ALGERIAN REPUBLIC DEMOCRATIC AND POPULAR
MINISTRY OF HIGH EDUCATION AND SCIENTIFIC RESEARCHES

University of Mohamed Khider Biskra
Faculty of Exact Science and Science of Life and Nature
Department of Computer Science

Dissertation For Master Degree Graduation In Computer Science
Field Computer Graphics & Computer Vision

Title:

Grayscale Image Colorization

Using Deep Learning

Proposed and supervised by:
Pr. Kamal Eddine Melkemi

Realized by:
Khaldi Mostafa

Defended the 25/06/2018, in front of the jury composed of:

Brima Salima	President
Mokhtari Bilal	Supervisor
Benameur Sabrina	Examiner

Academic year 2017-2018

Summary

Abstract

Humans can distinguish between objects by recognizing their shapes and colors thus they can tell if this is a car or a track, this ability becomes even rich and accurate by each time the humans learn from nature and environment, which computers stands blind if they are programmed for certain kind of objects and they encountered new input of a new object that they have no prerequisites about.

How to teach a computer? You can either write a fixed program or you can enable the computer to learn on its own. Living beings do not have any programmer writing a program for developing their skills, which then only has to be executed. They learn by themselves without the previous knowledge from external impressions and thus can solve problems better than any computer today. What qualities are needed to achieve such a behavior for devices like computers? Can such cognition be adapted from biology?

Introduction

Deep Learning is a new area of Machine Learning research, which has been introduced with the objective of moving Machine Learning closer to one of its original goals: Artificial Intelligence.

Deep Learning is about learning multiple levels of representation and abstraction that helps to make sense of data such as images, sounds, and texts. Deep Learning uses the concept of artificial neural networks that is based from the biological neural networks which is the main reason why the human is a fast learner, with such inspiration it is possible to teach a program to learn from images and recognize its content and analyze it.

Computers can act like humans, matter fact, they can be more efficient than human, If we compare computers and human brain, we will note that, theoretically, the computer should be more powerful than our brain, it comprises 10^9 transistors with a switching time of 10^{-9} seconds. The brain contains 10^{11} neurons, but these only have a switching time of about 10^{-3} seconds.

Images can contain a group of different objects, thus, every object is represented with a different shape and color, so every corner of an image can make sense, human perception can distinguish easily between these objects, and sometime, it is even harder for a human to classify images due to the objects overlapping or a partial deformation of the image, and to make a computer classify efficiently the images, it has to recognize and to learn every corner of the image, and distinguish the differences between the wide variety of objects, and by gathering more information of what the images can contain, an algorithm would act like a human brain that interprets any new information based on its prerequisites.

Grayscale image colorization was an interesting spot that made so many artists trying to express their skills by using an image editing software to colorize manually the grayscale images, the process depends strongly on the user skills, and it recquires a long time to make the colorization happens.

Deap learning provides a great way to do what artists previously done in a few days in a few seconds, by matching the brain ability of learning to create an artificial neural network that extracts features from colored images, and projects these prerequisites on a grayscale images, and so, we will be able to achieve our automatic colorization goal.

In this particular article, we will learn about neural networks, deep learning and how to colorize a grayscale image.

History

The history of neural networks begins in the early 1940's and thus nearly simultaneously with the history of programmable electronic computers. The youth of this field of research, as with the field of computer science itself, can be easily recognized due to the fact that many of the cited persons are still with us.[7]

- **1943:** Warren McCulloch and Walter Pitts introduced models of neurological networks, recreated threshold switches based on neurons and showed that even simple networks of this kind are able to calculate nearly any logic or arithmetic function. Furthermore, the first computer precursors ("electronic brains") were developed, among others supported by Konrad Zuse , who was tired of calculating ballistic trajectories by hand.[7]

- **1947:** Walter Pitts and Warren McCulloch indicated a practical field of application (which was not mentioned in their work from 1943), namely the recognition of spacial patterns by neural networks.[7]

- **1949:** Donald O. Hebb formulated the classical Hebbian rule which represents in its more generalized form the basis of nearly all neural learning procedures. The rule implies that the connection between two neurons is strengthened when both neurons are active at the same time. This change in strength is proportional to the product of the two activities. Hebb could postulate this rule, but due to the absence of neurological research he was not able to verify it.[7]

- **1950:** The neuropsychologist Karl Lashley defended the thesis that brain information storage is realized as a distributed system. His thesis was based on experiments on rats, where only the extent but not the location of the destroyed nerve tissue influences the rats' performance to find their way out of a labyrinth.[7]

- **1951:** For his dissertation Marvin Minsky developed the neurocomputer Snark, which has already been capable to adjust its weights3 automatically. But it has never been practically implemented, since it is capable to busily calculate, but nobody really knows what it calculates.[7]

- **1956:** Well known scientists and ambitious students met at the Dartmouth Summer Research Project and discussed, to put it crudely, how to simulate a brain. Differences between top-down and bottom-up research developed. While the early supporters of artificial intelligence wanted to simulate capabilities by means of software, supporters of neural networks wanted to achieve system behavior by imitating the smallest parts of the system the neurons.[7]

- **1957-1958:** At the MIT, Frank Rosenblatt, Charles Wightman and their coworkers developed the first successful neurocomputer, the Mark I perceptron , which was capable to recognize simple numerics by means of a 20×20 pixel image sensor and electromechanically worked with 512 motor driven potentiometers each potentiometer representing one variable weight.[7]

- **1959:** Frank Rosenblatt described different versions of the perceptron, formulated and verified his perceptron convergence theorem. He described neuron layers mimicking the retina, threshold switches, and a learning rule adjusting the connecting weights.[7]

- **1961:** Karl Steinbuch introduced technical realizations of associative memory, which can be seen as predecessors of today's neural associative memories. Additionally, he described concepts for neural techniques and analyzed their possibilities and limits.[7]

- **1969:** Marvin Minsky and Seymour Papert published a precise mathematical analysis of the perceptron to show that the perceptron model was not capable of representing many important problems (keywords: XOR problem and linear separability), and so put an end to overestimation, popularity and research funds. The implication that more powerful models would show exactly the same problems and the forecast that the entire field would be a research dead end resulted in a nearly complete decline in research funds for the next 15 years no matter how incorrect these forecasts were from today's point of view. [7]

- **1985:** John Hopfield published an article describing a way of finding acceptable solutions for the Travelling Sales man problem by using Hopfield nets.[7]

- **1986:** The back propagation of error learning procedure as a generalization of the delta rule was separately developed and widely published by the Parallel Distributed Processing Group : Non-linearly-separable problems could be solved by multilayer perceptrons, and Marvin Minsky's negative evaluations were disproven at a single blow. At the same time a certain kind of fatigue spread in the field of artificial intelligence, caused by a series of failures and unfulfilled hopes.[7]

- **From this time on:** the development of the field of research has almost been explosive. It can no longer be itemized, but some of its results will be seen in the following.[7]

1 | Neural Networks

Artificial neural networks are an attempt at modeling the information processing capabilities of nervous systems. Thus, first of all, we need to consider the essential properties of biological neural networks from the viewpoint of information processing. This will allow us to design abstract models of artificial neural networks, which can then be simulated and analyzed.[9]

We start by considering biological systems. Artificial neural networks have aroused so much interest in recent years, not only because they exhibit interesting properties, but also because they try to mirror the kind of information processing capabilities of nervous systems. Since information processing consists of transforming signals, we deal with the biological mechanisms for their generation and transmission. We discuss those biological processes by which neurons produce signals, and absorb and modify them in order to retransmit the result. In this way biological neural networks give us a clue regarding the properties which would be interesting to include in our artificial networks.[9]

1.1 The biological neural network

The biological neuron is a nerve cell that provides the fundamental functional unit for the nervous systems of all living being. Neurons exist to communicate with one another, and pass electrochemical impulses across synapses, from one cell to the next, as long as the impulse is strong enough to activate the release of chemicals across a synaptic cleft. The strength of the impulse must surpass a minimum threshold or chemicals will not be released.[8]

The Figure 1.1 presents the major parts of the nerve cell:

- Soma.
- Dendrites.
- Axons.
- Synapses.

