CS60050 Machine Learning
Autumn 2016-17

Term Project Report

ArtificiallyArtistic

Project title

Artistic Rendering of Images

Group members

(sorted by roll number)

14CS10004 Aniket Suri
14CS10013 Pradeep Dogga
14CS10057 Projjal Chanda
14CS10060 Ashrujit Ghoshal
14CS10061 Sayan Ghosh
14CS10062 Sourav Pal
14CS30019 Mousam Roy
14CS30032 Sayan Mandal
14CS30043 Arundhati Banerjee
14CS30044 Sohan Patro

Problem definition

Separation of style and content of images and subsequent application of a set of styles to an input image, implemented first using bilinear models and then with Convolutional Neural Networks and extension of the CNN model to doodles and videos.

Methodology

Bilinear Model Approach

Perceptual systems routinely separate the “content” and “style” factors of their observations, classifying familiar words spoken in an unfamiliar accent, identifying a font or handwriting style across letters, or recognizing a familiar face or object seen under unfamiliar viewing conditions. These and many other basic perceptual tasks have in common the need to process separately two independent factors that underlie a set of observations. This article shows how perceptual systems may learn to solve these crucial two-factor tasks using simple and tractable bilinear models. By fitting such models to a training set of observations, the influences of style and content factors can be efficiently separated in a flexible representation that naturally supports generalization to unfamiliar styles or content classes.

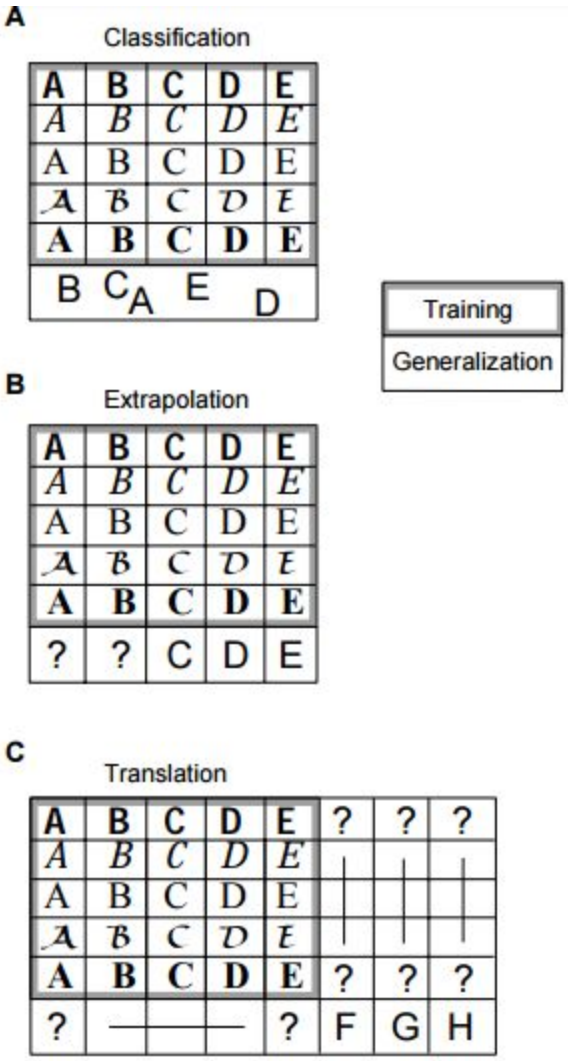


Figure 1: Given a labeled training set of observations in multiple styles (e.g., fonts) and content classes (e.g., letters), we want to (A) classify content observed in a new style, (B) extrapolate a new style to unobserved content classes, and (C) translate from new content observed only in new styles into known styles or content classes

Figure 1 illustrates three abstract tasks that fall under this framework: classification, extrapolation, and translation. Examples of these abstract tasks in the domain of typography include classifying known characters in a novel font, extrapolating the missing characters of an incomplete novel font, or translating novel characters from a novel font into a familiar font.

Bilinear models are two-factor models with the mathematical property of separability: their outputs are linear in either factor when the other is held constant. Their combination of representational expressiveness and efficient learning procedures enables bilinear models to

overcome two principal drawbacks of existing factor models that might be applied to learning the tasks in Figure 1.
Here we will use the terms style and content generically to refer to any two independent factors underlying a set of perceptual observations.

Bilinear Models

We have explored two bilinear models, closely related to each other, which we distinguish by the labels symmetric and asymmetric. This section describes the two models and illustrates them on a simple data set of face images.

Symmetric Model

In the symmetric model, we represent both style **s** and content **c** with vectors of parameters, denoted $\mathbf{a}^s$ and $\mathbf{b}^c$ and with dimensionalities **I** and **J**, respectively. Let $\mathbf{y}^{sc}$ denote a K-dimensional observation vector in style **s** and content class **c**. We assume that $\mathbf{y}^{sc}$ is a bilinear function of as and bc given most generally by the form

$$y_k^{sc} = \sum_{i=1}^{I} \sum_{j=1}^{J} w_{ijk} a_i^s b_j^c.$$

Here i, j, and k denote the components of style, content, and observation vectors, respectively.1 The $\mathbf{w}_{ijk}$ terms are independent of style and content and characterize the interaction of these two factors. Their meaning becomes clearer when we rewrite equation in vector form. Letting $\mathbf{W}_k$ denote the **I × J** matrix with entries $\{\mathbf{w}_{ijk}\}$, equation can be written as

$$y_k^{sc} = \mathbf{a}^{s^T} \mathbf{W}_k \mathbf{b}^c. \qquad (2.2)$$

In equation 2.2, the K matrices $\mathbf{W}_k$ describe a bilinear map from the style and content vector spaces to the K-dimensional observation space. The interaction terms have another interpretation, which can be seen by writing the symmetric model in a different vector form. Letting $\mathbf{w}_{ij}$ denote the K-dimensional vector with components $\{\mathbf{w}_{ijk}\}$, equation can be written as

$$\mathbf{y}^{sc} = \sum_{i,j} \mathbf{w}_{ij} a_i^s b_j^c. \qquad (2.3)$$

In equation 2.3, the $\mathbf{w}_{ijk}$ terms represent **I × J** basis vectors of dimension **K**, and the observation $\mathbf{y}^{sc}$ is generated by mixing these basis vectors with coefficients given by the tensor product of $\mathbf{a}^s$ and $\mathbf{b}^c$.

