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.

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.

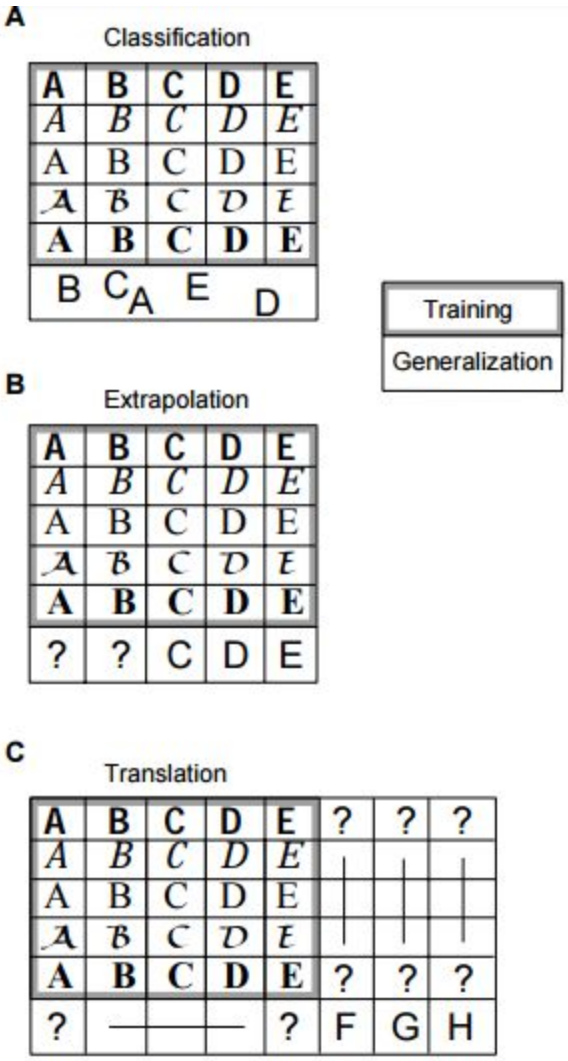


Figure 1: Given a labeled training set of observations in multiple styles (e.g., fonts) and content classes (e.g., letters), we want to (A) classify content observed in a new style, (B) extrapolate a new style to unobserved content classes, and (C) translate from new content observed only in new styles 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 $\mathbf{b}^c$.