The neuron is made up of a nerve cell consisting of a soma (cell body) that has many dendrites but only one axon. The single axon can branch hundreds of times, however. Dendrites are thin structures that arise from the main cell body. Axons are nerve fibers with a special cellular extension that comes from the cell body.

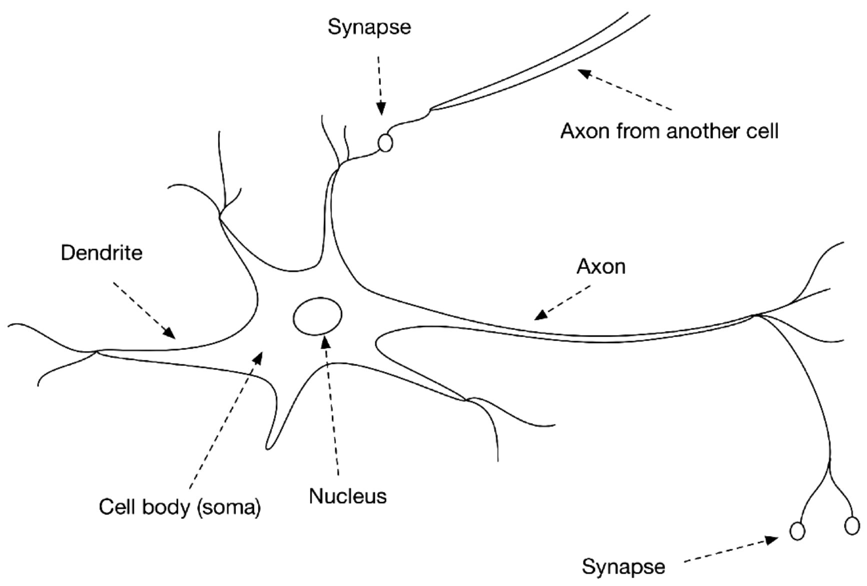


Figure 1.1: The biological neuron

1.1.1 Synapses

Synapses are the connecting junction between axon and dendrites. The majority of synapses send signals from the axon of a neuron to the dendrite of another neuron. The exceptions for this case are when a neuron might lack dendrites, or a neuron lacks an axon, or a synapse, which connects an axon to another axon.[8]

1.1.2 Dendrites

Dendrites have fibers branching out from the soma in a bushy network around the nerve cell. Dendrites allow the cell to receive signals from connected neighboring neurons and each dendrite is able to perform multiplication by that dendrite's weight value. Here multiplication means an increase or decrease in the ratio of synaptic neurotransmitters to signal chemicals introduced into the dendrite.[8]

1.1.3 Axons

Axons are the single, long fibers extending from the main soma. They stretch out longer distances than dendrites and measure generally 1 centimeter in length (100 times the diameter of the soma). Eventually, the axon will branch and connect to other dendrites. Neurons are able to send electrochemical pulses through cross membrane voltage changes generating action potential. This signal travels along the cell's axon and activates synaptic connections with other neurons.[8]

1.1.4 Information flow across the biological neuron

Synapses that increase the potential are considered excitatory, and those that decrease the potential are considered inhibitory. Plasticity refers the long term changes in strength of connections in response to input stimulus. Neurons also have been shown to form new connections over time and even migrate. These combined mechanics of connection change drive the learning process in the biological brain.[8]

1.1.5 From biological to artificial

The living being brain has been shown to be responsible for the fundamental components of the mind. We can study the basic components of the brain and understand them. Research has shown ways to map out functionality of the brain and track signals as they move through neurons.[8]

1.2 The perceptron

1.2.1 Definition

The perceptron is a linear model with a simple input output relationship as depicted in Figure 1.2, which shows we're summing n number of inputs multiplied with their associated weights and then sending this input to a another function with a defined threshold. Normally with perceptrons, this is a Heaviside step function with a threshold value of 0.5. This function will give a real valued single value (0 or a 1), depending on the input.[4]

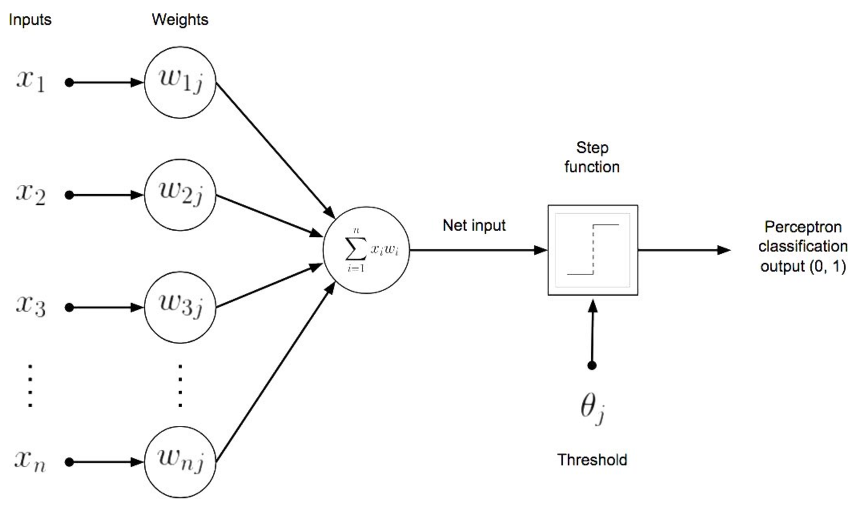


Figure 1.2: Single layer perceptron

We can model the decision boundary and the classification output in the Heaviside step function equation, as follows:

$$f(x) = \begin{cases} 3 & x<0 \\ 0 & x>=0 \end{cases} \tag{1.1}$$

To produce the input to the activation function, we take the dot product of the input and the connection weights. We see this summation in the left side of Figure 1.2 as the input to the summation function. Table 1.1 provides an explanation of how the summation function is performed as well as notes about the parameters involved in the summation function.[5]

Function parameter	Description
w	Vector of real valued weights on the connections
$w.x$	Dot product
n	Number of inputs to the perceptron
b	The bias term

Table 1.1: The summation function parameters

The output of activation function is the output for the perceptron and gives us a classification of the input values. If the bias value is negative, it forces the learned weights sum to be a much greater value to get a classification output. The bias moves the decision boundary around for the model. Input values do not affect the bias, but the bias is learned through the perceptron learning algorithm. As a basic linear classifier we consider the single-layer perceptron to be the simplest form of the family of shallow neural networks.[5]

1.2.2 The perceptron learning algorithm

The perceptron learning algorithm changes the weights in the perceptron model until all input records are all correctly classified. The algorithm will not come to an end if the learning input is not linearly separable. A linearly separable dataset is one for which we can find the values of a hyperplane that will cleanly divide the two classes of the dataset.

