomes obvious that the kaggle dataset does not contain scanned handwritten images but images created by writing on a touchscreen or something similar because of the thin strokes used. This issue could be solved by using a technique called dilation implemented in the popular python computer vision package openCV. Dilation is nothing more than filling pixels with the value of nearby pixels. So values near the thin strokes forming a symbol are decreased leading to a wider stroke as shown in the image below.



The discussed steps lead to images that seem a bit to perfect for scanned images. So some noise is added by superimposing the image with a mask containing fragments of random size and location. The images below show an image with artificial fragments and a segment of a manually scanned image with some real world noise in the lower part of the image.



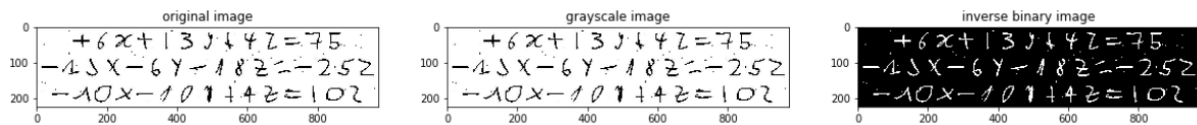
The major advantage of this synthetic creating process comes to hand now. All symbols contained in these newly stored images are well-known and can be exported to a file annotating each image of the newly created dataset. Additionally some aggregated data like the number of symbols in each line is added to allow an easy evaluation of the next task.

file	n_rows	n_symbols	symbols	A	b	x	dilation	n_fragments	fragment_size
sle_synthetic_2020-10-23-09-44-37496.jpg	3	[14, 15, 15]	[[-, 3, x, -, 1, 7, y, +, 3, z, -, -, 5, 6], [...	[[-3 17 3], [-20 12 -9], [18 14 5]]	[-56 -54 154]	[12 2 -18]	3	100	5
sle_synthetic_2020-10-23-09-44-39194.jpg	3	[15, 15, 14]	[[+, 5, x, -, 1, 4, y, -, 1, 6, z, -, 1, 5, 9]...	[[5 -14 -16], [14 15 -11], [13 -8 -8]]	[159 67 -39]	[-11 3 -16]	3	100	5
sle_synthetic_2020-10-23-09-44-41341.jpg	3	[15, 15, 16]	[[-, 1, 7, x, -, 1, 3, y, +, 0, z, -, 3, 0, 1]...	[[-17 -13 0], [0 8 -16], [-11 -20 -17]]	[301 -112 357]	[- 7 -14 0]	3	100	5
sle_synthetic_2020-10-23-09-44-43405.jpg	3	[14, 16, 14]	[[-, 5, x, -, 1, 1, y, +, 1, 2, z, -, 2, 8], [...	[[-5 -11 12], [10 -14 10], [8 16 0]]	[28 -82 184]	[-3 13 13]	3	100	5
sle_synthetic_2020-10-23-09-44-44748.jpg	3	[15, 14, 16]	[[+, 1, 2, x, +, 0, y, -, 3, z, -, -, 1, 2, 0]...	[[12 0 -3], [13 -11 -3], [5 18 -19]]	[-120 152 -605]	[- 7 -19 12]	3	100	5

