Smart Lineart

AI/neural network project for Krita

Agata Cacko

01.03.2024

Spis treści

1. Short description **2**

2. Purpose **2**

3. Requirements **3**

4. Implementation **4**

4.1. Short description of convolution 5

5. Challenges **5**

5.1. Choosing the right network 5

5.1.1. The network described in the article 5

5.1.2. Using a custom-designed network 6

5.2. Brush size/line thickness 6

5.3. Brush preset/style 7

6. Technical details **9**

6.1. Training 9

6.2. Inclusion in Krita 9

7. Estimated effort **9**

1. Short description

The goal is to create a tool for artists that converts a sketch into a lineart. Under the hood the tool will use a neural network for conversion. It will only contain convolutional layers and it should allow for customization of the output (line thickness, brush style etc.).

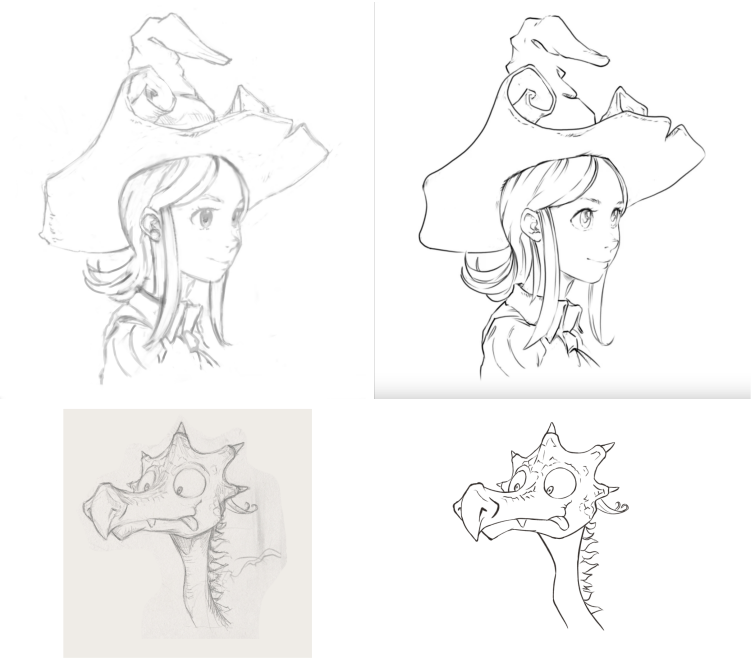
2. Purpose

The intent of this project is to provide artists with a quick way to convert their sketches into high quality lineart.

The process of creating a lineart from a sketch is often tedious and, depending on the artstyle, may not have that much creative or artistic thought behind it, which makes it perfect for automation.

Currently, without any automation, the usual process for an artist look like this:

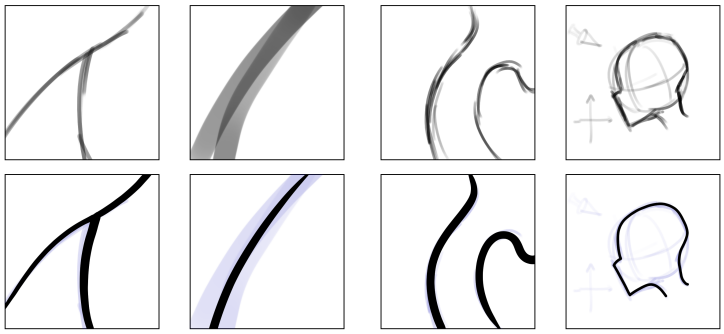
1. Making a sketch.
   - Depending on an artist, it can be more rough or more precise. We are only considering precise sketches here. They don't need to contain all the details, though.
   - It is usually either made with lower opacity brushes, or pencil-like, textured brushes.
2. Drawing a lineart over it, using stabilizers to ensure nice, smooth lines. It's common to either add details or fix some mistakes of the sketch at this stage.
3. (Optional) Add color and post-processing.



Pictures made by David Revoy (CC-BY),
except the dragon sketch which was made by David Revoy and Lynx3D (CC-BY).