The perceptron learning algorithm initializes the weight vector with small random values. The perceptron learning algorithm takes each input record, as we can see in Figure 1.2, and computes the output classification to verify against the actual classification label. To produce the classification, the columns are matched up to weights where n is the number of dimensions in both our input and weights. The first input value is the bias input, which is always 1.0 because we don't affect the bias input. The first weight is our bias in this diagram. The dot product of the input vector and the weight vector allow the input to feature in the activation function, as we've previously discussed.

If the classification is correct, no weight adjustment is made. If the classification is incorrect, the weights are adjusted accordingly. Weights are updated between individual training examples in an "online learning" fashion. This loop continues until all of the input examples are correctly classified. If the dataset is not linearly separable, the training algorithm will not end. Table 1.2 demonstrates a dataset that is not linearly separable, the XOR logic function.[8]

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.2: The XOR function

A basic perceptron can not solve the XOR logic modeling problem, illustrating an early limitation of the perceptron model.

1.2.3 Limitations of the early perceptron

After initial promise, the perceptron was found to be limited in the types of patterns it could recognize. The initial inability to solve nonlinear (e.g., datasets that are not linearly separable) problems was seen as a failure for the field of neural networks. However, what the general industry did not widely realize was that a multilayer perceptron could solve the XOR problem, among many other nonlinear problems.[8]

1.3 The Feed-Forward multilayers neural network

The most well known and simplest to understand neural network is the feed-forward multilayer neural network. It is a neural network with an input layer, one or more hidden layers, and an output layer. Each layer has one or more artificial neurons. These artificial neurons are similar to their perceptron precursor yet have a different activation function depending on the layer's specific purpose in the network. We'll look more closely at the layer types in multilayer perceptrons later in the chapter. For now, let's look more closely at this evolved artificial neuron that emerged from the limitations of the single-layer perceptron.[8]

1.3.1 Evolution of the artificial neuron

The artificial neuron of the multilayer perceptron is similar to its predecessor, the perceptron, but it adds flexibility in the type of activation layer we can use. Figure 1.3 shows an updated diagram for the artificial neuron that is based on the perceptron.

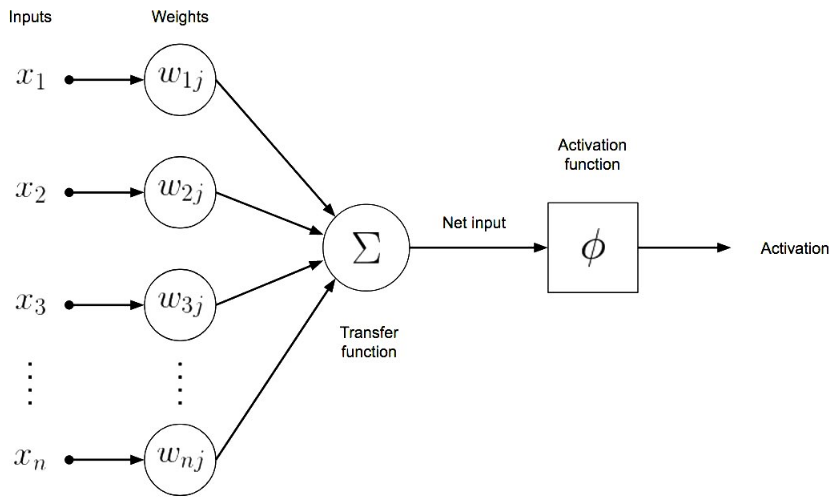


Figure 1.3: Artificial neuron for a multilayer perceptron

This diagram is similar to Figure 1.2 for the single-layer perceptron, yet we notice a more generalized activation function. We'll develop this diagram further in a detailed look at the artificial neuron as we proceed.[8]

1.3.2 Artificial neuron input

The artificial neuron Figure 1.4 takes input that, based on the weights on the connections, can be ignored (by a 0.0 weight on an input connection) or passed on to the activation function. The activation function also has the ability to filter out data if it does not provide a non-zero activation value as output.[8]

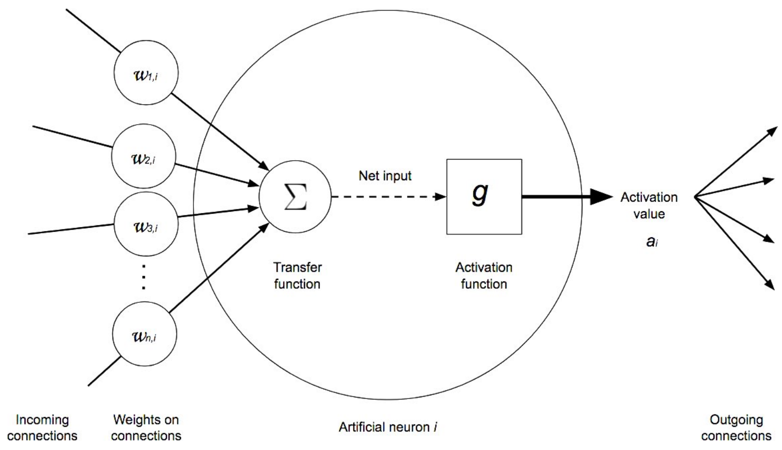


Figure 1.4: Details of an artificial neuron in a multilayer perceptron neural network

We express the net input to a neuron as the weights on connections multiplied by activation incoming on connection, as shown in previous figure. For the input layer, we're just taking the feature at that specific index, and the activation function is linear (it passes on the feature value). For hidden layers, the input is the activation from other neurons. Mathematically, we can express the net input (total weighted input) of the artificial neuron as

$$input\_sum_i = W_i.A_i \tag{1.2}$$

where W_i is the vector of all weights leading into neuron i and A_i is the vector of activation values for the inputs to neuron i. Let's build on this equation by accounting for the bias term that is added per layer (explained further below):

$$input\_sum_i = W_i.A_i + b \tag{1.3}$$

To produce output from the neuron, we'd then wrap this net input with an activation function σ, as demonstrated in the following equation:

$$a_i = \sigma(input\_sum_i) \tag{1.4}$$

We can then expand this function with the definition of $input_i$:

$$a_i = \sigma(input\_sum_i) \tag{1.5}$$

This activation value for neuron i is the output value passed on to the next layer through connections to other artificial neurons (multiplied by weights on connections) as an input value.

If our activation function is the *sigmoid* function, we'll have this:

$$g(z) = \frac{1}{(1 + e^{-z})} \tag{1.6}$$

This output will have the range $[0, 1]$, which is the same output as the logistic regression function.

The inputs are the data from which you want to produce information, and the connection weights and biases are the quantities that govern the activity, activating it or not. As with the perceptron, there is a learning algorithm to change the weights and bias value for each artificial neuron. During the training phase, the weights and biases change as the network learns. We'll cover the learning algorithm for neural networks later in the chapter.

Just as biological neurons don't pass on every electro-chemical impulse they receive, artificial neurons are not just wires or diodes passing on a signal. They are designed to be selective. They filter the data they receive, and aggregate, convert, and transmit only certain information to the next neuron(s) in the network. As these filters and transformations work on data, they convert raw input data to useful information in the context of the larger multilayer perceptron neural network. We illustrate this effect more in the next section.