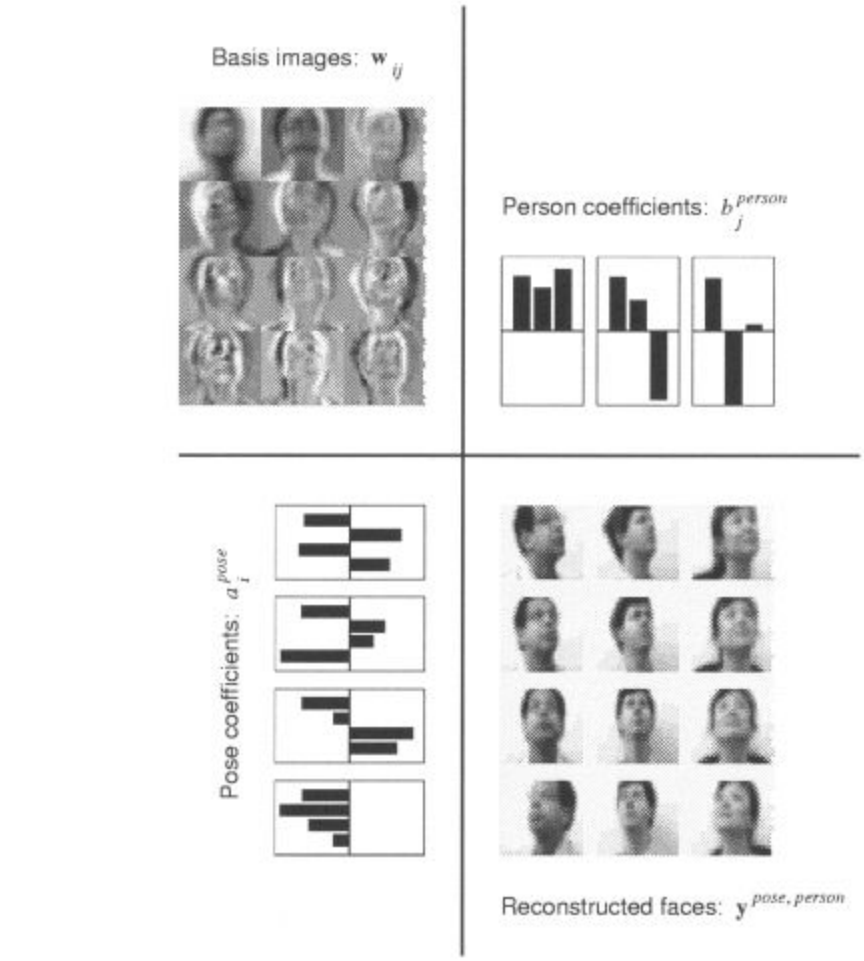


Figure 2: Illustration of a symmetric bilinear model for a small set of faces . The two factors for this example are person and pose. One vector of coefficients, **(ai)pose**, describes the pose, and a second vector, **(bj)person** , describes the person. To render a particular person under a particular pose, the vectors **(ai)pose** and **(bj)person** multiply along the four rows and three columns of the array of basis images **wij**. The weighted sum of basis images yields the reconstructed faces **(y)pose,person**.

Asymmetric Model

Sometimes linear combinations of a few basis styles learned during training may not describe new styles well. We can obtain more flexible, asymmetric models by letting the interaction terms **wijk** themselves vary with style.Then the first equation becomes

$$y_k^{sc} = \sum_{i,j} w_{ijk}^s a_i^s b_j^c.$$

Without loss of generality, we can combine the style-specific terms of first equation into

$$a_{jk}^s = \sum_i w_{ijk}^s a_i^s, \qquad (2.4)$$

Giving

$$y_k^{sc} = \sum_j a_{jk}^s b_j^c. \qquad (2.5)$$

Again, there are two interpretations of the model, corresponding to different vector forms of equation 2.5. First, letting As denote the **K × J** matrix with entries **{(as)jk}**, equation 2.5 can be written as

$$\mathbf{y}^{sc} = \mathbf{A}^s \mathbf{b}^c. \qquad (2.6)$$

Here, we can think of the **(as)jk** terms as describing a style-specific linear map from content space to observation space. Alternatively, letting as **j** denote the K-dimensional vector with components **{(as)jk}**, equation 2.5 can be written as

$$\mathbf{y}^{sc} = \sum_j \mathbf{a}_j^s b_j^c. \qquad (2.7)$$

Now we can think of the **(as)jk** terms as describing a set of **J** style-specific basis vectors that are mixed according to content-specific coefficients **(bc)j** (independent of style) to produce the observations

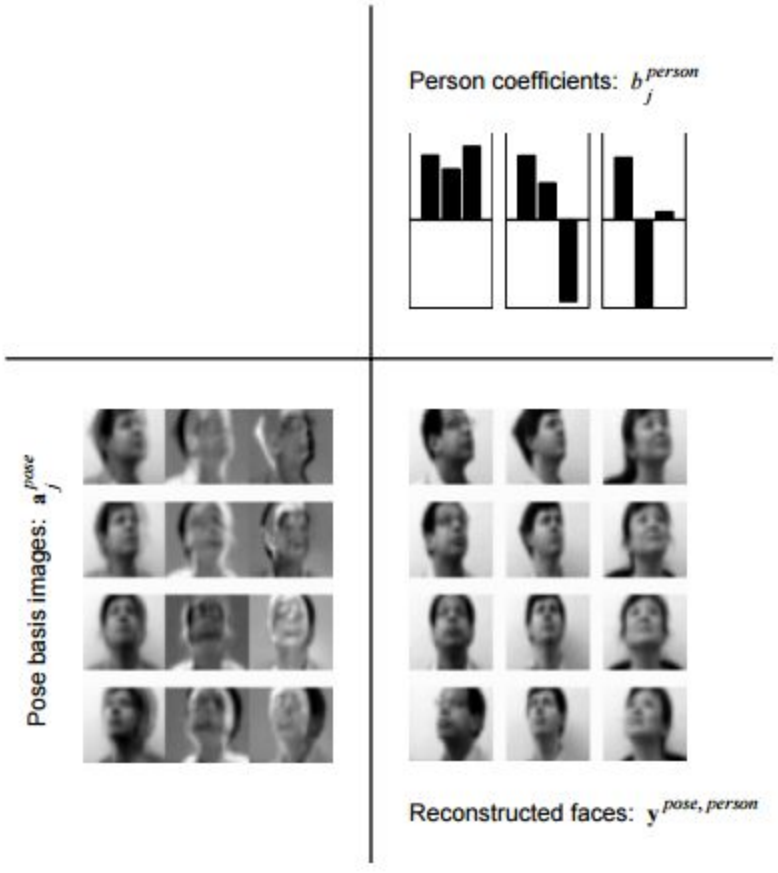


Figure 3: The images of Figure 2 represented by an asymmetric bilinear model with head pose as the style factor. Person-specific coefficients b_j^{person} multiply pose-specific basis images $\mathbf{a}_j^{pose}$; the sum reconstructs a given person in a given pose. The basis images are similar to an eigenface representation within a given pose (Moghaddam & Pentland, 1997), except that in this model the different basis images are constrained to allow one set of person coefficients to reconstruct the same face across different poses.