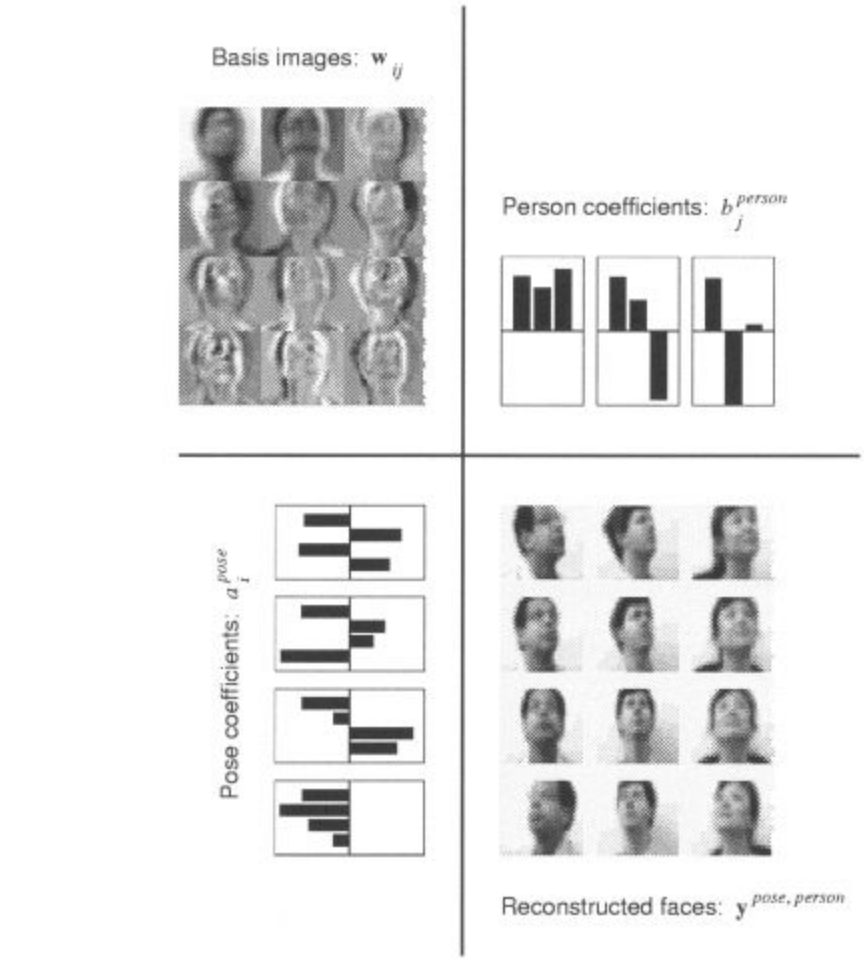


Figure 2: Illustration of a symmetric bilinear model for a small set of faces . The two factors for this example are person and pose. One vector of coefficients, **(ai)pose**, describes the pose, and a second vector, **(bj)person** , describes the person. To render a particular person under a particular 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

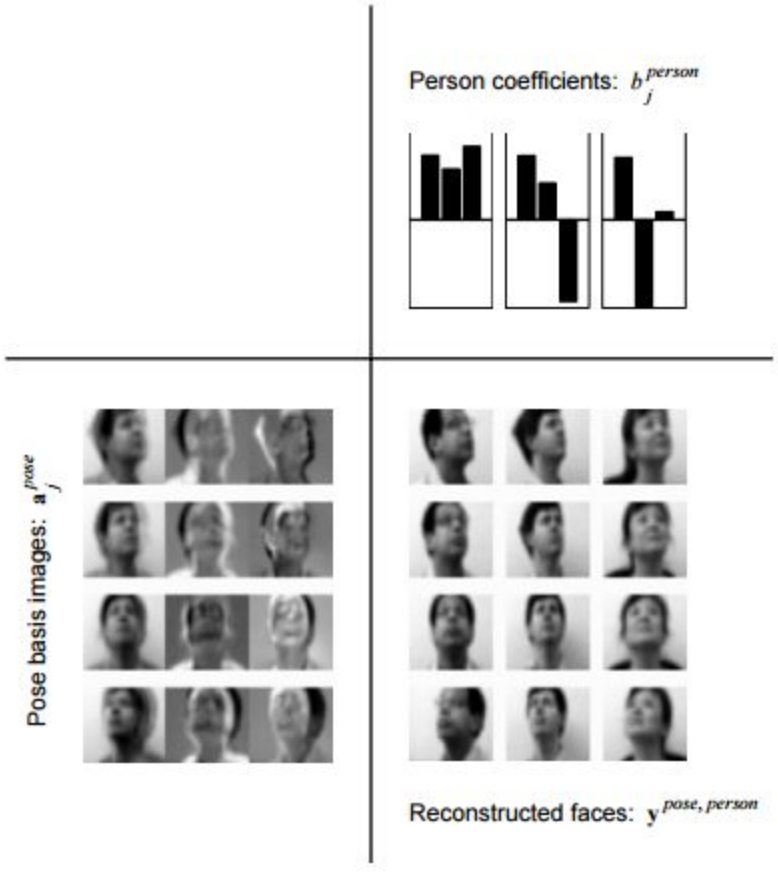


Figure 3: The images of Figure 2 represented by an asymmetric bilinear model with head pose as the style factor. Person-specific coefficients b_j^{person} multiply pose-specific basis images $\mathbf{a}_j^{pose}$; the sum reconstructs a given person in a given pose. The basis images are similar to an 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.

