Deep Learning Review

Final Report

Lazaros Avgeridis - sdi1600013
Themis Varveris - sdi1600015
Team 26

Computational Geometry Spring 2020

HELLENIC REPUBLIC
National and Kapodistrian University of Athens

Department of Informatics and Telecommunications

Contents

Abstract

For our course project we picked the 2015 paper "Deep Learning Review" from authors Yann LeCun, Yoshua Bengio and Geoffrey Hinton. More specifically, we chose to elaborate on the key differrences between Conventional Machine Learning and Deep Learning, Supervised Learning, Multilayer Neural Network Architectures, Backpropagation and Convolutional Neural Networks. Furthermore, in order to gain a deeper insight into ConvNets, we implemented the paper "A Neural Algorithm of Artistic Style" by Gatys et al. also published in 2015.

1 Conventional Machine Learning

Machine-learning technology powers many aspects of modern society: from web searches to content filtering on social networks to recommendations on e-commerce websites, and it is increasingly present in consumer products such as cameras and smartphones. Machine-learning systems are used to identify objects in images, transcribe speech into text, match news items, posts or products with users interests, and select relevant results of search. Increasingly, these applications make use of a class of techniques called deep learning.

Conventional machine-learning techniques were limited in their ability to process natural data in their raw form. For decades, constructing a pattern-recognition or machine-learning system required careful engineering and considerable domain expertise to design a feature extractor that transformed the raw data (such as the pixel values of an image) into a suitable internal representation or feature vector from which the learning subsystem, often a classifier, could detect or classify patterns in the input.

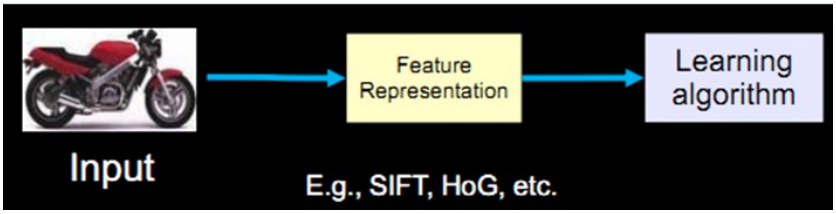


Figure 1: Machine Learning System

On the contrary, the key aspect of deep learning methods is that they allow a machine to be fed directly with raw data and to automatically discover the representations needed for detection or classification. Thus, human intervention is not required.

2 Supervised Learning

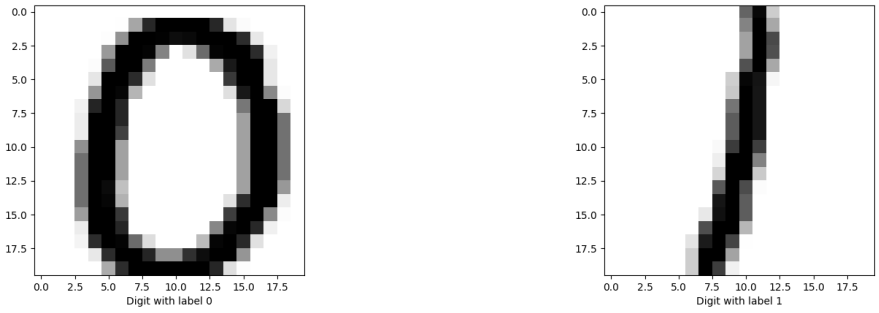
The most common form of machine learning, deep or not, is supervised learning. Our goal is to build a mathematical model from input data. The model is initially fit on a **training dataset**, that is a set of examples used to fit the parameters of the model. In practice, the training dataset often consists of pairs of an input vector (or scalar) and the corresponding output vector (or scalar), which is also commonly denoted as the **target** value (or **label**). After training, the performance of the model is measured on a different set of examples called a **test set**. This serves to test the generalization ability of the model - its ability to produce sensible answers on new inputs that it has never seen during training.

To measure the error (or distance) between the model's predictions and the actual target values, we compute an objective function. The machine then modifies its internal adjustable parameters to reduce this error. These adjustable parameters, often called weights, are real numbers that can be seen as knobs that define the input-output function of the machine. In a typical deep-learning system, there may be hundreds of millions of these adjustable weights, and hundreds of millions of labelled examples with which to train the machine. To properly adjust the weight vector, the learning algorithm computes a gradient vector that, for each weight, indicates by what amount the error would increase or decrease if the weight were increased by a tiny amount. The weight vector is then adjusted in the opposite direction to the gradient vector.

The objective function, averaged over all the training examples, can be seen as a kind of hilly landscape in the high-dimensional space of weight values. The negative gradient vector indicates the direction of steepest descent in this landscape, taking it closer to a minimum, where the output error is low on average.