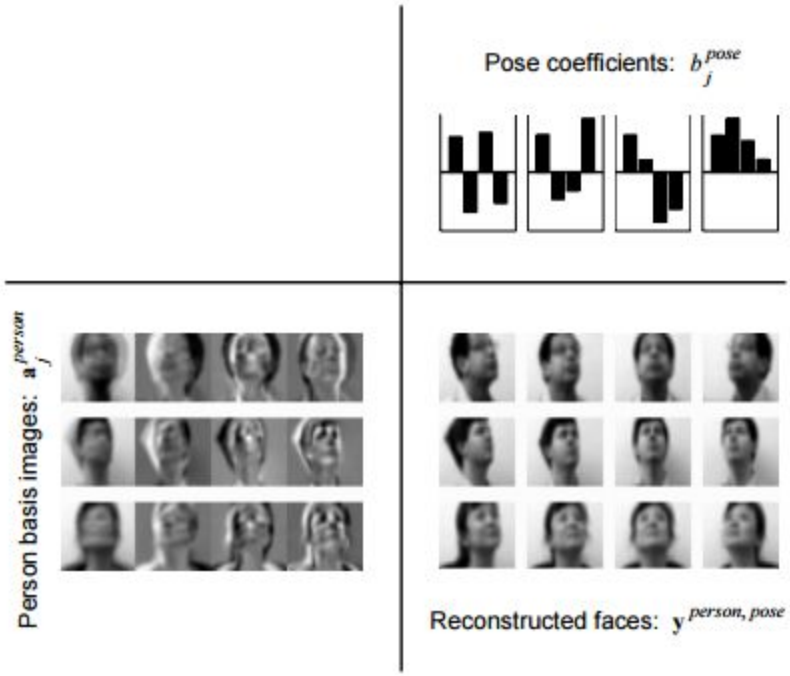
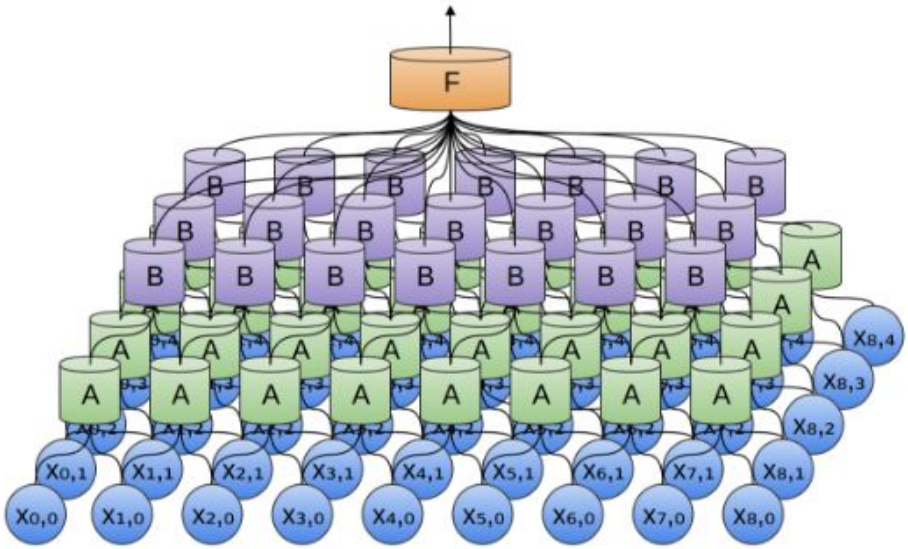


Figure 4: Asymmetric bilinear model applied to the data of Figure 2, treating identity as the style factor. Now pose-specific coefficients b_j^{pose} weight person-specific basis images $\mathbf{a}_j^{person}$ to reconstruct the face data. Across different faces, corresponding basis images play corresponding roles in rotating head position.

Convolutional Neural Network

In the last few years, deep neural networks have lead to breakthrough results on a variety of pattern recognition problems, such as computer vision and voice recognition. One of the essential components leading to these results has been a special kind of neural network called a *convolutional neural network*.

At its most basic, convolutional neural networks can be thought of as a kind of neural network that uses many identical copies of the same neuron.1 This allows the network to have lots of neurons and express computationally large models while keeping the number of actual parameters – the values describing how neurons behave – that need to be learned fairly small.

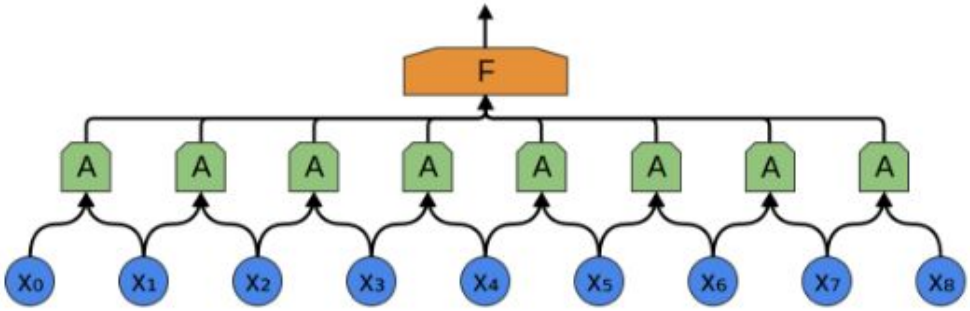


A 2D Convolutional Neural Network

This trick of having multiple copies of the same neuron is roughly analogous to the abstraction of functions in mathematics and computer science. When programming, we write a function once and use it in many places – not writing the same code a hundred times in different places makes it faster to program, and results in fewer bugs. Similarly, a convolutional neural network can learn a neuron once and use it in many places, making it easier to learn the model and reducing error.