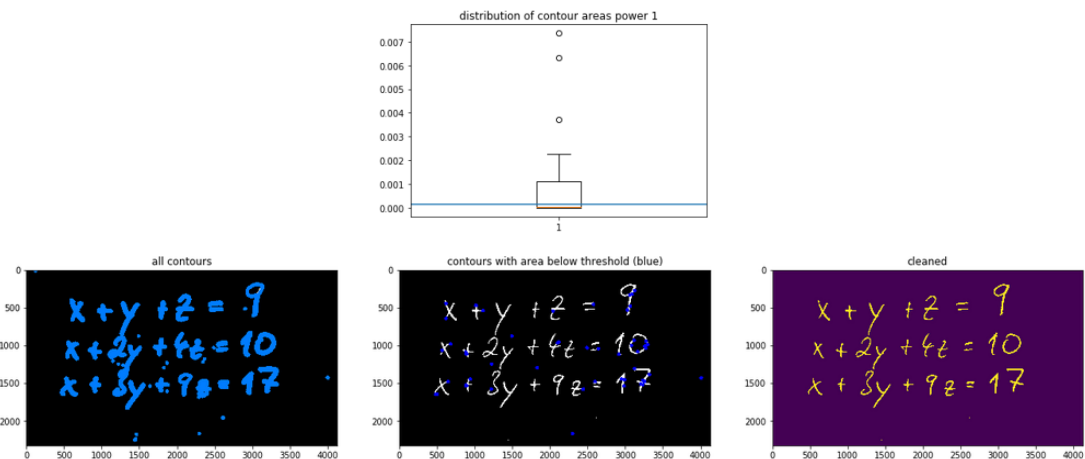
Parse SLE images

In this part the steps leading from a locally stored image to a set of images showing only a single symbol are laid out. To arrive at these results the images have to be preprocessed to enable common computer vision algorithms to detect shapes in the images that hopefully coincide with mathematical symbols.

First the images are converted to inverse binary, single channel images by combining different image channels and applying a thresholding process on pixel values resulting in values of either 0 or 255.



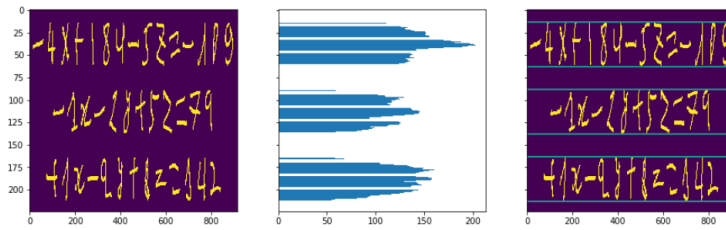
Understanding of a concept called contours in computer vision is important for the next steps. Contours are lines indicating the boundaries of areas of same pixel intensity. The first practical use in contours is finding small fragments and removing them from an image. Therefore, all contours are extracted and their areas are calculated. Based on a statistical threshold (10% of mean contour area) small contours are removed.



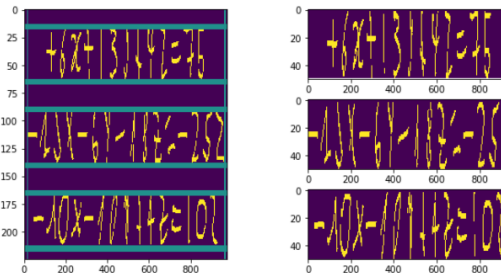
Now the 3 lines of the SLE must be parsed from the cleaned image. In this context the lines could be seen as a cluster of high pixel values in one of the two dimensions. By calculating a histogram of pixel values and applying a thresholding technique borders for each line can be extracted. For example an upper border is a line that has a mean pixel value below a certain threshold with a line following with a value above this threshold.

```
borders_0_row = [i - padding for i in range(len(hist_row)-1) if hist_row[i] <= threshold and hist_row[i+1]
```

The visualization below shows this approach for one dimension extracting the upper and lower border.



By applying this technique to both dimensions three images can be extracted showing only a single line each.



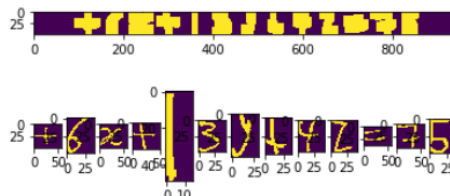
Best results were found with a combination of a high degree of dilation and a low threshold. More noise in the images will lead to problems extracting borders. A single fragment between two lines could result in a 4th border pair to be extracted. A mitigation approach could use dynamic values for dilation iterations or thresholds.

Given the single images finding the single symbols will be the next tasks. Again, the program relies on contours. But now some more details regarding contours must be specified. If an algorithm extracting contours is used on these images it will find not only contours from continuously drawn symbols like a '2' but also inner areas of a '0' or two contours for an equal sign '='. To mitigate this issue the hierarchy of contours is used. Hierarchy simply denotes the inheritance between contours. An inner contour of a '0' is a child of the contour of the stroke itself. If only parent contours are used inner contours are discarded.

```
contours, hierarchy = cv.findContours(image_dilation, cv.RETR_EXTERNAL,
cv.CHAIN_APPROX_SIMPLE)
```

The next problem can be solved with the help of dilation again. The two strokes of an equal sign are detached from another and cannot be identified as a single contour. But if dilation is used in the y direction pixels are filled until both bars are attached to each other. If a contour algorithm is used on this dilated image it finds only one contour. The edges of this contour can be applied on the original file and thereby the non-dilated equal sign as a whole is extracted.

```
kernel_x = np.ones([1,2])
image_dilation = cv.dilate(image, kernel_x, iterations = dila_iterations_x)
```



The result is a set of images showing only a single symbol each. But all these images are only as wide and high as the area the symbol was occupying in the original image. The image classifier from the next step needs these images to be of equal shape. Each image is resized to 28 pixels in its largest dimension and the other dimension is filled with empty lines or columns (padding). This ensures that the aspect ratio does not change. The size of 28x28 pixels is chosen to enable a combination with the MNIST digits set in a follow up project.