2.1 A Supervised Learning Example

We used python to implement a Machine Learning Procedure called Linear Regression on a dataset of handwritten 0/1 digits (subset of the MNIST dataset):



(a) Handwritten 0 (b) Handwritten 1

Figure 2: Subset of the MNIST dataset

We formulated the above problem as an **Ordinary Least Squares** regression problem. These types of problems have a closed-form solution, which is the one we actually applied here. Our training set consisted of 11623 images and our test set consisted of 2115 images. The dimensions of each image are 20x20, so the obvious choice for representing these images is by using a vector of 400 pixels. After training, our model achieved a 99% accuracy on the test set. In practise, the algorithm tries to fit a straight line to our data (see Figure 3).

3 Multilayer Architectures and Backpropagation

A deep-learning architecture is a multilayer stack of simple modules, all (or most) of which are subject to learning, and many of which compute non-linear input-output mappings. Each module in the stack transforms its input to increase both the selectivity and the invariance of the representation. With multiple non-linear layers, say a depth of 5 to 20, a system can implement extremely intricate functions of its inputs that are simultaneously sensitive

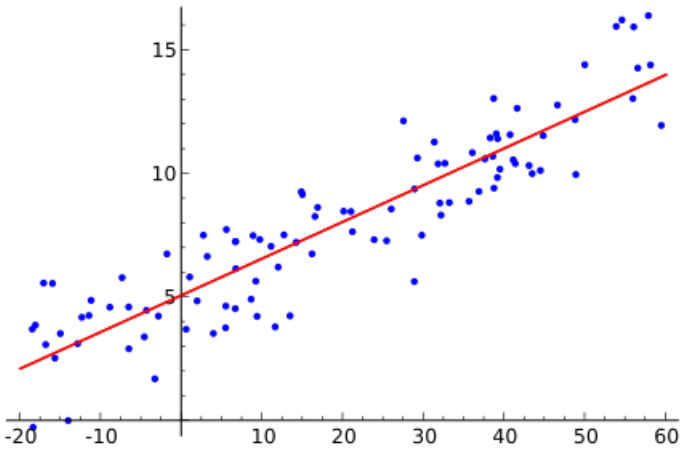


Figure 3: Linear Regression

to minute details and insensitive to large irrelevant variations such as the background, pose, lighting and surrounding objects.

3.1 Forward Pass

Many applications of deep learning use feedforward neural network architectures, which learn to map a fixed-size input (for example, an image) to a fixed-size output (for example, a probability for each of several categories). At each layer, we first compute the total input z to each unit, which is a weighted sum of the outputs of the units in the layer below. Then a non-linear function $f(.)$ is applied to z to get the output of the unit. The non-linear functions used in neural networks include the following:

1. $f(z) = \max(z,\ 0)$, the most popular non-linear function at present, called the rectified linear unit (ReLU).

2. $f(z) = \dfrac{1}{1 + e^{-z}}$, also called the logistic function or sigmoid.

3. $f(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$, which is the hyperbolic tangent or tanh.

Units that are not in the input or output layer are conventionally called *hidden units*. For simplicity, we have omitted bias terms (see Figure 4).

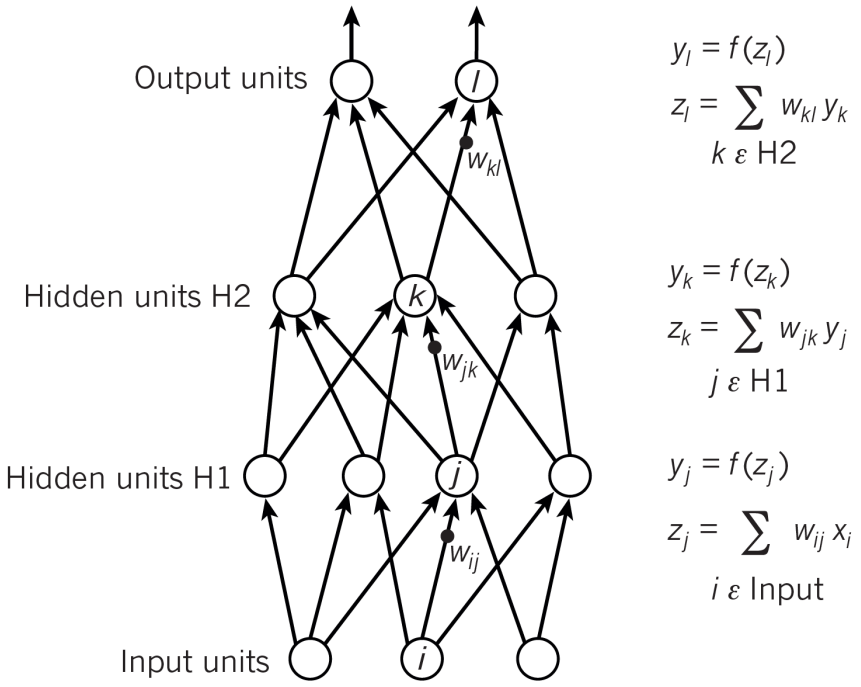


Figure 4: Forward Pass

3.2 Backpropagation

A neural network *model* consists of two components: (i) the network architecture, which defines how many layers, how many neurons, and how the neurons are connected and (ii) the parameters (values; also known as weights). In this section, we will talk about how to learn the parameters. First we will talk about parameter initialization and then optimization.