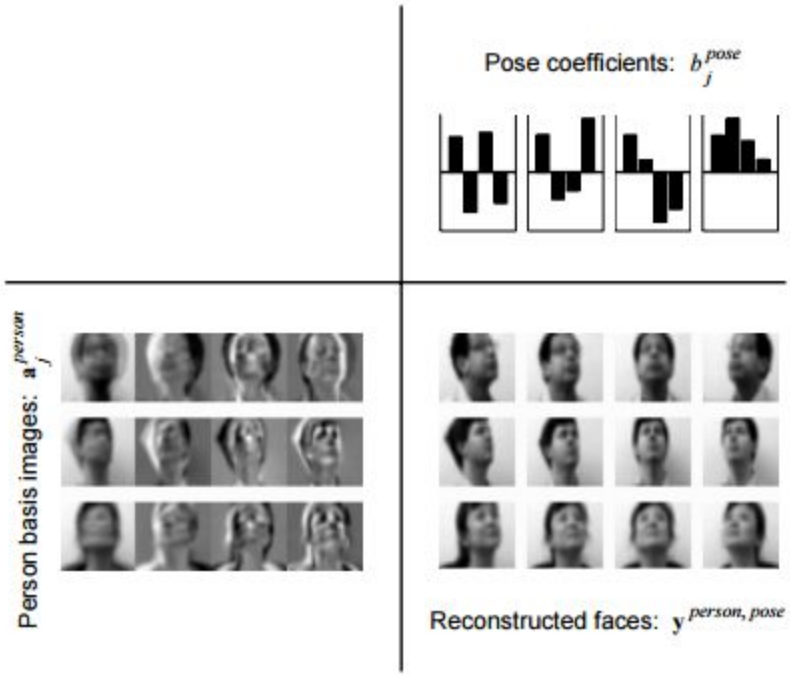
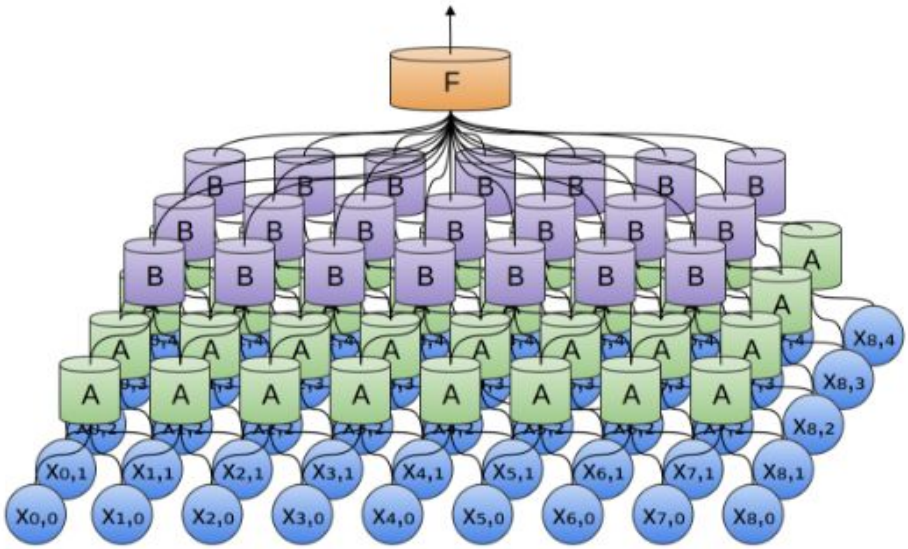


Figure 4: Asymmetric bilinear model applied to the data of Figure 2, treating identity as the style factor. Now pose-specific coefficients b_j^{pose} weight person-specific basis images $\mathbf{a}_j^{person}$ to reconstruct the face data. Across different faces, corresponding basis images play corresponding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