An evaluation is taken based on the annotations added to the synthetically created SLE images and the symbols found via parsing. The accuracy of SLE images parsed correctly (correct number of symbols was parsed for each line) is calculated and is around 91% for multiple sets containing each 200 synthetical images. This is a good

result and a more detailed look shows that in most cases only a single line shows a false number of parsed images differing by only 1.

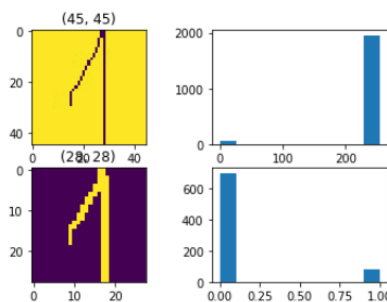
But results of this step must be seen in the context of the dataset creation process. Since the systematic of this process (how much noise is generated, padding added) was mostly deterministic and well-known scores on real-world datasets should be lower.

A follow up project could focus on a more realistic image postprocessing like rotations or other transformations during the SLE generation step. Additionally, setting thresholds manually like the pixel threshold for border identification or number of dilation steps must be generalized to fit a broad range of input images. These parameters could be passed to an additional machine learning model. Furthermore the restriction on integer parameters removes a lot of complexity from the real world like the possibility of fractional parameters or variable names also common like x_1 , x_2 , and x_3 instead of x , y , and z .

Training an image classifier

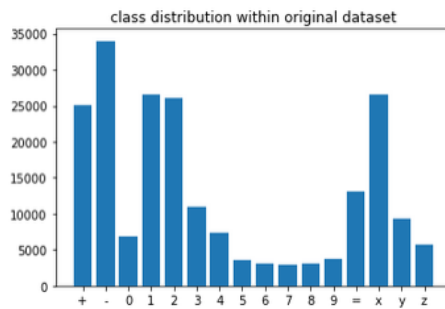
All symbols were parsed from the SLE as images but without knowing which symbol each image shows a solution for the SLE cannot be calculated. This task is a perfect challenge taken on with the help of a neural net. Such a model should be trained on a dataset different from the set it should perform inference on. In this project the image dataset from Kaggle is used for training, validation and a short testing. But the overall evaluation is done with the help of the parsed images.

First images have to be preprocessed before being used as inputs for training. Images must be resized to 28x28 pixels with only a single channel and pixel values rescaled.

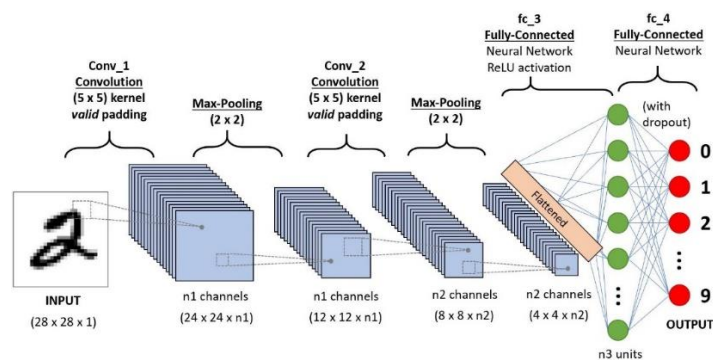


The neural itself is implemented with the help of the pytorch package. Pytorch models need data to be in a format called Tensors. To transform the Kaggle dataset to tensors we use the pytorch DatasetFolder class. It imports the images from the multiple folders and labeling them according to the folder names. Additionally a preprocessing function performing the above mentioned transformation process and the toTensor transformations are applied. Due to the huge dimensions of the dataset it is acceptable to drop 50% of the whole dataset resulting in a smaller subset. The resulting subset is then divided into two more subsets, one for training and one for validation.