3.2.1 Parameter Intialization

Before we start training the neural network, we must select an initial value for these parameters. We do not use the value zero as the initial value. Instead, we randomly initialize the parameters to small values (e.g., normally distributed around zero; $\mathcal{N}(0, 0.1)$). Once the parameters have been initialized, we can begin training the neural network with **(stochastic) gradient descent**.

The next step of the training process is to update the parameters. After a

single forward pass through the neural network, the output will be a predicted value $\hat{y}$. We can then compute the loss $\mathcal{L}$, or the error (cost) function E, which in our case is the residual error:

$$E = \frac{1}{2}(\hat{y} - t)^2 \tag{1}$$

where t is the actual *target* value of the input. The cost function E produces a single scalar value. Given this value, we now must update all parameters in layers of the neural network. For any given layer index ℓ, we update them:

$$W^{[\ell]} = W^{[\ell]} - \alpha \frac{\partial E}{\partial W^{[\ell]}} \tag{2}$$

$$b^{[\ell]} = b^{[\ell]} - \alpha \frac{\partial E}{\partial b^{[\ell]}} \tag{3}$$

where α is the learning rate, $W^{[\ell]}$ and $b^{[\ell]}$ are the weights and biases of layer ℓ respectively. To proceed, we must compute the gradient with respect to the parameters: $\partial E/\partial W^{[\ell]}$ and $\partial E/\partial b^{[\ell]}$.

3.2.2 Optimization

Our neural network in Figure 4 has the following parameters: $W^{[1]}$, $b^{[1]}$, $W^{[2]}$, $b^{[2]}$, $W^{[3]}$, $b^{[3]}$. To update them, we use stochastic gradient descent (SGD) using the update rules in Equations (2) and (3). Firstly, we can directly compute the gradient with respect to $W^{[3]}$, $\frac{\partial E}{\partial W^{[3]}}$. The reason for this is that the influence of $W^{[1]}$ on the error is more complex than that of $W^{[3]}$. This is because $W^{[3]}$ is closer to the output $\hat{y}$, in terms of number of computations.

Before we continue by computing the gradient for $W^{[2]}$, we introduce $a^{[\ell]}$ to denote the output values or *activations* of any given layer index ℓ. Notice that the network's input units are basically $a^{[0]}$. In other words $a^{[0]} = x_i$. So, we start working backwards and instead of deriving $\partial E/\partial W^{[2]}$, we can use the chain rule of calculus. If we look at the forward propagation (Figure 4), we can see that the error E depends on $\hat{y} = a^{[3]}$:

$$\frac{\partial E}{\partial W^{[2]}} = \frac{\partial E}{\partial a^{[3]}} \frac{\partial a^{[3]}}{?} \frac{?}{\partial W^{[2]}} \tag{4}$$

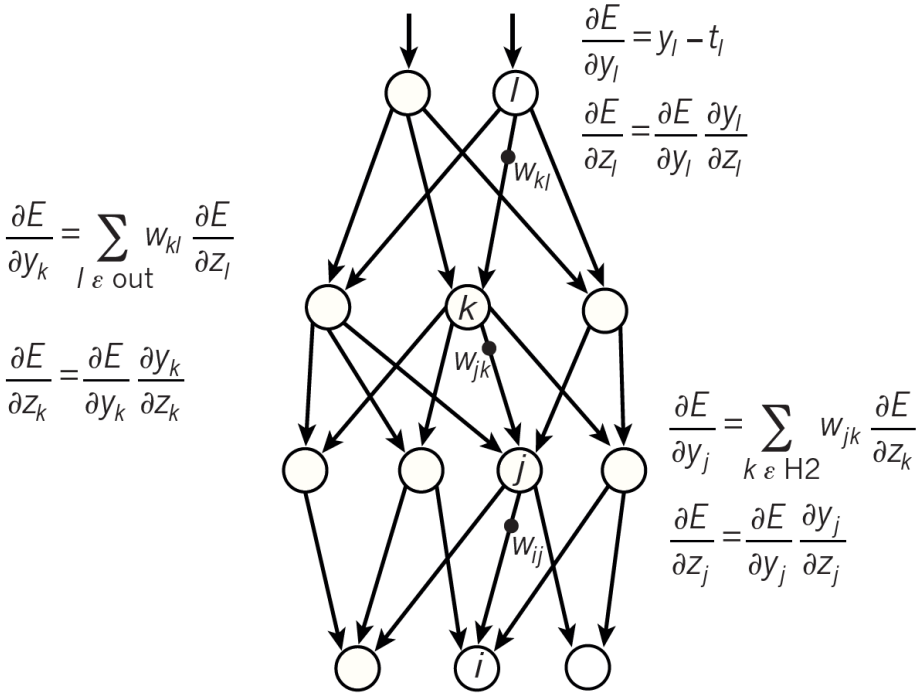


Figure 5: Backward Pass

We know that $a^{[3]}$ is directly related to $z^{[3]}$.