Artificial neurons can be defined by the kind of input they are able to receive (binary or continuous) and the kind of transform (activation function) they use to produce output.[8]

1.3.3 Connection weights

Weights in a neural network are coefficients that amplify or minimize the input signal to a given neuron in the network. In common representations of neural networks, these are the lines going from one point to another, the edges of the mathematical graph. Often, connections are notated as w in mathematical representations of neural networks.[5]

1.3.4 Biases

Biases are scalar values supplied to the input to ensure that at least a few nodes per layer are activated regardless of signal strength. Biases make the learning possible to happen by giving the network action in the event of low signal. They allow the network to try new interpretations or behaviors. Biases are generally notated b, and, like weights, biases are edited throughout the training phase.[4]

1.3.5 Activation function

The functions that control the artificial neuron's behavior are called activation functions. The transmission of that input is known as forward propagation. Activation functions transform the set of inputs, weights, and biases. Products of these transforms are input for the next node layer. Many (but not all) nonlinear transforms used in neural networks transform the data into a convenient range, such as 0 to 1. When an node passes on a nonzero value to another node, it is said to be activated.[4]

1.3.6 Feed-forward neural network architecture

We can better understand the structure of the full multilayer feed-forward neural network. With multilayer feed-forward neural networks, we have artificial neurons arranged into groups called layers. Building on the layer concept, we see that the multilayer neural network has the following concept:[4]

- A single input layer.
- One or many hidden layers, fully connected.
- A single output layer.

As Figure 1.5 shows, the neurons in each layer (represented by the circles) are all fully connected to all neurons in all adjacent layers.

The neurons in each layer all use the same kind of activation function (most of the time). For the input layer, the input is the raw vector input. The input to neurons of the other layers is the output of the previous layer's neurons. As data flows through the network in a feed-forward fashion, it is impacted by the connection weights and the activation function kind. Let's now take a look at the specifics of each layer type.[4]

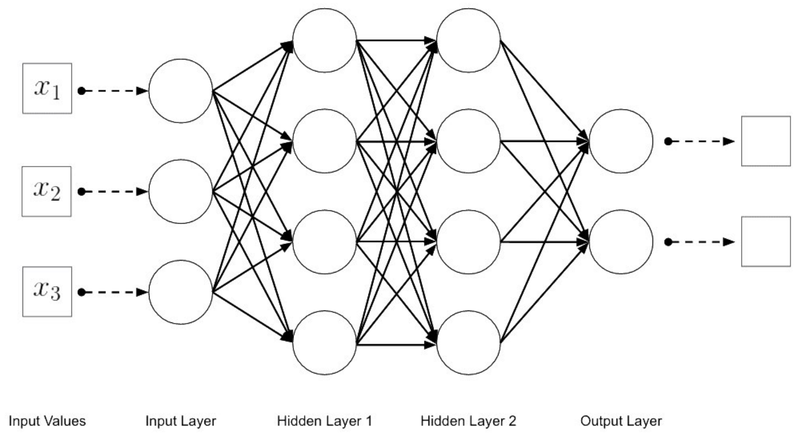


Figure 1.5: Fully connected multilayer feed-forward neural network topology

1.3.7 input layer

This layer is how we get input data (vectors) fed into our network. The number of neurons in an input layer is typically the same number as the input feature to the network. Input layers are followed by one or more hidden layers (explained in the next section). Input layers in classical feed-forward neural networks are fully connected to the next hidden layer, yet in other network architectures, the input layer might not be fully connected. [8]

1.3.8 Hidden layer

There are one or more hidden layers in a feed-forward neural network. The weight values on the connections between the layers are how neural networks encode the learned information extracted from the raw training data. Hidden layers are the key to allowing neural networks to model nonlinear functions, as we saw from the limitations of the single-layer perceptron networks.[8]

1.3.9 Output layer

We get the answer or prediction from our model from the output layer. Given that we are mapping an input space to an output space with the neural network model, the output layer gives us an output based on the input from the input layer. Depending on the setup of the neural network, the final output may be a real-valued output (regression) or a set of probabilities (classification). This is controlled by the type of activation function we use on the neurons in the output layer. The output layer typically uses either a $softmax$ or $sigmoid$ activation function for classification.[8]

1.3.10 Connection between layers

In a fully connected feed-forward network, the connections between layers are the outgoing connections from all neurons in the previous layer to all of the neurons in the next layer. We change these weights progressively as our algorithm finds the best solution it can with the back-propagation learning algorithm. We can understand the weights mathematically by thinking of them as the parameter vector to minimize error.[8]

2 | Deep Learning

2.1 Introduction to deep learning

Deep learning was hard to define for many because it has changed forms slowly over the past decade. One definition specifies that deep learning is a neural network with more than two layers. The problematic aspect to this definition is that it makes deep learning sound as if it has been around since the 1980s. We feel that neural networks had to be developed architecturally from the earlier network styles (in conjunction with a lot more processing power) before showing the spectacular results seen in more recent years. Following are some of the facets in this evolution of neural networks:[3]

- More neurons than previous networks.[3]
- More complex ways of connecting layers/neurons than the shallow neural networks.[3]
- Explosion in the amount of computing power available to train.[3]
- Automatic feature extraction. [3]
- More neurons means that the neuron count has risen over the years to express more complex models.[3]
- Layers also have evolved from each layer being fully connected in multilayer networks to locally connected patches of neurons between layers in Convolutional Neural Networks (CNNs) and recurrent connections to the same neuron in Recurrent Neural Networks.[1]
- More connections means that the networks have more parameters to optimize, and this required a powerful machines that has to be invented over the past 20 years.[1]

To further provide a clear vision of deep learning, here we define the four major architectures of deep networks:[10]

- Unsupervised Pre-trained Networks.
- Convolutional Neural Networks.
- Recurrent Neural Networks.
- Recursive Neural Networks.[10]

2.2 History of deep learning application

- Advances in modeling sequential data with recurrent neural networks appeared in the late-1980s and early 1990s by researchers such as Sepp Hochreiter. As time went on, the research community created better artificial neuron variants (e.g., Long Short-Term Memory [LSTM] Memory Cell and Memory Cell with Forget Gate) over the course of the late 1990s. The stage for a neural network resurgence was being set quietly in research labs around the world.
- During the 2000s, researchers and industry applications began to progressively apply these advances in products such as the following:
  + Self-driving cars.
  + Google Translate.
  + Amazon Echo.
  + AlphaGo.
- Self-driving cars in the 2006 Darpa Grand Challenge used many techniques beyond just deep learning. The top teams (Stanford and Carnegie Mellon University) were able to take advantage of the big improvements of image processing.
- Better image analysis allowed the planning systems in the cars to better choose paths through uncertain terrain and avoid obstacles more safely.
- Other advances in deep learning allowed models to more accurately translate and recognize audio data, driving value in the Google Translate and Amazon Echo line of products. Most recently, we've seen another complex game fall at the master level when the AlphaGo system beat the 9-dan professional Go player Lee Sedol.[8]

2.3 Deep learning applications

Deep learning continues to push the field forward in many domains and on many core machine learning problems. Here are just a few of the benchmark records deep learning has achieved in the last few years:

- Text-to-speech synthesis (Fan et al., Microsoft, Interspeech 2014).
- Language identification (Gonzalez-Dominguez et al., Google, Interspeech 2014).
- Large vocabulary speech recognition (Sak et al., Google, Interspeech 2014).
- Prosody contour prediction (Fernandez et al., IBM, Interspeech 2014).
- Medium vocabulary speech recognition (Geiger et al., Interspeech 2014).
- English-to-French translation (Sutskever et al., Google, NIPS 2014).
- Audio onset detection (Marchi et al., ICASSP 2014).
- Social signal classification (Brueckner & Schulter, ICASSP 2014).
- Arabic handwriting recognition (Bluche et al., DAS 2014).
- TIMIT phoneme recognition (Graves et al., ICASSP 2013).
- Optical character recognition (Breuel et al., ICDAR 2013).
- Image caption generation (Vinyals et al., Google, 2014).
- Video-to-textual description (Donahue et al., 2014).
- Syntactic parsing for natural language processing (Vinyals et al., Google, 2014).
- Photo-real talking heads (Soong and Wang, Microsoft, 2014).[8]

2.4 Futur deep learning applications

Based on these accomplishments, we can easily project deep learning to impact many applications over the next decade. Some of the more impressive demonstrations of applied deep learning include the following:

- Automated image sharpening.
- Automating image upscaling.
- WaveNet: generating human speech that can imitate anyone's voice.
- WaveNet: generating believable classical music.
- Speech reconstruction from silent video.
- Generating fonts.
- Image autofill for missing regions.
- Automated image captioning.
- Turning hand-drawn doodles into stylized artwork.[8]

2.5 Generative modeling

Generative modeling is not a new concept, but the level to which deep networks have taken it has begun to rival human creativity. From generating art to generating music to even writing juice reviews, we see deep learning applied in creative ways every day. Recent variants of generative modeling to note include the following: Inceptionism , Modeling artistic style, Generative Adversarial Networks, Recurrent Neural Networks.[8]

2.5.1 Inceptionism

Inceptionism is a technique in which a trained convolutional network is taken with its layers in reverse order and given an input image coupled with a prior constraint. The images are modified iteratively to enhance the output in a manner that could be described as "hallucinative." In examples for which the input involves images of the sky, we might see fish faces appear in clouds of the output image. This line of research from Google has shown discriminatory neural network models contain considerable information to generate images.[8]

2.5.2 Modeling artistic style

Variants of convolutional networks have shown to learn the style of specific painters and then generate a new image in this style of arbitrary photographs. Imagine having your family photo painted by Vincent van Gogh.[8]

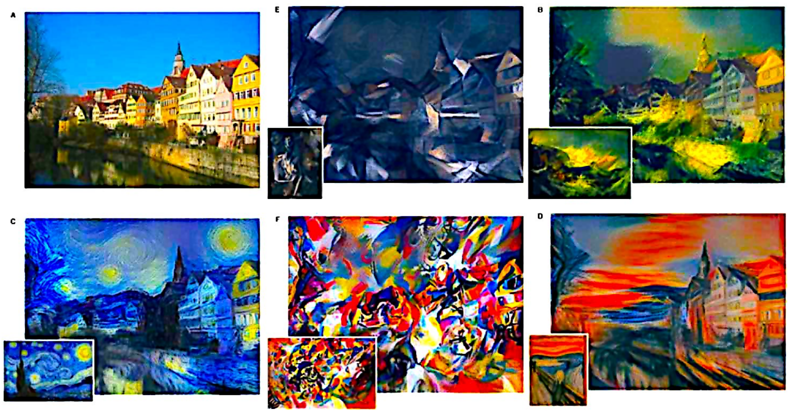


Figure 2.1: Fully connected multilayer feed-forward neural network topology

2.5.3 Generative Adversarial Networks

The generative visual output of a generative adversarial networks can best be described as synthesizing novel images by modeling the distributions of input data seen by the network.[8]

2.5.4 Recurrent Neural Networks

Recurrent Neural Networks have been shown to model sequences of characters and generate new sequences that are lucidly coherent. We take a look at an example of Recurrent Neural Networks in which we generate new lines of Shakespeare by modeling all of Shakespeare's other works.[8]

Another interesting application of Recurrent Neural Networks is the work by Lipton and Elkan in which the network models proper nouns like "Coors Light" and other aspects of beer jargon. The generated beer reviews can be guided with hints (e.g., "give me a 3-star review of a German lager") and are impressive.[8]

2.6 Common Architectural Principles of Deep Networks

Let's extend our understanding of the core components. We'll reexamine the core components again as follows and extend their coverage for the purposes of understanding deep networks:

- Parameters.
- Layers.
- Activation functions.
- Loss functions.
- Optimization methods.
- Hyperparameters.[8]

2.6.1 Parameters

Parameters are related to the x parameter vector in the equation $Ax = b$ in basic machine learning. Parameters in neural networks relate directly to the weights on the connections in the network. We take the dot product of the matrix A and the parameter vector x to get our current output column vector b. The closer our outcome vector b is to the actual values in the training data, the better our model is. We use methods of optimization such as gradient descent to find good values for the parameter vector to minimize loss across our training dataset.[8]

In deep networks, we still have a parameter vector representing the connection in the network model we're trying to optimize. The biggest change in deep networks with respect to parameters is how the layers are connected in the different architectures. In DBNs, we see two parallel sets of feed-forward connections with two separate networks. One network's layers are composed of RBMs (subnetworks in their own right) used to extract features for the other network. The other network in a DBN is a regular feed-forward multilayer neural network, which uses the features extracted from the RBMs–layer network to initialize its weights. This is just one example of many that we'll see over the course of this chapter in how parameters/weights are specialized in different deep network architectures.[8]

2.6.2 Layers

We learned before how input, hidden, and output layers define feed-forward neural networks. In this section we further expanded this architecture with more types of layers and discussed how they relate to specific architectures of deep networks. Layers also can be represented by subnetworks in certain architectures, as well. In the previous section, we used the example of DBNs having layers composed of RBMs. Layers are a fundamental architectural unit in deep networks. In DL4J we customize a layer by changing the type of activation function it uses (or subnetwork type in the case of RBMs). We'll also look at how you can use combinations of layers to achieve a goal (e.g., classification or regression). Finally, we'll also explore how each type of layer requires different hyperparameters (specific to the architecture) to get our network to learn initially. Further hyperparameter tuning can then be beneficial through reducing overfitting.[8]

2.6.3 Activation functions

Depending on the activation function you pick, you will find that some objective functions are more appropriate for different kinds of data (e.g., dense versus sparse). We group these design decisions for network architecture into two main areas across all architectures:

- Hidden layers.
- Output layers.

Hidden layers are concerned with extracting progressively higher-order features from the raw data. Depending on the architecture we're working with, we tend to use certain subsets of layer activation functions.[8]

2.6.4 Loss functions

Loss functions quantify the agreement between the predicted output (or label) and the ground truth output. We use loss functions to determine the penalty for an incorrect classification of an input vector. So far, the following are examples of some of the loss functions:

- Squared loss.
- Logistic loss.
- Hinge loss.
- Negative log likelihood.[8]

2.6.5 Optimization methods

Training a model in machine learning involves finding the best set of values for the parameter vector of the model. We can think of machine learning as an optimization problem in which we minimize the loss function with respect to the parameters of our prediction function (based on our model).[8]

2.6.6 Hyperparameters

Here we define a hyperparameter as any configuration setting that is free to be chosen by the user that might affect performance. Hyperparameters fall into several categories:

- Layer size.
- Magnitude (momentum, learning rate).
- Regularization (dropout, drop connect, L1, L2).
- Activations (and activation function families).
- Weight initialization strategy.
- Loss functions.
- Settings for epochs during training (mini-batch size).
- Normalization scheme for input data (vectorization).[8]