The visualization below shows the imbalance of the dataset. Images showing a '-' for example are seven times more common in the subset than images for '7'. To resolve this issue weights can be calculated as the inverse of the frequency of each class in the subset. Images are not passed to models in pytorch all at once but with the help of so-called dataloaders. These dataloaders are randomly sampling the data using the above calculated weights. Additionally a batch size can be specified. A batch can be seen like a basket of n datapoints and their labels (targets) according to the batch_size that is passed to the model at each iteration step.



Now it's time to define the Convolutional Net used for inference. These convolutional Nets are often used in image classification due to its superior performance. A convolutional net is an update to commonly used fully connected neural nets a convolutional layer and a pooling layer upstream of fully connected layers. The convolutional layer performs some feature extraction from the images and the pooling layer reduces the number of features thereafter. The code below shows the different layers of the Neural Net.



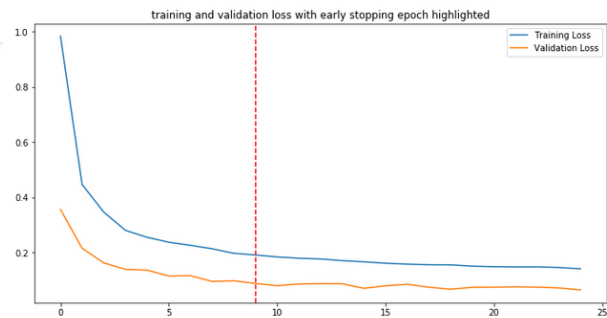
```
ConvNN(
    (activation): ReLU()
    (conv_layers): Sequential(
      (0): Conv2d(1, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU()
      (2): MaxPool2d(kernel_size=3, stride=3, padding=1, dilation=1, ceil_mode=False)
      (3): Dropout(p=0.1)
      (4): Conv2d(4, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (5): ReLU()
      (6): MaxPool2d(kernel_size=3, stride=3, padding=1, dilation=1, ceil_mode=False)
      (7): Dropout(p=0.1)
      (8): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (9): ReLU()
      (10): MaxPool2d(kernel_size=3, stride=3, padding=1, dilation=1, ceil_mode=False)
      (11): Dropout(p=0.1)
    )
    (fc_layers): Sequential(
      (0): Linear(in_features=64, out_features=256, bias=True)
      (1): ReLU()
      (2): Dropout(p=0.1)
      (3): Linear(in_features=256, out_features=256, bias=True)
      (4): ReLU()
      (5): Dropout(p=0.1)
      (6): Linear(in_features=256, out_features=16, bias=True)
    )
  )
)
```

After initialization of the net it can be used for training in the preprocessed Kaggle images. To prevent overfitting an early stopping mechanism was implemented stopping after a sum of 3 epochs without a 10% percent decrease in validation loss.

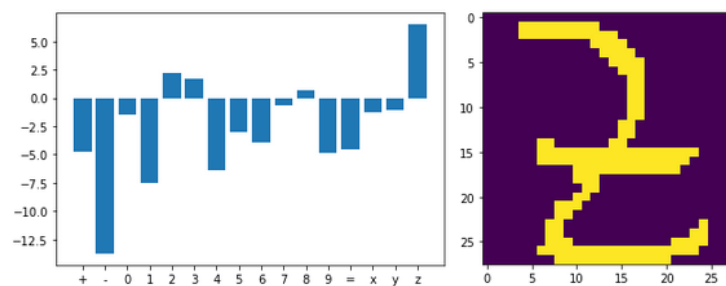

```

### Start training ###
Epoch: 1, Train Loss: 0.9840055360864478 Val Loss: 0.35560296303386174
Epoch: 2, Train Loss: 0.44589130463074034 Val Loss: 0.21446742344959616
Epoch: 3, Train Loss: 0.34576402430386693 Val Loss: 0.16167401372835846
Epoch: 4, Train Loss: 0.2795191487882644 Val Loss: 0.13807556966292153
Epoch: 5, Train Loss: 0.25444772147051714 Val Loss: 0.1352937314934625
no improvement
Epoch: 6, Train Loss: 0.23663605862679885 Val Loss: 0.11402670506002574
Epoch: 7, Train Loss: 0.22531399137238872 Val Loss: 0.1153959532410567
no improvement
Epoch: 8, Train Loss: 0.21277711190240514 Val Loss: 0.09441140503737286
Epoch: 9, Train Loss: 0.1963858026568033 Val Loss: 0.09663772382062351
no improvement
### Early stopping ###
Epoch: 10, Train Loss: 0.19037022791845667 Val Loss: 0.0870423419951585
Epoch: 11, Train Loss: 0.18317602671701172 Val Loss: 0.07901143299167966
Epoch: 12, Train Loss: 0.17852291139339305 Val Loss: 0.08509875559637993
Epoch: 13, Train Loss: 0.1759140640680569 Val Loss: 0.08649414980995097
Epoch: 14, Train Loss: 0.16988895203340632 Val Loss: 0.08609471460483198

```



Hidden layer dimensions, convolution layer parameters (kernel size, padding, channel dimensions) and early stopping parameters were found in the process of improving train loss while preventing overfitting in test data and can be found in the Jupyter notebooks and in the above image showing the architecture of the neural net.