$$\frac{\partial E}{\partial W^{[2]}} = \frac{\partial E}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{?}\frac{?}{\partial W^{[2]}} \tag{5}$$

Furthermore, we know that $z^{[3]}$ is directly related to $a^{[2]}$. Notice that we cannot use $W^{[2]}$ or $b^{[2]}$, because to produce $z^{[3]}$ in the forward pass (Figure 4), we compute the weighted sum between Hidden units H2 outputs ($a^{[2]}$) and the weight matrix $W^{[3]}$. A common element is required for backpropagation, in this case, $a^{[2]}$.

$$\frac{\partial E}{\partial W^{[2]}} = \frac{\partial E}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{?}\frac{?}{\partial W^{[2]}} \tag{6}$$

Again $a^{[2]}$ depends on $z^{[2]}$, which $z^{[2]}$ directly depends on $W^{[2]}$, which allows us to complete the chain:

$$\frac{\partial E}{\partial W^{[2]}} = \frac{\partial E}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial W^{[2]}} \tag{7}$$

The remaining gradients can be calculated in a similar manner using back-propagation.

4 Convolutional Neural Networks

Convolutional Neural Networks (ConvNets) are designed to process data that come in the form of multiple arrays, for example a colour image composed of three 2D arrays containing pixel intensities in the three colour channels.

4.1 Overview

The architecture of a typical ConvNet is structured as a series of stages. The first few stages are composed of two types of layers: *convolutional* layers and *pooling* layers. Units in a convolutional layer are organized in feature maps, within which each unit is connected to local patches in the feature maps of the previous layer through a set of weights called a *filter bank*. The result of this local weighted sum is then passed through a non-linearity such as a ReLU.

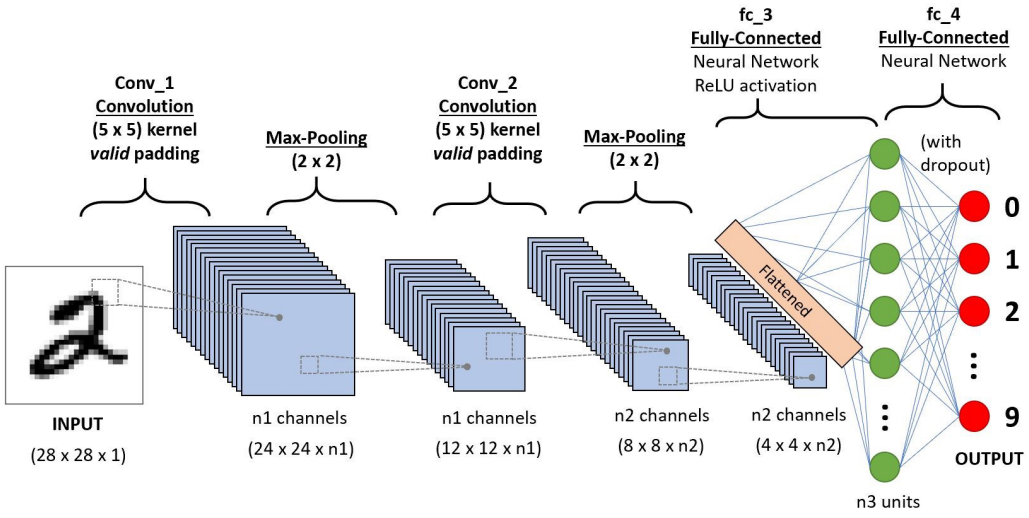


Figure 6: Convolutional Neural Network

4.2 Filters

The filter, similar to a filter encountered in signal processing, provides a measure for how close a patch of input resembles a feature. A feature may be vertical edge or an arch. The feature that the filter helps identify is not engineered manually but derived from the data through the learning algorithm. Assume we have an image of $n[H] \times n[W]$ dimensions and a filter of $f[H] \times f[W]$ dimensions. The convolved feature will be a feature of dimensions $(n[H] - f[H] + 1) \times (n[W] - f[W] + 1)$ dimensions.