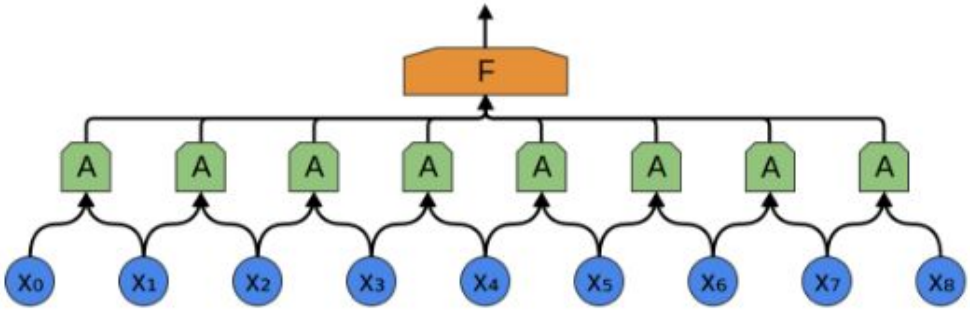


A 2D Convolutional Neural Network

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.

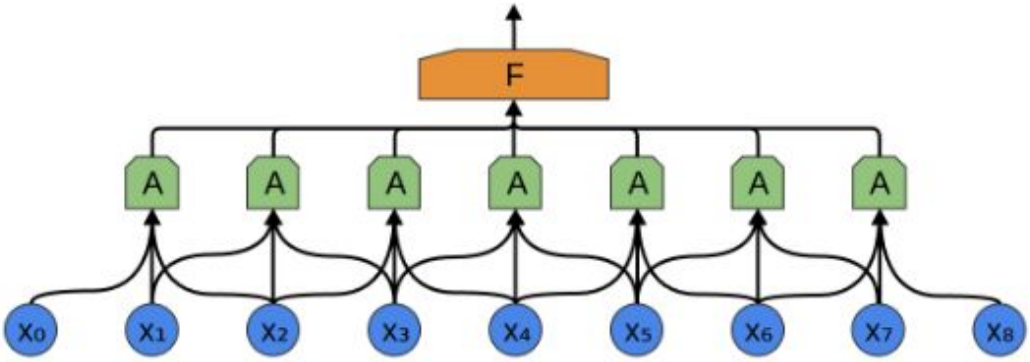
Structure Of Convolutional Neural Netwroks

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.



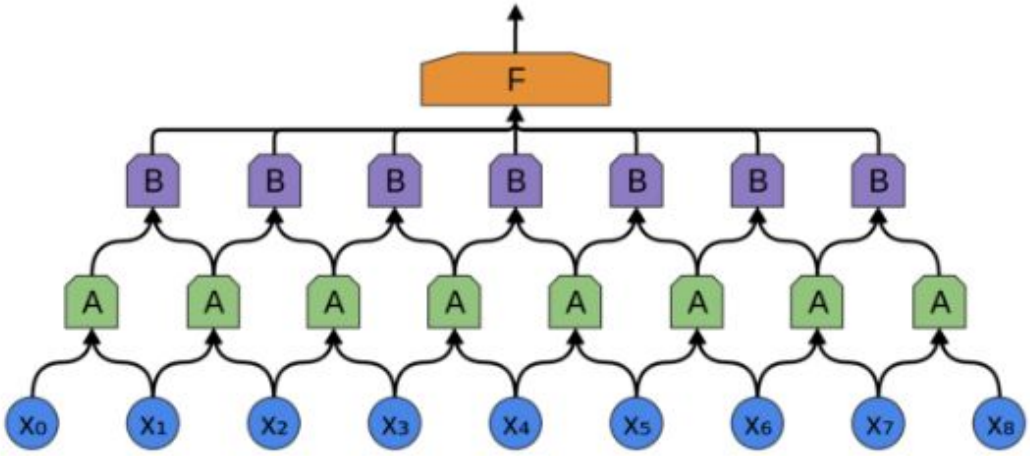
In the above example, A only looked at segments consisting of two points. This isn't realistic. Usually, a convolution layer's window would be much larger.

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.