An evaluation on the test data showed an accuracy on correct label predictions around 96%. The visualization below shows some predictions for a part of a batch from the testset.



An analysis of the most mislabeled symbols shows that mostly 'z', '-', '=', and '+' are mislabeled. An even closer look shows that some mislabeled images are even hard to interpret by humans like the image of a 'z' in the left upper corner. Scores of Convolutional nets used for inference on the MNIST dataset can be used as a benchmark for the results. The simpler models score test error rates around 1%. But these scores must be seen in the context of a much cleaner dataset and greater resources in form of time and computational power. Additionally this net was the first Convolutional Net implemented by the author.

Further evaluation could be done by letting the ConvNN perform training and inference on the MNIST dataset and compare the results directly to the scores published on the MNIST page.

Classification on parsed images and solving the SLE

This step sums up all of the previous tasks by using inference on the parsed images and calculating a solution for the original SLE. Given the accuracy score of the previous steps a meager result is anticipated. To compute the correct result of the SLE all symbols have to be parsed correctly (score: 92.5%) and labeled correctly (score:

96%). A SLE with double characters for parameters has 12 symbols (+/- XX variable 3 times = XX) on each line. This means the probability of parsing and digitalizing an image correctly is only around $92.5\% * 96\%^{36} = 2\%$.

The same preprocessing steps as used with the Kaggle dataset were used with the parsed images.

In depth analysis of the synthetic created images shows that for example '=' is often misclassified as '=' with a much higher rate than in the test image dataset. Reason for that was, that '=' symbols were parsed to images that showed the sign in the upper half of the image similar to the upper half an "0" image. This flaw led to a change in the image parsing process outputting images showing each symbol exactly centered.



To visualize how significantly single misclassified images can lead to complete failure on an SLE all steps of the processing are shown with the help of a single SLE. The image below shows a synthetically created SLE image.

$-20x - 20y + 14z = -680$
 $-7x - 15y - 17z = 120$
 $+18x - 12y + 18z = -300$

All symbols were parsed without errors but in the classification step two failures were made at the original symbols 'z' and '8' in the first line classifying them as '2' and 'y'.



was classified as '2' and



as 'y' leading to a SLE and a matrix of:

$$\begin{aligned} -20x - 20y + 14\mathbf{2} &= -6\mathbf{y}0 \\ -7x - 15y - 17z &= 120 \\ +18x - 12y + 18z &= -300 \end{aligned}$$

$$\begin{aligned} A &= \begin{bmatrix} -20. & -20. & \mathbf{xxx} \\ -7. & -15. & -17. \\ 18. & -12. & 18. \end{bmatrix} \\ b &= ['-6\mathbf{y}0', '120', '-300'] \end{aligned}$$

with a missing parameter for z and a false component for b in the first line. Thus preventing a correct calculation of the solution. The correct digital representation of the SLE would have been:

$$\begin{aligned} -20x - 20y + 14\mathbf{z} &= -6\mathbf{8}0 \\ -7x - 15y - 17z &= 120 \\ +18x - 12y + 18z &= -300 \end{aligned}$$

This original SLE is now used to explain the process of finding a correct solution for a digital SLE. First lines are parsed according to the positions of the equal signs. The left sides are taken to compute A and the right sides already show the components of column vector b.