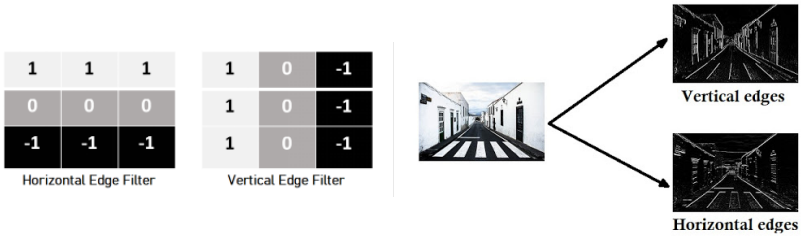


Figure 7: Filters

4.3 Strides

The amount of movement between applications of the filter to the input image is referred to as the *stride*, and it is almost always symmetrical in height and width dimensions. The dimensions of the output image after the convolving with stride S will be $\left\lfloor \frac{n[H] + 2p - f[H]}{S} + 1 \right\rfloor \times \left\lfloor \frac{n[W] + 2p - f[W]}{S} + 1 \right\rfloor$. In example, when the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.

4.4 Padding

Sometimes it is convenient to pad the input volume with zeros around the border. The size of this *zero-padding* is a hyperparameter. The nice feature of zero padding is that it will allow us to control the shrinkage of dimensions after applying filters larger than 1x1 (most commonly we use it to exactly

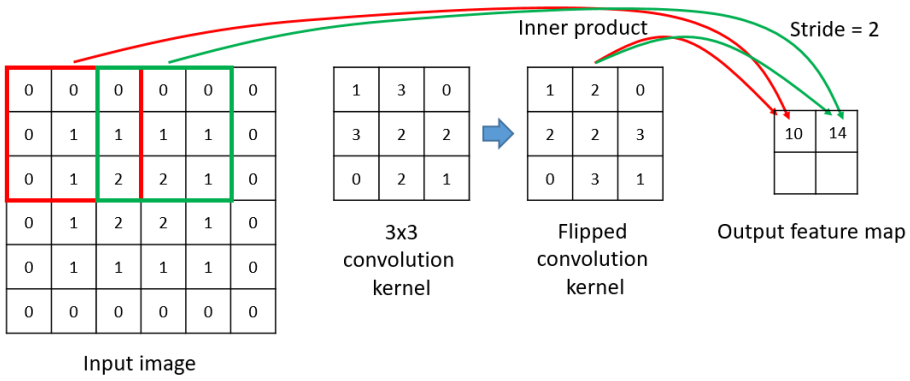


Figure 8: Stride = 2

preserve the spatial size of the input volume so the input and output width and height are the same - *same padding*). Thre is also the option of *valid padding*, which actually uses no padding at all. In addition, with the help of padding we avoid loosing information at the boundaries of an image, e.g. when weights in a filter drop rapidly away from its center. Another important note is that the filters are doing weighted sum over input patches, thus we generally want a padding that is as neutral as possible for the summation, i.e. $f(i,j) = w(i,j) + x(i,j) + w(i+1, j+1) \times 0 + \ldots$ which further justifies the zero-padding as a generic solution.

Assume we have an image of $n[H] \times n[W]$ dimensions and a filter of $f[H] \times f[W]$ dimensions. The convolved feature with padding p will be a feature of $(n[H] + 2p - f[H] + 1) \times (n[W] + 2p - f[W] + 1)$ dimensions.

4.5 Pooling

It is common to periodically insert a *pooling* layer in-between successive Conv layers in a ConvNet architecture. Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. As a result, they effectively reduce the amount of parameters and computation in the network, and hence the also control *overfitting*. Average pooling involves calculating the average for each patch of the feature map. Max pooling, is a pooling operation that calculates the maximum value in each patch of each feature map.

The most common form is a pooling layer with filters of size 2×2 applied

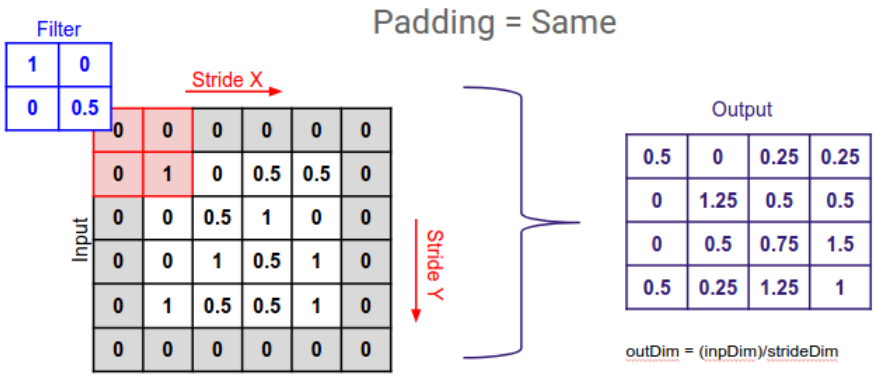


Figure 9: Zero Padding

with a stride of 2. This form of pooling downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every max pooling operation would in this case be taking a max over 4 numbers (little 2×2 region in some depth slice). The depth dimension remains unchanged.