3 | Convolutional Neural Networks

3.1 Introduction to Convolutional Neural Networks (CNNs)

The goal of a CNN is to extract higher-order features in the data via convolutions. They are well suited to object recognition with images and consistently top image classification competitions. They can identify faces, individuals, street signs, and so many kind of different objects. CNNs overlap with text analysis via optical character recognition, but they are also useful when analyzing words as discrete textual units. They're also good at analyzing sound.[11]

The efficacy of CNNs in image recognition is one of the main reasons why the world recognizes the power of deep learning. As Figure 3.1 illustrates, CNNs are good at building position and rotation invariant features from raw image data.[11]

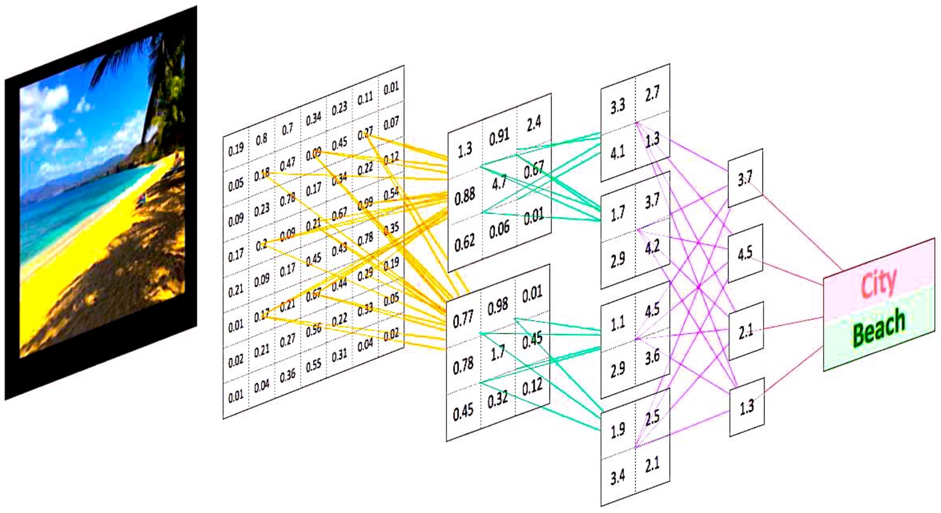


Figure 3.1: CNNs and computer vision

CNNs are powering major projects in computer vision, which has obvious applications for self-driving cars, drones, and treatments for the visually impaired.[2]

CNNs also have been used in other tasks such as natural language translation or generation and sentiment analysis. A convolution is a powerful concept for helping to build a more robust feature space based on a signal.[2]

3.2 Biological Inspiration

The biological inspiration for CNNs is the visual cortex in animals. The cells in the visual cortex are sensitive to small subregions of the input. We call this the visual field (or receptive field). These smaller subregions are tiled together to cover the entire visual field. The cells are well suited to exploit the strong spatially local correlation found in the types of images our brains process, and act as local filters over the input space. There are two classes of cells in this region of the brain. The simple cells activate when they detect edge-like patterns, and the more complex cells activate when they have a larger receptive field and are invariant to the position of the pattern.[8]

3.3 CNNs overview

Feed-forward multilayer neural networks take input as a single one-dimensional vector and transform the data with one or more hidden layers (fully connected). The network then gives a result from the output layer. The issue we run into with traditional multilayer neural networks and image data is that these networks don't scale well with image data as input. The images to train on are only 32 pixels wide by 32 pixels in height with 3 channels of RGB information. This creates 3,072 weights per neuron in the first hidden layer, however, and we'll probably want more than one neuron in that hidden layer. In many cases, we'll want multiple hidden layers in our multilayer neural network, which will multiply those weights, as well.[8]

A normal image could easily be 300 pixels in width by 300 pixels in height with 3 channels of RGB information. This would create 270,000 connection weights per hidden neuron. This shows how quickly a fully connected multilayer network creates a massive number of connections when modeling image data. The structure of image data allows us to change the architecture of a neural network in a way that we can take advantage of this structure. With CNNs, we can arrange the neurons in a three-dimensional structure for which we have the following: [8]

- Width.
- Height.
- Depth.

These attributes of the input match up to an image structure for which we have:

- Image width in pixels.
- Image height in pixels.
- RGB channels as the depth.

We can consider this structure to be a three-dimensional volume of neurons. A significant aspect to how CNNs evolved from previous feed-forward variants is how they achieved computational efficiency with new layer types. We'll cover this arrangement in more depth momentarily.[8]

3.4 CNNs architecture overview

CNNs transform the input data from the input layer through all connected layers into a set of class scores given by the output layer. There are many variations of the CNN architecture, but they are based on the pattern of layers, as demonstrated in Figure 3.2.[8]

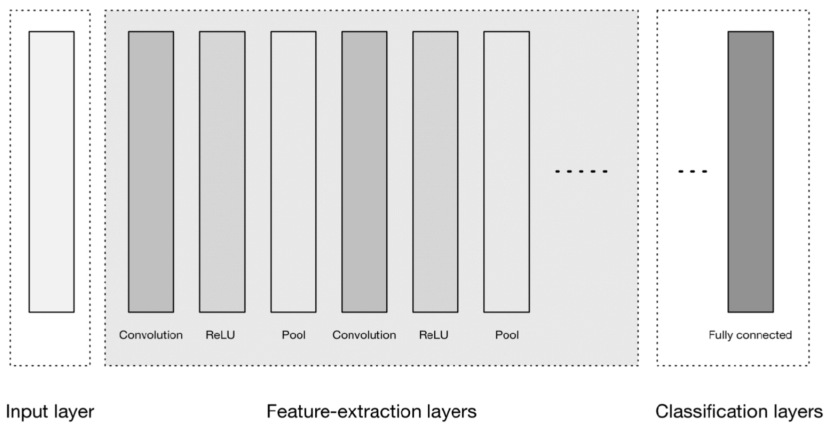


Figure 3.2: High level CNN architecture

3.4.1 Input layer

Input layers are where we load and store the raw input data of the image for processing in the network. This input data specifies the width, height, and number of channels. Typically, the number of channels is three, for the RGB values for each pixel. [8]

3.4.2 Feature-extraction (learning) layers

Convolutional layers are considered the core building blocks of CNN architectures. As Figure 3.3 illustrates, convolutional layers transform the input data by using a patch of locally connecting neurons from the previous layer. The layer will compute a dot product between the region of the neurons in the input layer and the weights to which they are locally connected in the output layer. [8]

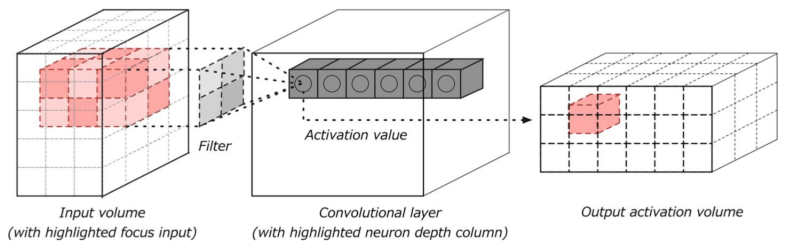


Figure 3.3: Convolutional layer with input and output volumes

The resulting output generally has the same spatial dimensions (or smaller spatial dimensions) but sometimes increases the number of elements in the third dimension of the output (depth dimension). Let's take a closer look at a key concept in these layers, called a convolution.[8]