Then the left sides are parsed according to the positions of variables x, y, and z outputting the elements of A.

$$A = \begin{pmatrix} -20 & -20 & 14 \\ -7 & -15 & -17 \\ 18 & -12 & 18 \end{pmatrix}, b = \begin{pmatrix} -680 \\ 120 \\ -300 \end{pmatrix}$$

Given A and b the solution can be calculated according:

$$\text{solution} = A^{-1} \times b = \begin{pmatrix} -20 & -20 & 14 \\ -7 & -15 & -17 \\ 18 & -12 & 18 \end{pmatrix}^{-1} \times \begin{pmatrix} -680 \\ 120 \\ -300 \end{pmatrix} = \begin{pmatrix} 10 \\ 10 \\ -20 \end{pmatrix}$$

Command line tool

As an implementation of the findings specified above a command line tool should be created. A scanned SLE image and the trained model parameter should function as inputs enabling the calculation of a solution used as output of the tool. Therefore the tool must implement the logic from the image parsing and the symbol inference and solution calculation steps. Basically all steps used in analysis were implemented as modular functions enabling the 1:1 usage in the command line tool.

Only input and output logic must be implemented. The script should be called with a command like:

```
python sle_solver.py \path\to\sle\image \path\to\model\parameters
```

The API is specified with the help of the python argparse package. Arguments can be specified individually with additional information like default values or if they are required as input.

```
parser.add_argument("image_path", help="path to the image file")
```

Then only the functions residing in helper scripts must be called one after another:

```
#parse the images
print("### parsing image ###")
src = cv.imread(image_path)
image = src.copy()
print("#transforming image")
image_transformed = pars_helpers.transform_image(image, False)
print("#cleaning image from fragments")
image_cleaned = pars_helpers.remove_fragments(image_transformed, fraction = 0.2,
plot = False )
print("#searching for borders")
borders_x, borders_y = pars_helpers.find_borders(image_cleaned, threshold = 1,
dilation = 30, padding = 1, plot = False)
print("#extracting equations as sections")
sections = pars_helpers.extract_sections(image_cleaned, borders_x, borders_y, plot
= False)
...
```

The script outputs it's most probable representation for A and b to the user. If an element could not be parsed it is set to 'xxx'. If a solution can be calculated from this representation it will be shown.

```
###solution found ###
### results ###
+0x+9y-10z=-159
+11x+19y-13z=-254
-13x-14y-9z=61
A [['+0' '+9' '-10']
['+11' '+19' '-13']
['-13' '-14' '-9']]
b ['-159' '-254' '61']
### the suggested solution is: ###
x [ 3. -11.  6.]
```

Final evaluation

Now it's time to feed our tool with a larger dataset of synthetically created images. Using asset of 2000 images leads to an accuracy score of 11.5% percent correctly parsed and classified images. This result is even 5 times the accuracy that was anticipated.

Additionally some tests were performed by using the photomath app on the synthetic dataset showing a score of 0%. The app was not able to parse the equations. But both programs could not be compared 1 by 1 because photomath app accepts a wide range of mathematical tasks and this project was artificially reduced to only apply to SLEs. Manually scanned images of SLEs were not examined at all due to the moderate performance on the synthetical dataset. But performance scores near 100% of the photomath app on well formatted SLEs show that better scores are possible using more complex approaches and models.

A follow-up project should focus on these improvements as well as on real-world datasets. Classification and parsing performance must be increased significantly to improve overall performance:

$$p_{success} = p_{parsed} * p_{classified_i}^n$$

Conclusion

Curiosity in implementation of a state-of-the-art Convolutional Net solving a real world problem was the main motivation for this project. But normally real-world problems are still problems because the solution could not be found on well-trodden paths. The SLE solver project developed a well-documented outline for the given task but failed measured at accuracy scores on correct solved SLEs.

Industry standard tools like the photomath app score nearly perfect accuracy scores but were most probably developed by a highly experienced team of developers given access to huge resources compared to the resources available in development of the given project. Splitting the overall score back into the parsing and the classification score it could be reasonable to speak of a certain amount of success overall.

Follow up projects can start from this baseline. Adding complexity as well as increasing accuracy scores should be central points to these projects. For example, following topics should be tackled:

- create a real-world dataset of labeled SLE images (both handwritten and printed)
- automate the image parsing step regarding parameter selection
- evaluate these parameters on the real-world dataset
- increase image classifier scores (benchmark are still scores from the MNIST webpage)
- establish a server-side sle solver tool, create API and frontend

References:

image of a convolutional neural network: Sumit Saha (<https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>)

photomath, <https://photomath.app/en/>, 17.10.2020

Saxton, D. et al. (2019), <https://arxiv.org/abs/1904.01557>

Xai Nano, <https://www.kaggle.com/xainano/handwrittenmathsymbols>, 18.10.2020

LeCun et al., <http://yann.lecun.com/exdb/mnist/>, 17.10.2020