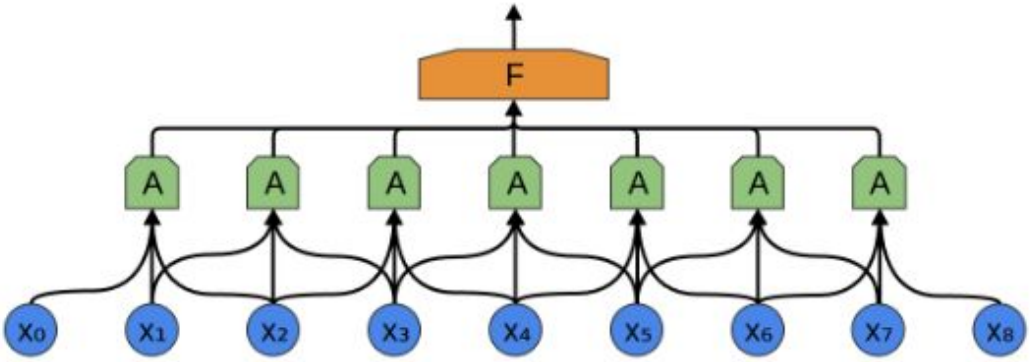
Structure Of Convolutional Neural Netwroks

So, we create a group of neurons, A, that look at small time segments of our data. A looks at all such segments, computing certain *features*. Then, the output of this *convolutional layer* is fed into a fully-connected layer, F.



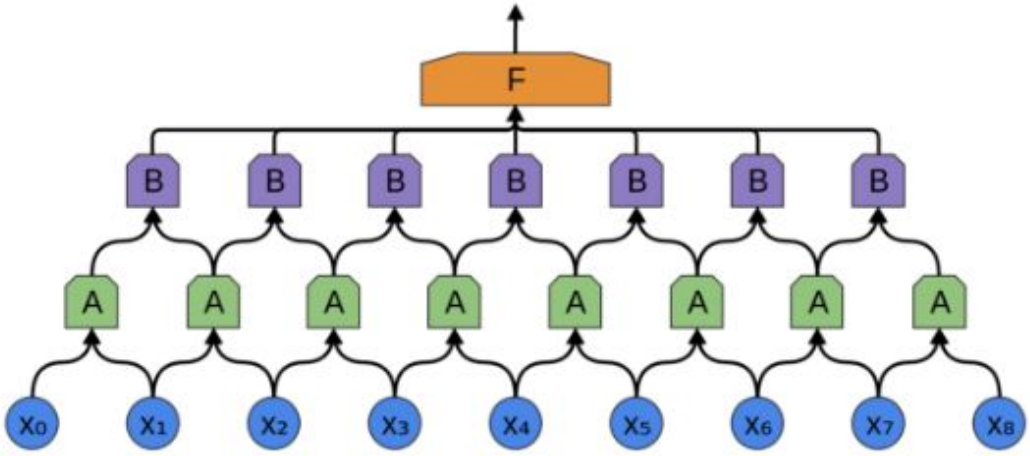
In the above example, A only looked at segments consisting of two points. This isn't realistic. Usually, a convolution layer's window would be much larger.

In the following example, A looks at 3 points. That isn't realistic either – sadly, it's tricky to visualize A connecting to lots of points.



One very nice property of convolutional layers is that they're composable. You can feed the output of one convolutional layer into another. With each layer, the network can detect higher-level, more abstract features.

In the following example, we have a new group of neurons, B. B is used to create another convolutional layer stacked on top of the previous one.

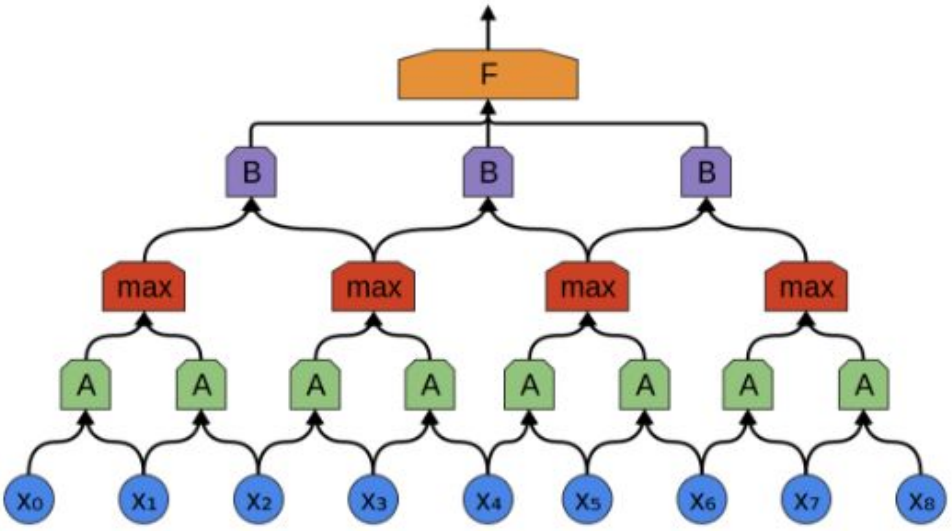


Convolutional layers are often interweaved with pooling layers. In particular, there is a kind of layer called a max-pooling layer that is extremely popular.

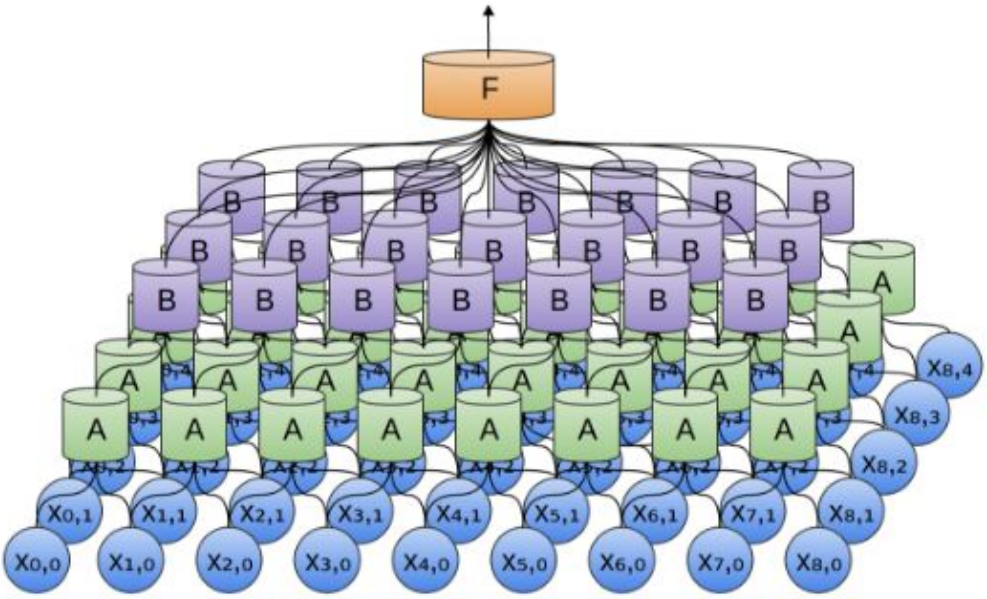
Often, from a high level perspective, we don't care about the precise point in time a feature is present. If a shift in frequency occurs slightly earlier or later, does it matter?