Convolution is defined as a mathematical operation describing a rule for how to merge two sets of information. It is important in both physics and mathematics and defines a bridge between the space/time domain and the frequency domain through the use of Fourier transforms. It takes input, applies a convolution kernel, and gives us a feature map as output.[8]

The convolution operation, shown in Figure 3.4, is known as the feature detector of a CNN. The input to a convolution can be raw data or a feature map output from another convolution. It is often interpreted as a filter in which the kernel filters input data for certain kinds of information; for example, an edge kernel lets pass through only information from the edge of an image.[8]

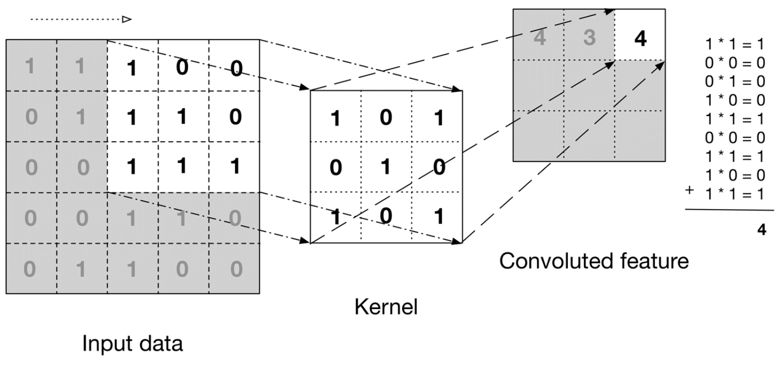


Figure 3.4: The convolution operation

The figure illustrates how the kernel is slid across the input data to produce the convoluted feature (output) data. At each step, the kernel is multiplied by the input data values within its bounds, creating a single entry in the output feature map. In practice the output is large if the feature we're looking for is detected in the input.[8]

We commonly refer to the sets of weights in a convolutional layer as a filter (or kernel). This filter is convolved with the input and the result is a feature map (or activation map).[8]

Convolutional layers perform transformations on the input data volume that are a function of the activations in the input volume and the parameters (weights and biases of the neurons). The activation map for each filter is stacked together along the depth dimension to construct the 3D output volume.[8]

Convolutional layers have parameters for the layer and additional hyperparameters . Gradient descent is used to train the parameters in this layer such that the class scores are consistent with the labels in the training set. Following are the major components of convolutional layers:[8]

- Filters.
- Activation maps.
- Parameter sharing.
- Layer-specific hyperparameters.
- Learned filters and renders.
- Rectified Linear Unit (ReLU) activation functions.[8]

Let's take a look of the specifics of each component.

3.4.2.1 Filters

Filters are a function that has a width and height smaller than the width and height of the input volume.

Filters (e.g., convolutions) are applied across the width and height of the input volume in a sliding window manner, as demonstrated in Figure 3.4. Filters are also applied for every depth of the input volume. We compute the output of the filter by producing the dot product of the filter and the input region. The architecture of CNNs is set up such that the learned filters produce the strongest activation to spatially local input patterns. This means that filters are learned that will activate on patterns (or features) only when the patterns occur in the training data in their respective field. As we move farther along in layers in a CNN, we encounter filters that can recognize nonlinear combinations of features and are increasingly global in how they can detect patterns. High-performing convolutional architectures (which we'll see later in this section) have shown network depth to be an important factor in CNNs.[8]

3.4.2.2 Activation maps

An activation is a numerical result if a neuron decided to let information pass through. This is a function of the inputs to the activation function, the weights on the connections (for the inputs, and the type of activation function itself). When we say the filter "activates," we mean that the filter lets information pass through it from the input volume into the output volume.[8]

We slide each filter across the spatial dimensions (width, height) of the input volume during the forward pass of information through the CNN. This produces a two-dimensional output called an activation map for that specific filter. Figure 3.5 depicts how this activation map relates to our previously introduced concept of a convoluted feature. The activation map on the right in Figure 3.5 is rendered differently to illustrate how convolutional activation maps are commonly rendered in the literature.[8]

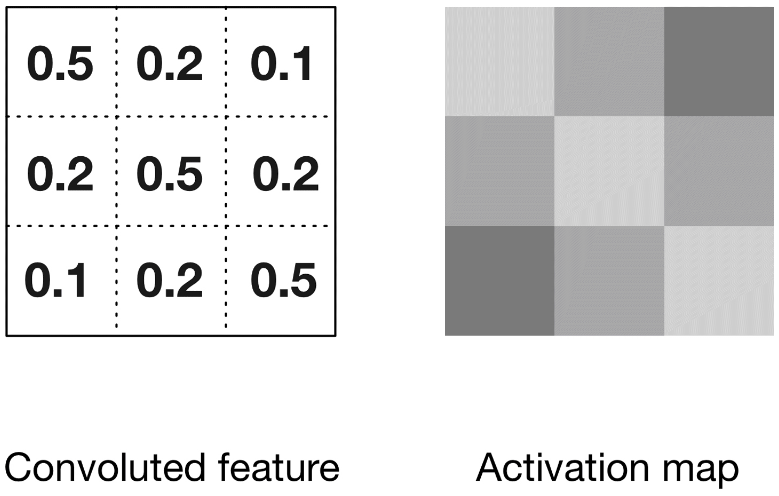


Figure 3.5: Convolution and activation maps

To compute the activation map, we slide the filter across the input volume depth slice. We calculate the dot product between the entries in the filter and the input volume. The filter represents the weights that are being multiplied by the moving window (subset) of input activations. Networks learn filters that activate when they see certain types of features in the input data in a specific spatial position.[8]

We create the three-dimensional output volume for the convolution layer by stacking these activation maps along the depth dimension in the output, as shown in Figure 3.6. The output volume will have entries that we consider the output of a neuron that look at only a small window of the input volume.[8]

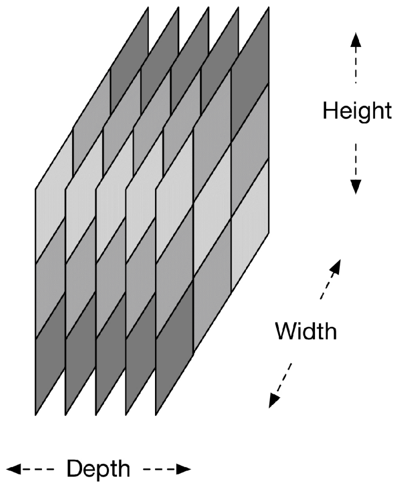


Figure 3.6: Activation volume output of convolutional layer

In some cases, this output will be the result of parameters shared with neurons in the same activation map. Each neuron generating the output volume is connected to only a local region of the input volume, as depicted in Figure 3.7.[8]

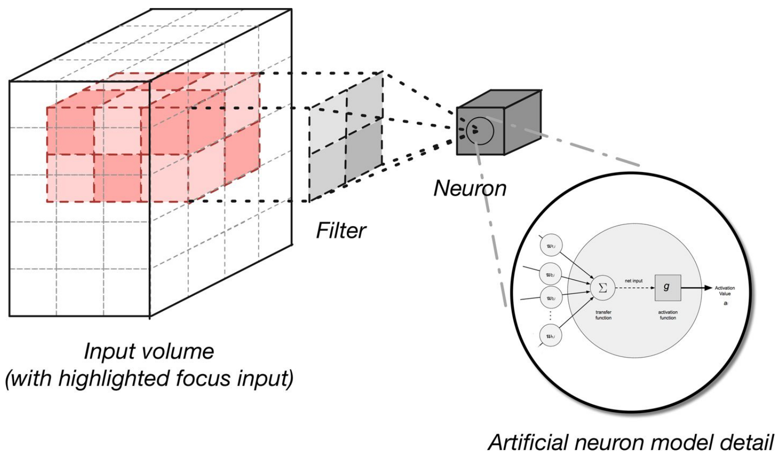

Input volume
(with highlighted focus input)
Filter
Neuron
Artificial neuron model detail