Our solution will aid at drawing the lineart. Note that the artist will know in advance that they're going to use a tool, so we can assume that the sketch will be the most suitable for automated inking: it will not have any shading or cross-hatching, and it will not have any more mistakes. The artist will be able to fix mistakes on the final lineart, or go back to the sketch layer, fix the sketch, and run the automated inking tool again. The artist will likewise be able to add more details to the lineart using a suitable brush for seamless look, and proceed to the next stage of adding colors and post-processing as usual.

To be successful, the tool must create smooth lines and the result must already look good without any changes. It should simplify the sketch by merging lines that are close together, and it should drop any lines that are very faint.



How the tool should convert the sketch into lineart

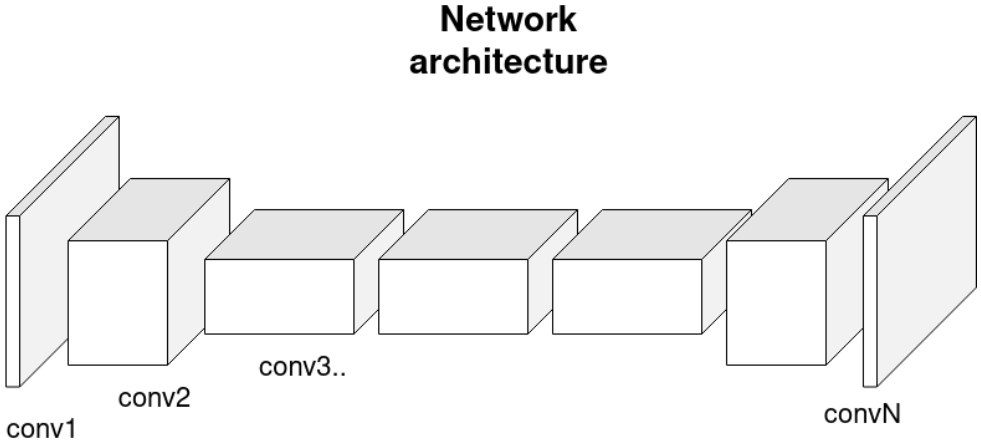
3. Requirements

The main idea of implementation comes from the article "Learning to Simplify: Fully Convolutional Networks for Rough Sketch Cleanup" by Simo-Serra and Iizuka, 2016, `https://esslab.jp/~ess/publications/SimoSerraSIGGRAPH2016.pdf`. However, due to the fact that we want to apply it to practical use, there are several places where our goals differ from the authors'.

- Common goals:
  - it can be used on an image of any size (therefore we should avoid fully-connected/dense (even with dropout) layers, and the network should be fully convolutional);
  - it should merge multiple sketch lines into one;
  - it can work on a desaturated version of the sketch (note: needs to be proper desaturation, based on HSY' or similar, not on HSV);
- Difference in output requirements:
  - the result should be appealing enough to be left as the final lineart in the painting (note that the results in the article looked more technical, but that was also apparent in the dataset they used; using more artistic target images should result in more appealing results from the trained network as well);

  - the lineart that it produces doesn't require it, but can be further edited by the artist (in fact, a workflow of converting the sketch into a lineart, coming back to the sketch to change some details, and converting it again to a lineart should be possible);
  - it *doesn't* need to handle shading, crosshatching and similar features of the initial sketch: the most common usecase would be to create sketch and immediately turn it into lineart, and artists will know this and draw sketches that give the best results (and not confuse the algorithm);
  - There must be a way to customize the result: thickness of lines, the style of lines etc.– which notably wasn't addressed in the article and isn't trivial to add, but seems necessary for our project;
- Difference in performance requirements:
  - it should allow processing the image on all kinds of CPU, and preferably on a wide range of GPUs;
  - it shouldn't take more than, let's say, 1GB, or maybe 2GB of memory – both on GPU and CPU, on GPU due to hardware requirements (see `https://store.steampowered.com/hwsurvey/`, and on CPU because RAM is used by every other program on the system and the free space might be limited as well;
  - processing the image shouldn't take longer than a minute on a standard laptop;
  - we can assume A4 300dpi as the default, but not only image size, and optimize the method to work best for people using that size in particular. Note that the bigger the input image is, the bigger the amount of memory needed for inference, which is why we need to limit the amount of memory needed to minimum (for context: the article's network, without any optimizations, can infer A4 300dpi images using 5GB of memory, and nearly 20GB for A4 600 dpi).