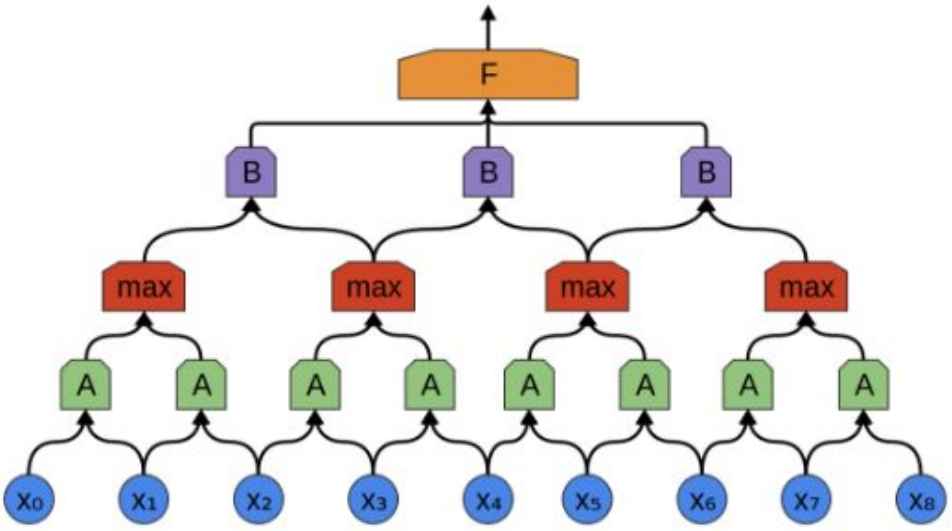


Convolutional layers are often interweaved with pooling layers. In particular, there is a kind of layer called a max-pooling layer that is extremely popular.

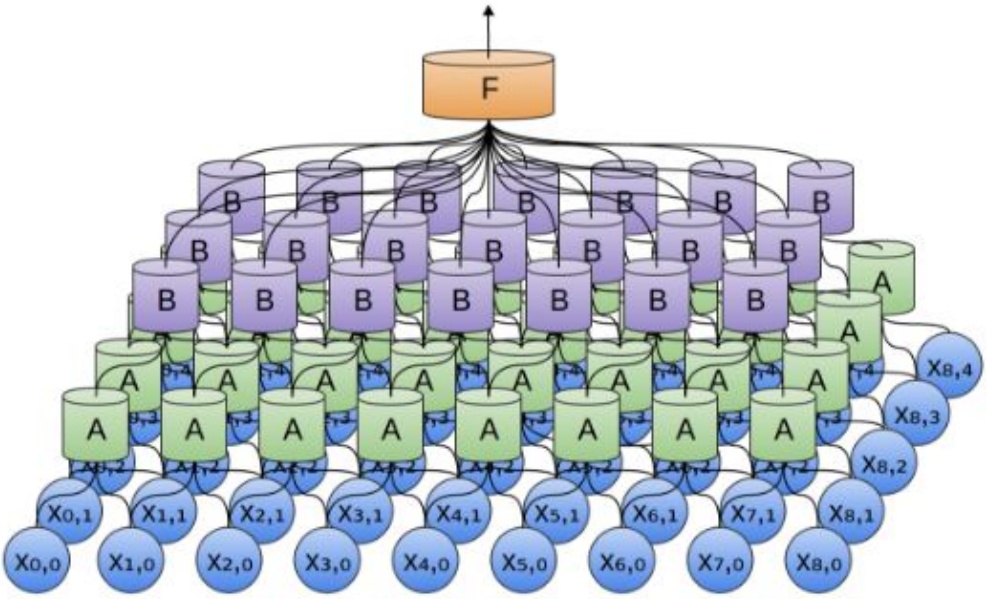
Often, from a high level perspective, we don't care about the precise point in time a feature is present. If a shift in frequency occurs slightly earlier or later, does it matter?

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.