A max-pooling layer takes the maximum of features over small blocks of a previous layer. The output tells us if a feature was present in a region of the previous layer, but not precisely where.

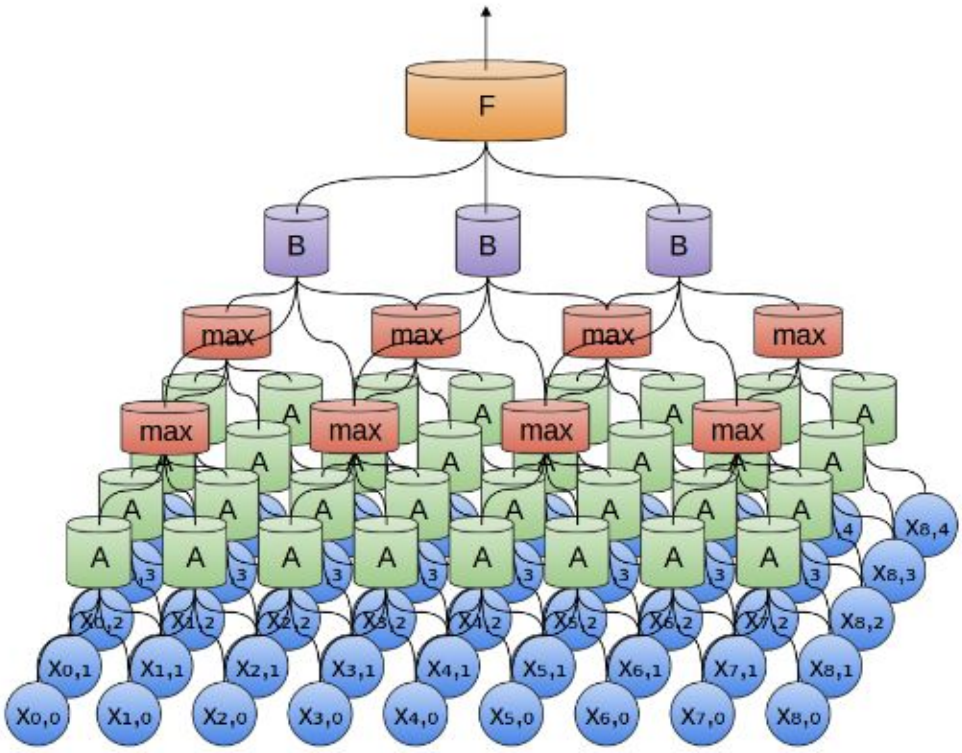
Max-pooling layers kind of "zoom out". They allow later convolutional layers to work on larger sections of the data, because a small patch after the pooling layer corresponds to a much larger patch before it. They also make us invariant to some very small transformations of the data.



In our previous examples, we've used 1-dimensional convolutional layers. However, convolutional layers can work on higher-dimensional data as well. In fact, the most famous successes of convolutional neural networks are applying 2D convolutional neural networks to recognizing images.

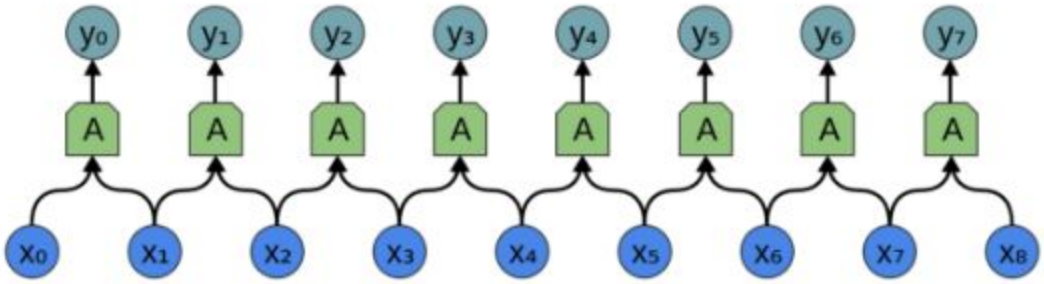


With Max pooling layer in between,



Formalizing Convolutional Neural Network

Consider a 1-dimensional convolutional layer with inputs {xn} and outputs {yn}:



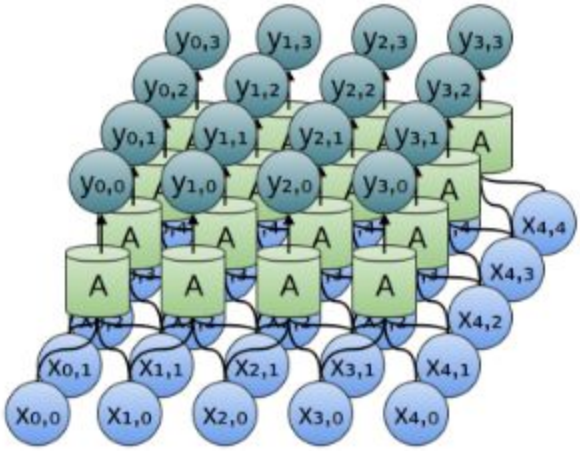
It's relatively easy to describe the outputs in terms of the inputs:

$$y_n = A(x_n, x_{n+1}, \dots)$$

$$y_0 = A(x_0, x_1)$$

$$y_1 = A(x_1, x_2)$$

Similarly, if we consider a 2-dimensional convolutional layer, with inputs {xn,m} and outputs {yn,m}:



We can, again, write down the outputs in terms of the inputs:

$$y_{n,m} = A\begin{pmatrix} x_{n,\,m}, & x_{n+1,\,m}, & \dots, \\ x_{n,\,m+1}, & x_{n+1,\,m+1}, & \dots, \\ & \dots & \end{pmatrix}$$

For example:

$$y_{0,0} = A\begin{pmatrix} x_{0,0}, & x_{1,0}, \\ x_{0,1}, & x_{1,1} \end{pmatrix}$$

$$y_{1,0} = A\begin{pmatrix} x_{1,0}, & x_{2,0}, \\ x_{1,1}, & x_{2,1} \end{pmatrix}$$

If one combines this with the equation for $A(x)$,

$$A(x) = \sigma(Wx + b)$$

Semantic Style Transfer and Turning Two-Bit Doodles Into Artwork

L-BFGS algorithm

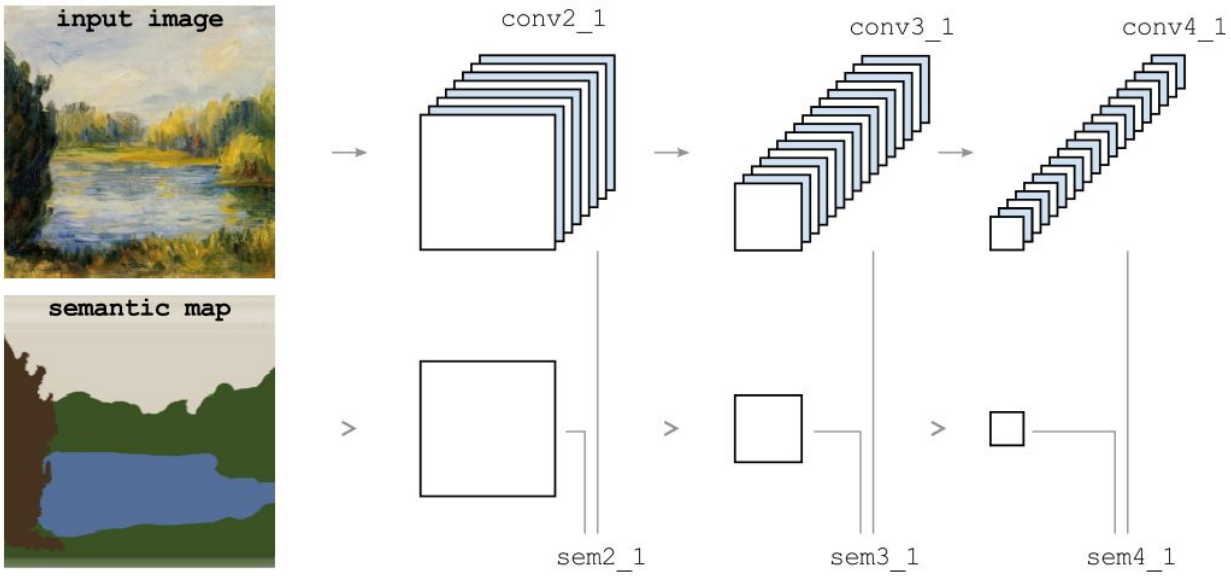


Figure 3: Our augmented CNN that uses regular filters of N channels (top), concatenated with a semantic map of M=1 channel (bottom) either output from another network capable of labeling pixels or as manual annotations.

Model:

Our contribution builds on a patch-based approach (Li and Wand 2016) to style transfer, using optimization to minimize content reconstruction error E_c (weighted by α) and style remapping error E_s (weight β).

See (Gatys, Ecker, and Bethge 2015) for details about E c .

$$E = \alpha E_c + \beta E_s \quad (1)$$

Architecture

The most commonly used CNNs for image synthesis is VGG (Simonyan and Zisserman 2014), which combines pooling and convolution layers l with 3 × 3 filters (e.g. the first layer after third pool is named conv4_1). Intermediate post-activation results are labeled x^l and consist of N channels, which capture patterns from the images for each region of the image: grain, colors, texture, strokes, etc. Other architectures tend to skip pixels regularly, compress data, or optimized for classification—resulting in low-quality synthesis (Nikulin and Novak 2016).

Our augmented network concatenates additional semantic channels m^l of size M at the same resolution, computed by down-sampling a static semantic map specified as input. The result is a new output with N +M channels, denoted s^l and labeled accordingly for each layer (e.g. sem4_1).

Before concatenation, the semantic channels are weighted by parameter γ to provide an additional user control point:

$$s^l = x^l \,||\, \gamma m^l \quad (2)$$

For style images, the activations for the input image and its semantic map are concatenated together as s^l_s. For the output image, the current activations x^l and the input content's semantic map are concatenated as s^l . Note that the semantic part of this vector is, therefore, static during the optimization process (implemented using L-BFGS).

This architecture allows specifying manually authored semantic maps, which proves to be a very convenient tool for user control—addressing the unpredictability of current generative algorithms. It also lets us transparently integrate recent pixel labeling CNNs (Thoma 2016), and leverage any advances in this field to apply them to image synthesis.

Algorithm:

Patches of k × k are extracted from the semantic layers and denoted by the function **Ψ,** respectively $\mathbf{\Psi(s^l_s)}$ for the input style patches and $\mathbf{\Psi(s^l)}$ for the current image patches. For any patch i in the current image and layer l, its nearest neighbor NN(i) is computed using normalized cross correlation—taking into account weighted semantic map:

$$\mathrm{NN}(i) := \arg\min_j \frac{\Psi_i(\mathbf{s}) \cdot \Psi_j(\mathbf{s}_s)}{|\Psi_i(\mathbf{s})| \cdot |\Psi_j(\mathbf{s}_s)|} \quad (3)$$

The style error $\mathbf{E_s}$ between all the patches i of layer l in the current image to the closest style patch is defined as the sum of the Euclidean distances:

$$E_s(\mathbf{s}, \mathbf{s}_s) = \sum_i ||\Psi_i(\mathbf{s}) - \Psi_{\mathrm{NN}(i)}(\mathbf{s}_s)||^2 \quad (4)$$

Note that the information from the semantic map in ml is used to compute the best matching patches and contributes to the loss value, but is not part of the derivative of the loss relative to the current pixels; only the differences in activation $\mathbf{x}^l$ compared to the style patches cause an adjustment of the image itself via the .

By using an augmented CNN that's compatible with the original, existing patch-based implementations can use the additional semantic information without changes. If the semantic

map and m^l is zero, the original algorithm (Li and Wand 2016) is intact. In fact, the introduction of the γ parameter from Equation 2 provides a convenient way to introduce semantic style transfer incrementally.