4. Implementation



The fully convolutional layers is a good idea due to being very tolerant to all sorts of input sizes.

4.1. Short description of convolution

Mathematically, a convolution kernel is a matrix of size $k \times k \times n \times m$ (which you can see as m kernels of $k \times k \times n$ size), with input of size $a \times b \times n$ and output of size $a' \times b' \times m$. The kernel takes a $k \times k \times n$ portion of the input, multiplies item-by-item, and sums up everything. Since there are m of those 3D kernels, you get m output variables for every $k \times k \times n$ portion of the input. The usual value for k is either 3 or 5, and n and m depend on how deep in the network the layer is located (usually if it's in the middle, it has the biggest values of n and m). This is how you get one "pixel" of the output: a matrix of size $1 \times 1 \times m$.

How the $k \times k \times n$ portions are selected is defined by *stride* and *padding*. *Stride* defines how many "pixels" the kernel will skip when moving to the next portion of the input. The usual values for *stride* are 1 (which keeps the input and output sizes roughly the same), 2 (which skips every other "pixel" and makes the output size roughly half the size of the input), and 1/2 (which doubles the output size). *Padding* on the other hand is used to ensure that the output sizes (a, b and a', b') are exactly the same, twice as big or half the sizes of the input sizes by adding zeroed "pixels" around the actual input.

5. Challenges

5.1. Choosing the right network

Considering that we're proposing to be using a fully convolutional network, there is a hard limit on how many pixels it "sees" at once. That number is equal to $\sum_{i=0}^{n} k_i - 1$, where k_i is the kernel size of the i-th convolutional layer. Further I'll refer to that number as *frame*.

There are three main parameters of the network: the number of layers, the kernel sizes, and the third dimensions of the kernels.

Using a bigger network can ensure that learning the task is possible, and improve the results. On the other hand, especially if the *frame* is too big in comparison to the level of detail and the sketch brush thickness, the network might have a harder time to generalize. Moreover networks with more layers will take not only more time and more memory, but also more iterations to be trained[1].

Smaller networks might learn faster and easier, but might also not fit all the learning data.

5.1.1. The network described in the article

The obvious choice would be to choose the network described in the article we base our work on (Simo-Serra 2016).

That network has twenty three (23) convolutional layers with ReLU as an activation function (one 5x5, three 4x4, and nineteen 3x3), it uses a custom loss function with a loss map, Adadelta as the learning method, and batch normalization layers to speed up the learning process.

[1] The problem is called *Gradient vanishing*, see `https://en.wikipedia.org/wiki/Vanishing_gradient_problem`

The network is pretty big – in the standard state, using floating numbers, it requires over 5GB of memory to infer a single A4 300 dpi picture with just one channel (grey). It doesn't fit our restrictions. The network can be quantized, reducing the memory usage, however that introduces a risk: all quantization have a much higher chance to reduce accuracy than to improve it, and there are cases when standard quantization methods don't work well for a specific network. It's a fairly new topic of research as well.

Therefore, we might want to consider using a smaller network. Authors of the article mentioned neither experimentation nor thought process in regards to the size of the network, so we can't use that for estimation of the best size for our purpose.

5.1.2. Using a custom-designed network

Such network should still be based on the same principle, which is fully convolutional neural network, with ReLU and sigmoid activation layers and batch normalization.

What we could change is:

- number of layers,
- depths of kernels,
- loss map.

Different number of layers This leads to a smaller *frame*. That might be desirable or not.

The frame on the original network is, as far as I can tell, exactly 51 pixels. That might actually be too little, concluding from my talk with an artist, so I don't think we should lower the number of layers, if anything, maybe even increase it.

Different depths of kernels This doesn't change the frame, but does change the number of parameters, leading to a bigger or smaller capacity of the network and bigger or smaller memory usage. I think this should be the first thing we should reduce.

Different loss map That doesn't change the network size at all, but affects the learning process.

The loss map was selected to make the learning process not focus too much on the thick lines (on the target image), but strike a balance between thin and thick lines. The authors of the article say that without the loss map, the learning process took longer. Question remains whether it really is necessary in our case, but the difference in speed were significant enough that we might want to keep the loss map defined the same way.