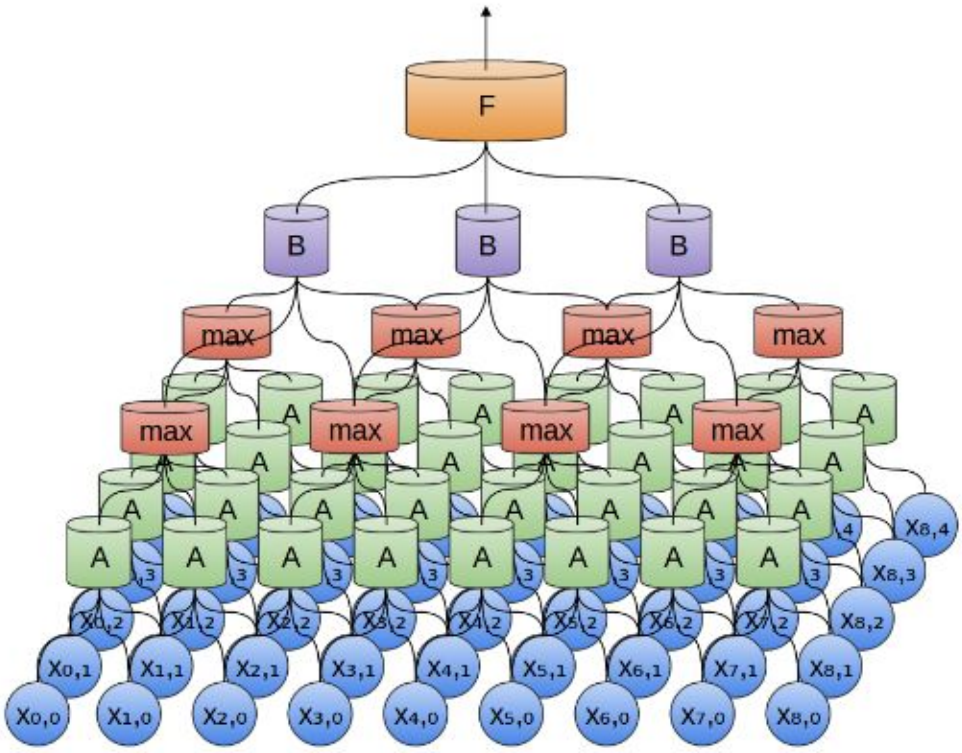
Max-pooling layers kind of "zoom out". They allow later convolutional layers to work on larger sections of the data, because a small patch after the pooling layer corresponds to a much larger patch before it. They also make us invariant to some very small transformations of the data.



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.

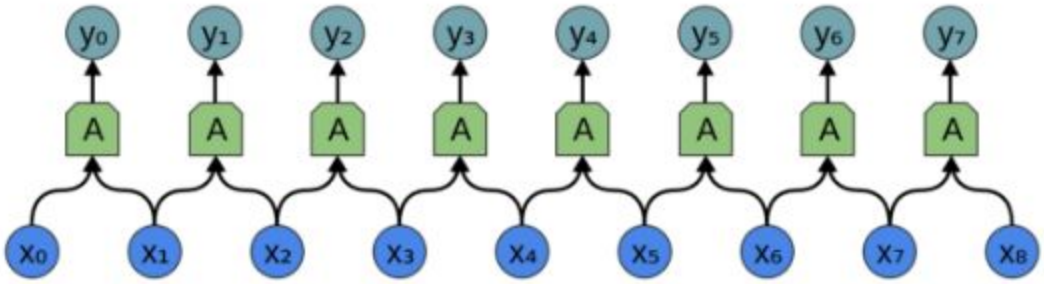


With Max pooling layer in between,



Formalizing Convolutional Neural Network

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



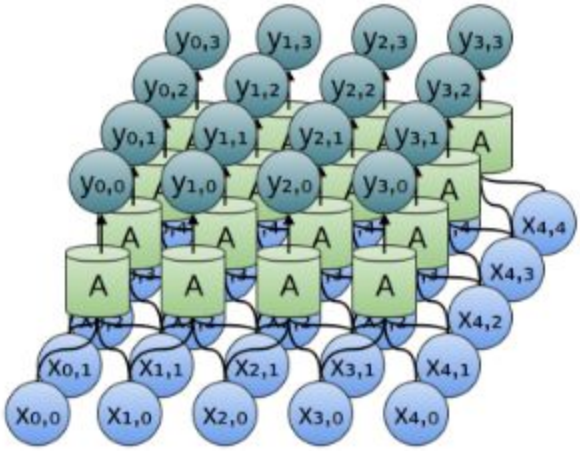
It's relatively easy to describe the outputs in terms of the inputs:

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

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

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

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 $A(x)$,

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

<we can include a timeline of month-by-month work>

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 :

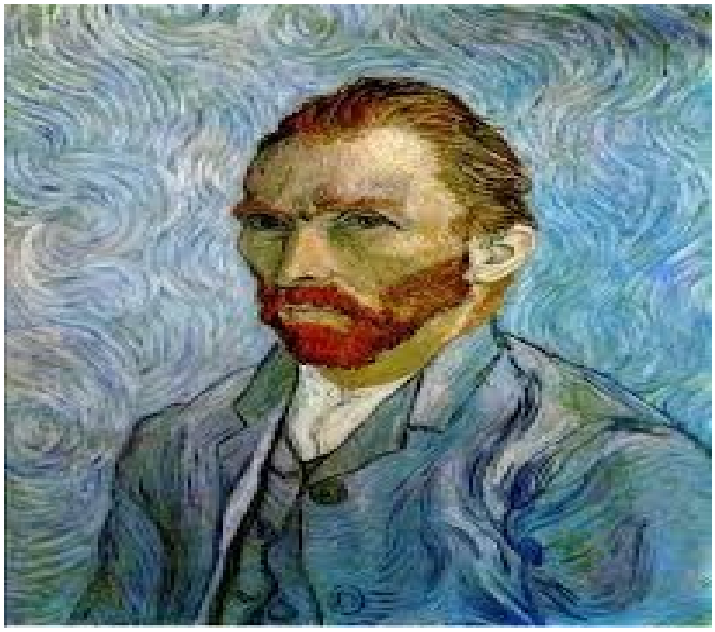
Output:

#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

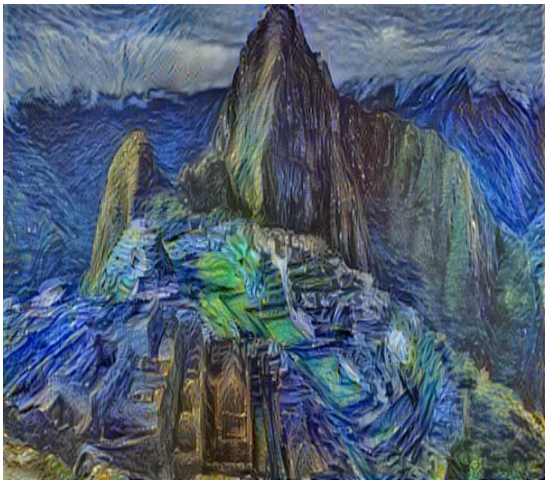
Input:

Output:

#4

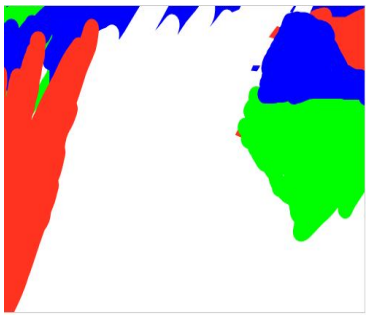
Input:

Output:

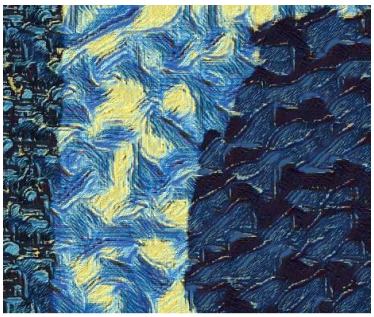


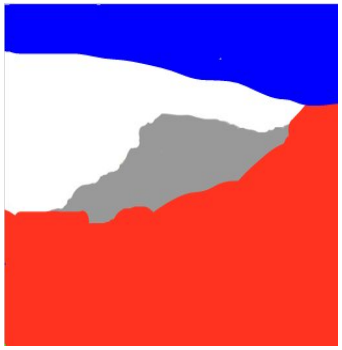
2.Neural Doodles.

Style **Doodle** **Output**










3. Videos

Style Image:

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