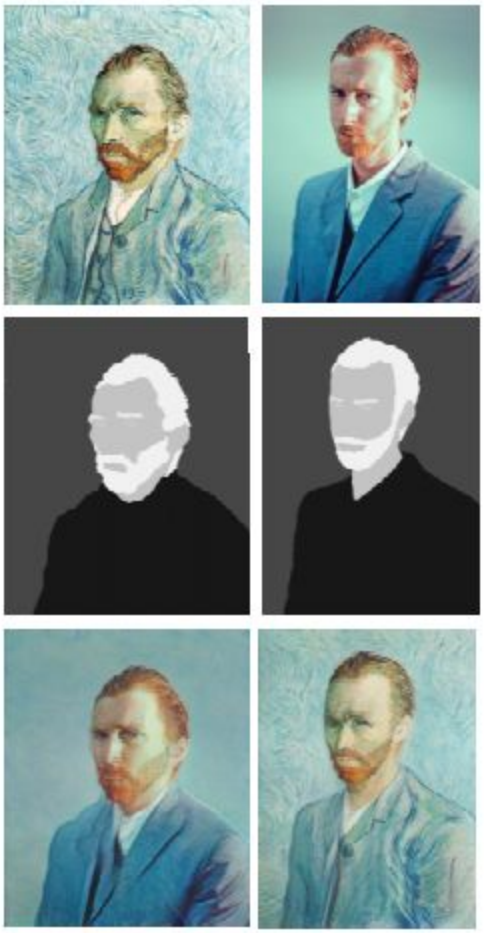


Figure 4: Examples of semantic style transfer with Van Gogh painting. Annotations for nose and mouth are not required as the images are similar, however carefully annotating the eyeballs helps when generating photo-quality portraits. [Photo by Seth Johnson, concept by Kyle McDonald.]

Artistic Style Transfer For Videos

We use the following notation: $\mathbf{p}^{(i)}$ is the i th frame of the original video, a is the style image and $\mathbf{x}^{(i)}$ are the stylized frames to be generated. Furthermore, we denote by $\mathbf{x'}^{(i)}$ the initialization of the style optimization algorithm at frame i. By $\mathbf{x}^j$ we denote the j^{th} component of a vector x.

Multi-pass Algorithm:

We found that the output image tends to have less contrast and is less diverse near image boundaries than in other areas of the image. For mostly static videos this effect is hardly visible. However, in cases of strong camera motion the areas from image boundaries move

towards other parts of the image, which leads to a lower image quality over time when combined with our temporal constraint. Therefore, we developed a multi-pass algorithm which processes the whole sequence in multiple passes and alternates between the forward and backward direction. Every pass consists of a relatively low number of iterations without full convergence. At the beginning, we process every frame independently. After that, we blend frames with non-disoccluded parts of previous frames warped according to the optical flow, then run the optimization algorithm for some iterations initialized with this blend. We repeat this blending and optimization to convergence.

Let $\mathbf{x'}^{(i)(j)}$ be the initialization of frame i in pass j and $\mathbf{x}^{(i)(j)}$ the corresponding output after some iterations of the optimization algorithm. When processed in forward direction, the initialization of frame i is created as follows:

$$\boldsymbol{x}'^{(i)(j)} = \begin{cases} \boldsymbol{x}^{(i)(j-1)} & \text{if } i = 1, \\ \delta \boldsymbol{c}^{(i-1,i)} \circ \omega_{i-1}^{i}\left(\boldsymbol{x}^{(i-1)(j)}\right) + \left(\overline{\delta}\mathbf{1} + \delta \overline{\boldsymbol{c}}^{(i-1,i)}\right) \circ \boldsymbol{x}^{(i)(j-1)} & \text{else.} \end{cases} \tag{11}$$

Here ◦ denotes element-wise vector multiplication, δ and δ'= 1−δ are the blend factors, 1 is a vector of all ones, and c' = 1 − c. Analogously, the initialization for a backward direction pass is:

$$\boldsymbol{x}'^{(i)(j)} = \begin{cases} \boldsymbol{x}^{(i)(j-1)} & \text{if } i = N_{\text{frames}} \\ \delta \boldsymbol{c}^{(i+1,i)} \circ \omega_{i+1}^{i}\left(\boldsymbol{x}^{(i+1)(j)}\right) + \left(\overline{\delta}\mathbf{1} + \delta \overline{\boldsymbol{c}}^{(i+1,i)}\right) \circ \boldsymbol{x}^{(i)(j-1)} & \text{else} \end{cases} \tag{12}$$

The multi-pass algorithm can be combined with the temporal consistency loss described above. We achieved good results when we disabled the temporal consistency loss in several initial passes and enabled it in later passes only after the images had stabilized.

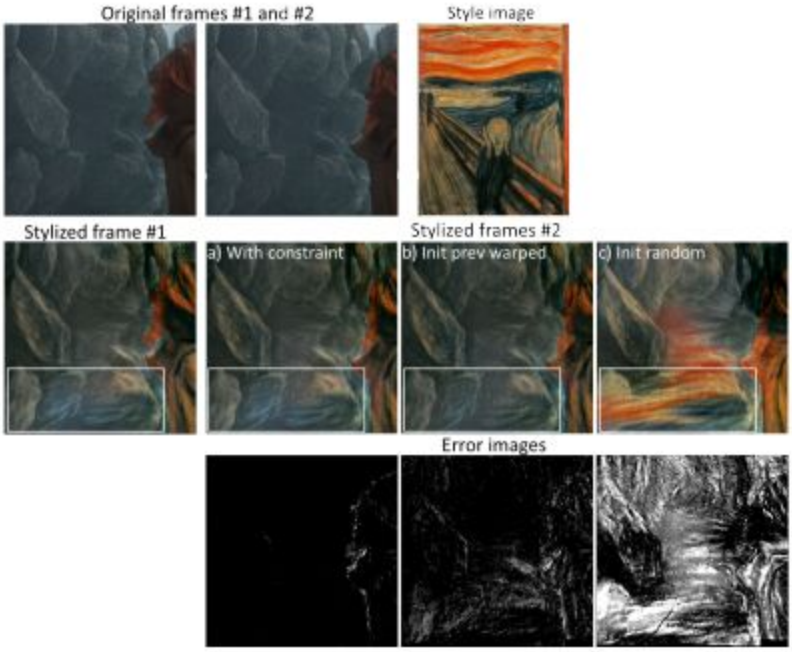


Fig. 2. Close-up of a scene from Sintel, combined with *The Scream* painting. **a)** With temporal constraint **b)** Initialized with previous image warped, but without the constraint **c)** Initialized randomly. The marked regions show most visible differences. *Error images* show the contrast-enhanced absolute difference between frame #1 and frame #2 warped back using ground truth optical flow, as used in our evaluation. The effect of the temporal constraint is very clear in the error images and in the corresponding video.