5.2. Brush size/line thickness

There are several ways to add line/brush thickness to the network.

The standard practice would be to inject this kind of information as an input to a "fully-connected" dense layer, but since we won't have it in our network, we need to work around that in some way.

Here are my suggestions:

1. As another channel for the input image (which in our case will have just one channel, describing the value of the pixel)
    - Pro: whole network can learn it properly.
    - Con: it will be a huge amount of repeated data, since every pixel will have the exact same value in the second channel. And if we wanted to add more variables, then there will be more and more channels, with more and more repeated data.
    - Con: it doubles the amount of parameters through the whole network (doubling the memory, inference/learning time etc.).
2. Inject the data as another channel just before the last layer.
    - Pro: way less parameters.
    - Con: it might possibly not work as well.
3. Train the network on one line thickness, then retrain the last layer on different line thicknesses. Then either keep those last layers as separate setting for users to choose from, or, preferably, if possible, allow for all kinds of sizes by doing linear extrapolation from the parameters of the last layer of different line thickness.
    - Pro: way less parameters.
    - Con: the linear extrapolation is a bit complicated and might not really work as expected. I don't think it's a standard practice, so it would be a great risk to attempt that.

I believe we'll have the best chance of succeeding by using the method 2., injecting the info into the last layer.

Note that for training the network to understand the line thickness, we'd need images with particular brush with particular, known brush size. It might be a good idea to train on the same sketches with multiple target images of different brush sizes.

5.3. Brush preset/style

1. We could have a set of brushes, and a number indicating which brush the lineart should be in, the same way as in 5.2.
    - Pro: It's straight-forward, and might work.
    - Con: The index of the brush in a set won't be linearly related to the number in the channel, in the same way brush size is; so learning will be harder.
    - Con: it only allows for specific brushes.
2. After training a network for one brush, we could freeze all layers except for the last one and retrain it to for different styles of lineart.
    - Pro: More probable to succeed.
    - Krita will need to keep all of those networks or network layers saved separately.
    - Con: Still only a set of brushes/styles.

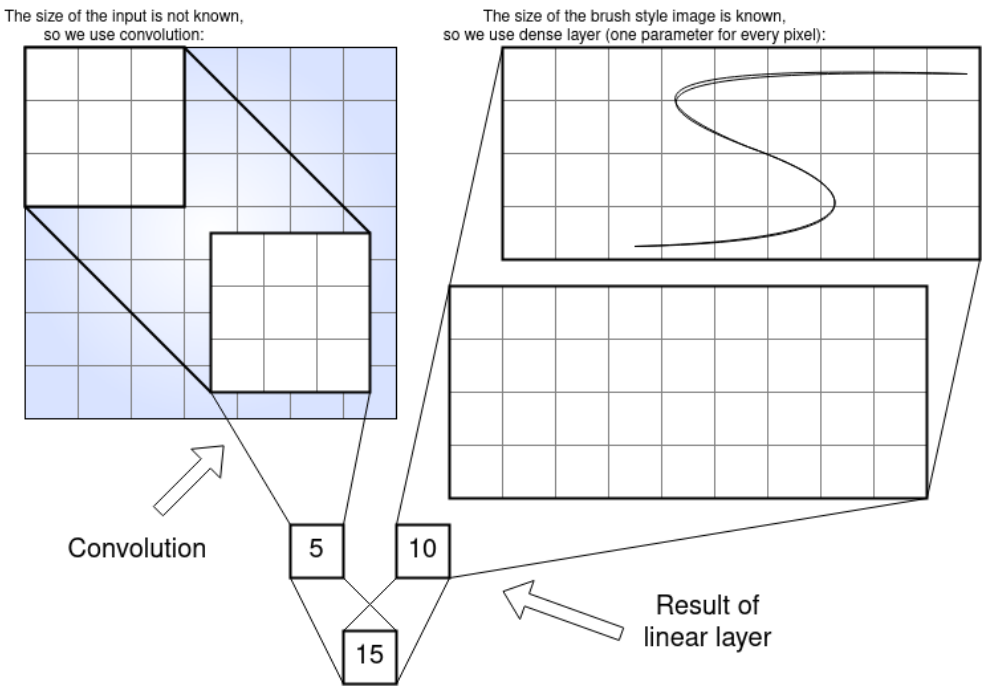
3. Ideally, we should inject the picture of the brush (going from 0% to 100% sensors, like in Krita's Brush Editor – it should be trivial to generate from a brush preset) in some point of the network. That would allow for usage of all kinds of brushes, even custom ones made by users, however it is not trivial to do.