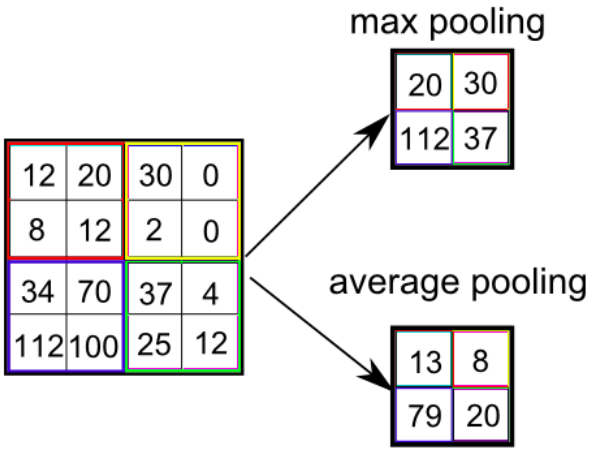


Figure 10: Pooling

5 Neural Style Transfer

5.1 Problem Description

Neural Style Transfer (NST) is a deep learning technique, which merges two images, namely a **"content" image (C)** and a **"style" image (S)**, to create a **"generated" image (X)**. The generated image X combines the content of image C with the style of image S.

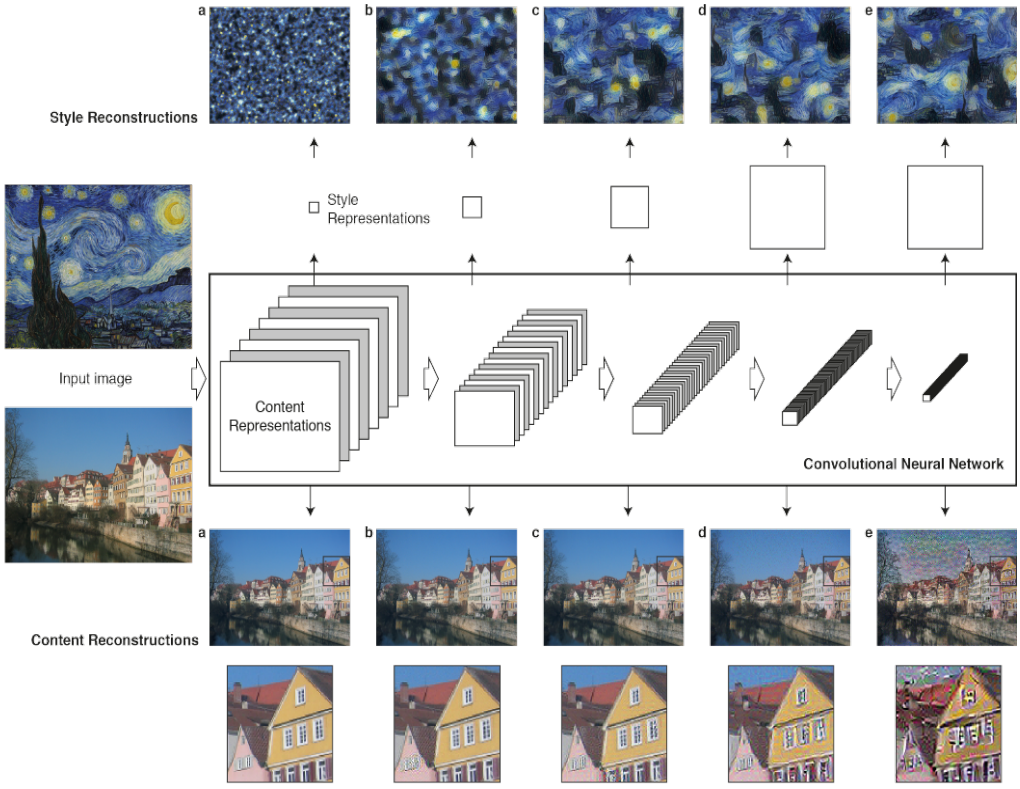


Figure 11: Content and Style Reconstructions

5.2 Observations

In a **Convolutional Neural Network (CNN)** a given input image is represented as a set of filtered images at each processing stage in the CNN. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some downsampling mechanism (e.g. max-pooling) leading to a decrease in the total number of units per layer of the network.

Regarding the **content reconstructions**, we can visualize the information at different processing stages in the CNN by reconstructing it from only knowing the networks responses in a particular layer. We reconstruct the content image from layers: (**a**) 'conv1_1', (**b**) 'conv2_1', (**c**) 'conv3_1', (**d**) 'conv4_1', (**e**) 'conv5_1' of the original VGG-Network. We find that reconstruction from lower layers is almost perfect (**a,b,c**). In higher layers of the network, detailed pixel information is lost while the high-level content of the image is preserved (**d,e**).