Obstacles Faced:

For Neural Doodle implementation, we often got errors on the server, as there was insufficient memory space available .
For implementation of artistic rendering of videos, we faced a lot of problems. First we were unable to install torch on the server. We tried running it on our machines. We finally could run it in one of our machines but a single test took an extremely long amount of time to run and we could not try out more test cases.

Technologies and libraries

For rendering of 2D images and doodles, we used Python as the main programming language. The coding of the video rendering part was done in Lua and C++. A bit of Bash scripting was involved as well.

Google's TensorFlow was chosen as the AI library for the purpose for the project since it is open-source and widely popular. Also, using TensorFlow in Python is extremely simple; many complex operations are often reduced to one or two lines of code. This allowed us to dedicate our time and concentration on actual techniques involved rather than worrying about the implementation details. We also extensively used the numpy and scipy libraries of Python.

Scope of future work

Although we had initially planned upon extending the style transfer model to 3D images towards the end of the project, we could not accomplish the same due to constraints of time.

There is good scope to apply the same techniques to stylize 3D images and to understand what additional things we need to take care of while handling 3D images, as compared to normal images.

Results

1. **For Artistic rendering of 2D images**

Style Used**:**

Below are some of the inputs and corresponding outputs of the from the trained model.

#1

Input:

Output:

#2

Input :

Output:

#3

Input :

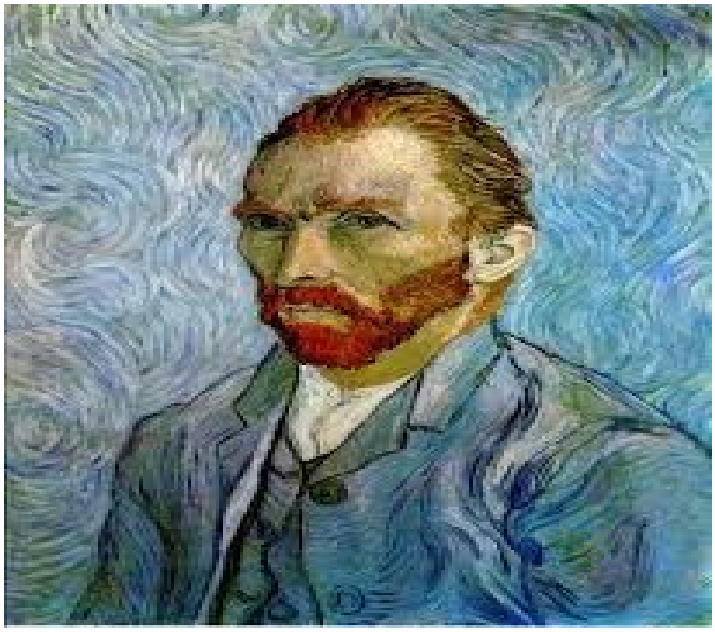
Output:

#4

Input:

Output:

Style Used:



Below are some of the inputs and corresponding outputs of the from the trained model.

#1

Input:

Output:

#2

Input:

Output:

#3

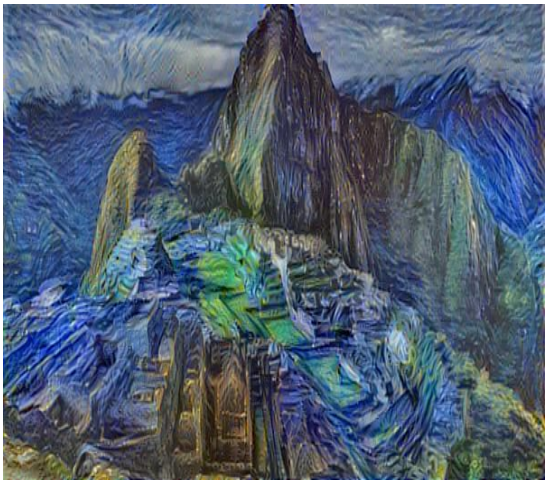
Input:

Output:

#4

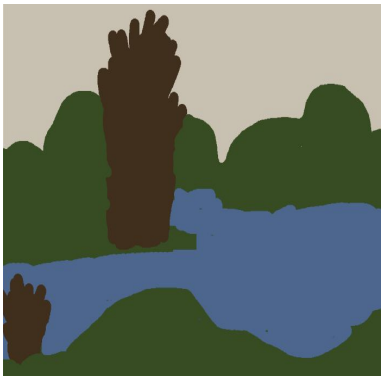
Input:

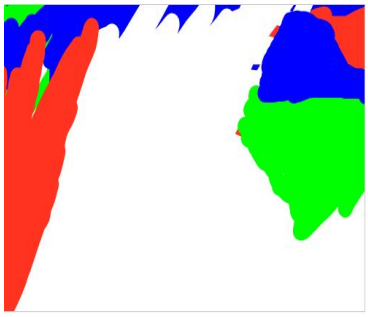
Output:



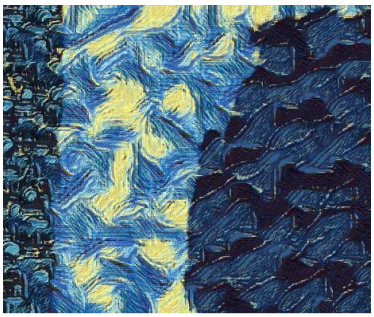
2. Neural Doodles

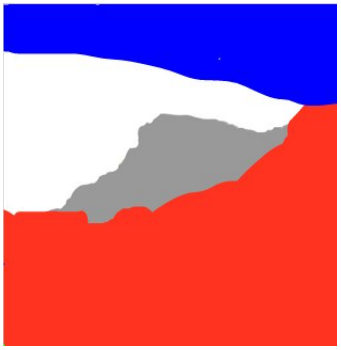
Style | Doodle | Output










3. Videos

Style Image:

This is a link to the input as well as output video.