   One idea I had is to change either the last or the one-before-last layer from fully convolutional to a custom one, which would contain both a kernel for convolutional part and a linear set of weights for the brush style image, and just sum them together. In PyTorch, this shouldn't be difficult to do (from what I gathered, just from using tensor operations, the library will know how to do the backward pass itself, and the learning process should all work automagically, as with typical layers). It might work, and I believe it's worth checking out. It is a novel idea for a network architecture, though.

   The diagram below shows this idea. As you can see, there are two inputs to the custom layer: the input image (result of previous convolutional layers) and the brush image. There are two sets of parameters: the kernel, just like in every convolutional layer, and a parameter for every pixel in the brush image (note: for this to work, the brush image needs to always be the same, known size, since that will be the size of the matrix of the parameters). The result of the convolution and the result of the linear layer operating on the brush image are summed up together and form the output.

   The diagram is simplified a bit: the input image, the convolutional kernel and the linear layer are shown to have no depth. This is to make the idea easier to read and understand. The convolution will work as usual, and the linear layer will have the depth depending on the depth of the output: if it will be placed in the last layer, the depth will be 1 (to output just the output greyscale image with just one channel), and if it will be placed in the one-before-last layer, the depth will match the depth of the output of that layer (also the output of the convolutional layer) (using the variables defined in 4.1, its depth will equal m, and the output of that linear layer will be $1 \times 1 \times m$, and that vector will be added to every pixel to form the final output of this custom layer).

   This idea needs to be thoroughly tested as a proof of concept before using it in the final project.



6. Technical details

Note that since models are often interchangeable (many neural networks libraries can load neural networks from other libraries), we can separate training from usage in Krita.

6.1. Training

I suggest making Python scripts using PyTorch library. It's a well-known, good library for machine learning and neural networks.

6.2. Inclusion in Krita

It seems like the best approach would be to include either just OpenVine, or ONNX with OpenVine included as a backend for ONNX. That will make sure that our users can use this tool on all kinds of hardware.

7. Estimated effort

The project consists of three parts:

1. Pre-production and experimentation
    - Creating Python scripts for the network and training.
    - Experimenting with different network sizes and shapes to find the one that best fits our use case.

- Experimenting with different solutions for the line thickness and brush styles.

2. Training the network.
3. Implementing a way to use the network in Krita.

Note that the part 2. and 3. can happen simultaneously. Developer's time is required in parts 1. and 3., while training time is required in parts 1. and 2.

Training time can be estimated based on the article, where the authors mention that training the network took 3 weeks of training on Titan X GPU. Today's best GPUs are significantly faster, and I suspect our network will be smaller, but we also need to account for experimentation and the addition of complexity due to line thickness and brush style.

I would estimate whole training time to take 6 weeks on Titan X, which means for example 2.5 weeks on Nvidia Quadro RTX A6000, or 3.4 weeks on Quadro RTX 5000, or 1.2 weeks on Nvidia RTX 4090, though of course that's all based on benchmarks (I used `https://gpu.userbenchmark.com/Compare/`, which is for general or gaming purposes, not neural networks, but it's a good resource for consistent results). Note that we should know the estimated final training time after the experimentation stage, knowing both the size of the network and the GPU, and after already training the network for several epochs on that GPUs.

Dmitry estimates Part 3. to take two months (one for OpenVine and one for ONNX Runtime, assuming we'll use them both). Part 1. should take around another month or two.

To sum that up:

- Training time (time required from GPU): probably around 3-4 weeks, depending on the GPU
- Development time: 4 months.
- Total project time: 6-10 months (the higher estimation is to account for unknown unknowns).

I do want to emphasize that this is still a very tentative, preliminary estimation.

Note that training time on modern fast GPUs is reasonably "cheap", with 1-2k euros per month, so the upper bound of training cost should be around 3k, but more likely it would amount to maybe 1.5k.