When it comes to the **style reconstructions**, on top of the original CNN representation we built a new feature space the captures the style of an input image. The style representation computes correlations between the different features in different layers of the CNN. We reconstruct the style of the input image from style representations built on different subsets of CNN layers ((**a**) 'conv1_1', (**b**) 'conv1_1' and 'conv2_1', (**c**) 'conv1_1', 'conv2_1' and 'conv3_1', (**d**) 'conv1_1', 'conv2_1', 'conv3_1' and 'conv4_1', (**e**) 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1' and 'conv5_1'). This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.

The key finding of this paper is that the representations of *content* and *style* in the Convolutional Neural Network are separable. That is, we can manipulate both representations independently to produce new, perceptually meaningful images. We demonstrate this by generating new, artistic images that combine the style of several well-known paintings with the content of an arbitrarily chosen photograph. In particular, we derive the neural representations for the content and style of an image from the feature responses of a high performing Deep Neural Network trained on object recognition.

Here is a glimpse of our generated images:

(a) Content Image (b) Style Image (c) Generated Image

Figure 12: Dit in the style of 'The Scream' by Edvard Munch

(a) Content Image (b) Style Image (c) Generated Image

Figure 13: The Parthenon in the style of 'The Starry Night' by Van Gogh

5.3 Methods

For the results presented above we used the feature space provided by the 16 convolutional and 5 pooling layers of the (pretrained) 19 layer VGG-Network, a Convolutional Neural Network that was introduced and extensively described in [1]. We did not use any of the fully connected layers. The paper suggests that replacing the *max-pooling* operation by *average pooling* improves the gradient flow and one obtains slightly more appealing results.

5.3.1 Content Loss

A given input image C is encoded in each layer of the CNN by the filter responses to that image. A layer with n_C distinct filters has n_C feature maps each of size $n_H \times n_W$, where n_H is the height and n_W is the width of each feature map. So the responses in a layer ℓ can be stored in a matrix $F^{[\ell]} \in R^{n_C \times m_{H,W}}$, where $m_{H,W} = n_H \times n_W$ and $F_{ij}^{[\ell]}$ is the activation of the i^{th} filter at position j in layer ℓ (Figure 14). So let C and X be the original

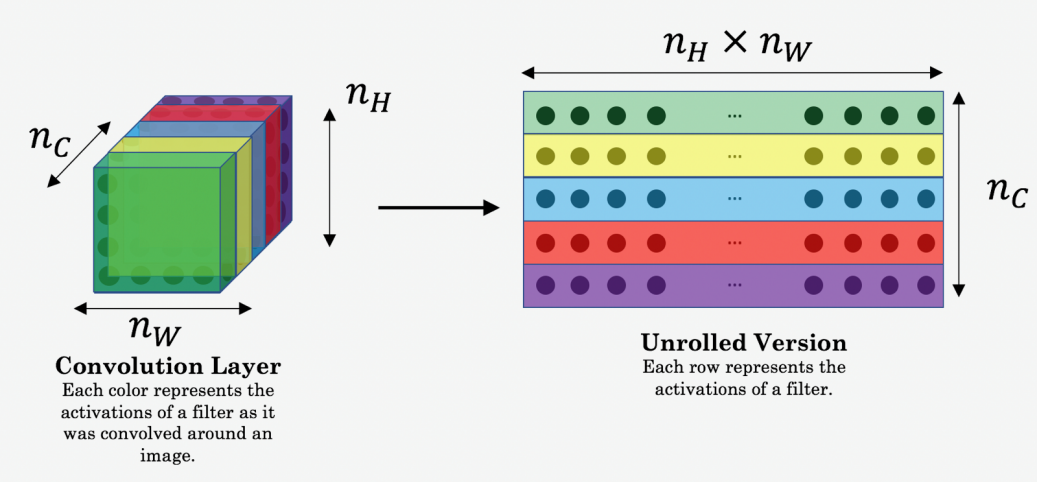


Figure 14: CNN Layer "Unrolling"

image and the image that is generated and $C^{[\ell]}$ and $X^{[\ell]}$ their respective feature representation in layer ℓ. We then define the (squared-error) loss between the two feature representations

$$\mathcal{L}_{content}(C, X) = \frac{1}{2} \sum_{i,j} (C_{ij}^{[\ell]} - X_{i,j}^{[\ell]})^2 \tag{8}$$

5.3.2 Style Loss

On top of the CNN responses in each layer of the network we built a style representation that computes the correlations between the different filter responses, where the expectation is taken over the spatial extend of the input image. These feature correlations are given by the "Gram matrix" $G^{[\ell]} \in R^{n_C \times n_C}$, where $G_{ij}^{[\ell]}$ is the inner product between the vectorized feature map i and j in layer ℓ. In other words $G_{ij}^{[\ell]}$ compares how similar the activations of filter i are to the activations of filter j. If they are highly similar, we would expect them to have a large inner product, and thus $G_{ij}^{[\ell]}$ to be large.

$$G_{ij}^{[\ell]} = \sum_k F_{ik}^{[\ell]} F_{jk}^{[\ell]} \tag{9}$$

We compute the Gram matrix by multiplying the "unrolled" filter matrix with their transpose (Figures 14, 15). The result is a matrix of dimension

$n_C \times n_C$.

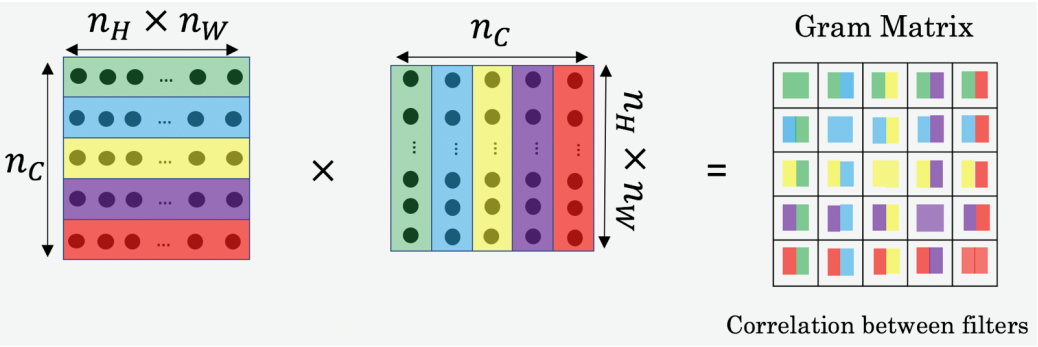


Figure 15: Gram Matrix Computation

One important part of the gram matrix is that the diagonal elements such as $G_{ii}^{[\ell]}$ also measure how active filter i is. For example, suppose filter i is detecting vertical textures in the image. Then $G_{ii}^{[\ell]}$ measures how common vertical textures are in the image as a whole: If $G_{ii}^{[\ell]}$ is large, this means that the image has a lot of vertical texture. By capturing the prevalence of different types of features ($G_{ii}^{[\ell]}$), as well as how much different features occur together ($G_{ij}^{[\ell]}$), the Style matrix $G^{[\ell]}$ measures the style of an image.

To measure the contribution of a specific layer ℓ to the total style loss we basically minimise the mean-squared distance between the entries of the Gram matrix from the original image and the Gram matrix of the image to be generated. So let S and X be the original (input) image and the image that is generated. The contribution of a *single layer* ℓ to the style total loss is then:

$$\mathcal{L}_{style}^{[\ell]}(S, X) = \frac{1}{4 \times (n_C)^2 \times (m_{H,W})^2} \sum_{i,j} (G_{ij}^{(S)} - G_{ij}^{(X)})^2 \tag{10}$$

In the equation above, we should write $G_{ij}^{(S)[\ell]}$ and $G_{ij}^{(X)[\ell]}$, but we dropped the superscript $[\ell]$ to simplify the notation. The total style loss is

$$\mathcal{L}_{style}(S, X) = \sum_{\ell=0}^{L} w_\ell \mathcal{L}_{style}^{[\ell]}(S, X) \tag{11}$$

where w_ℓ are weighting factors of the contribution of each layer to the total style loss.

5.3.3 Total Loss

To generate the images that mix the content of a photograph with the style of a painting we jointly minimise the distance of a white noise image from the content representation of the photograph in one layer of the network and the style representation of the painting in a number of layers of the CNN. So let C be the photograph, S be the artwork and X be the generated image. Using equations (8) and (11) the loss function we minimize is

$$\mathcal{L}_{total}(C, S, X) = \alpha \mathcal{L}_{content}(C, X) + \beta \mathcal{L}_{style}(S, X) \qquad (12)$$

where α and β are the weighting factors for content and style reconstruction respectively.

In the future we will examine if adding a total variation loss term to the total loss function above yields even better results.

References

[1] Simonyan K. and Zisserman A. Very Deep Convolutional Networks for Large-Scale Image Recognition. `http://arxiv.org/abs/1409.1556`, (2015).

[2] LeCun Y., Bengio Y., and Hinton G. Deep Learning. `https://www.nature.com/articles/nature14539`, (2015).

[3] Gatys L. A., Ecker A. S., and Bethge M. A Neural Algorithm of Artistic Style. `https://arxiv.org/abs/1508.06576`, (2015).