Figure 3.7: Generating and activation output volume

We control local connectivity of this process with the hyperparameter called the receptive field, which controls how much of the width and height of the input volume against which our filter maps.[8]

Filters define a smaller bounded region to generate activation maps from the input volumes. They are connected to only a subset of the input volume through the dynamics of local connectivity described earlier. This allows us to still have quality feature extraction while reducing the number of parameters per layer we need to train. Convolutional layers reduce the parameter count further by using a technique called parameter sharing.[8]

3.4.2.3 Parameter sharing

CNNs use a parameter-sharing scheme to control the total parameter count. This helps training time because we'll use fewer resources to learn the training dataset. To implement parameter sharing in CNNs, we first denote a single two-dimensional slice of depth as a "depth slice." We then constrain the neurons in each depth slice to use the same weights and bias. This gives us significantly fewer parameters (or weights) for a given convolutional layer. We are not able to take advantage of parameter sharing when the input images we're training on have a specific centered structure. We see this effect in faces when we always expect a specific feature to appear in a specific place (for centered faces). In this case we'd probably not use parameter sharing. Parameter sharing is what gives CNNs invariance to translation/position as well.[8]

3.4.2.4 Learned filters and renders

Figure 3.8 presents an example of the learned 96 filters of size $11 \times 11 \times 3$. With the parameter-sharing scheme, we see that detecting a horizontal edge is useful in many places in the image due to the translationally invariant nature of images. This means that we're able to learn the horizontal edge in one place and then not worry about learning it as a feature in all positions in the image.[8]

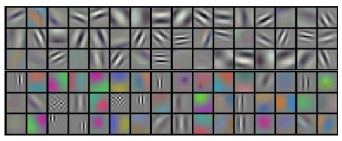


Figure 3.8: Example filters learned by Krizhevsky et al. 96 filters , $11 \times 11 \times 3$

Breaking this up a bit, let's think about a 2D image. If we subdivide an image into four sections, the neural network will then learn position-invariant features of the image. The reason these are position invariant is because of how the network subdivides the data into quadrants. It then learns portions of the image at a time and pools the results. This allows the neural network to learn an overall representation that isn't local to any particular set of features.[8]

3.4.2.5 ReLU activation functions

With CNNs, we often see ReLU layers used. The ReLU layer will apply an element-wise activation function over the input data thresholding for example, $max(0, x)$ at zero, giving us the same dimension output as the input to the layer. Running this function over the input volume will change the pixel values but will not change the spatial dimensions of the input data in the output. ReLU layers do not have parameters nor additional hyperparameters.[8]

3.4.2.6 Layer-specific hyperparameters

Following are the hyperparameters that dictate the spatial arrangement and size of the output volume from a convolutional layer are:

- **Filter (or kernel) size (field size):** every filter is small spatially with respect to the width and height of the filter size. An example of this is how the first convolutional layer might have a $5 \times 5 \times 3$ sized filter. This would mean the filter is 5 pixels wide by 5 pixels tall, with 3 representing the color channels, assuming that the input image was in 3-channel RGB color.

- **Output depth:** we can manually pick the depth of the output volume. The depth hyperparameter controls the neuron count in the convolutional layer that is connected to the same region of the input volume.

- **Stride:** configures how far our sliding filter window will move per application of the filter function. Each time we apply the filter function to the input column, we create a new depth column in the output volume. Lower settings for stride (for example, 1 specifies only a single unit step) will allocate more depth columns in the output volume. This also will yield more heavily overlapping

- receptive fields between the columns, leading to larger output volumes. The opposite is true when we specify higher stride values. These higher stride values give us less overlap and smaller output volumes spatially.

- **Zero-padding:** the last hyperparameter is zero-padding, with which we can control the spatial size of the output volumes. We'd want to do this for cases in which we want to maintain the spatial size of the input volume in the output volume.[8]

3.4.3 Classification layers

We have one or more fully connected layers to take the higher-order features and produce class probabilities or scores. These layers are fully connected to all of the neurons in the previous layer, as their name implies. The output of these layers produces typically a two-dimensional output of the dimensions $[b \times N]$, where b is the number of examples in the mini-batch and N is the number of classes we're interested in scoring. [8]

3.4.4 Pooling layers

Pooling layers are commonly inserted between successive convolutional layers. We want to follow convolutional layers with pooling layers to progressively reduce the spatial size (width and height) of the data representation. Pooling layers reduce the data representation progressively over the network and help control overfitting. The pooling layer operates independently on every depth slice of the input.[8]

The pooling layer uses the $max()$ operation to resize the input data spatially (width, height). This operation is referred to as max pooling. With a 2×2 filter size, the $max()$ operation is taking the largest of four numbers in the filter area. This operation does not affect the depth dimension.[8]

Pooling layers use filters to perform the downsampling process on the input volume. These layers perform downsampling operations along the spatial dimension of the input data. This means that if the input image were 32 pixels wide by 32 pixels tall, the output image would be smaller in width and height (e.g., 16 pixels wide by 16 pixels tall). The most common setup for a pooling layer is to apply 2×2 filters with a stride of 2. This will downsample each depth slice in the input volume by a factor of two on the spatial dimensions (width and height). This downsampling operation will result in 75% of the activations being discarded.[8]

Pooling layers do not have parameters for the layer but do have additional hyperparameters. This layer does not involve parameters, because It is not common to use zero-padding for pooling layers.[8]

3.4.5 Fully connected layers

We use this layer to compute class scores that we'll use as output of the network (e.g., the output layer at the end of the network). The dimensions of the output volume is $[1 \times 1 \times N]$, where N is the number of output classes we're evaluating. In the case of the CIFAR dataset we discussed above, N would be 10 for the 10 classes of objects in the dataset. This layer has a connection between all of its neurons and every neuron in the previous layer. Fully connected layers have the normal parameters for the layer and hyperparameters. Fully connected layers perform transformations on the input data volume that are a function of the activations in the input volume and the parameters (weights and biases of the neurons).[8]

3.5 Other applications of CNNs

Beyond normal two-dimensional image data, we also see CNNs applied to three-dimensional datasets. Here are some examples of these alternative uses:[8]

- MRI data.
- 3D shape data.
- Graph data.
- NLP applications.

The position-invariant nature of CNNs has proven useful in these domains because we're not limited to hand-coding our features to appear in certain spots in the feature vector.[8]

3.6 Popular architectures of CNNs

Following is a list of some of the more popular architectures of CNNs.[8]

- **LeNet**

  One of the earliest successful architectures of CNNs Developed by Yann Lecun Originally used to read digits in images

- **AlexNet**

  Helped popularize CNNs in computer vision Developed by Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton Won the ILSVRC 2012

- **ZF Net**

  Won the ILSVRC 2013 Developed by Matthew Zeiler and Rob Fergus Introduced the visualization concept of the Deconvolutional Network

- **GoogLeNet**

  Won the ILSVRC 2014 Developed by Christian Szegedy and his team at Google Codenamed "Inception," one variation has 22 layers

- **VGGNet**

  Runner-Up in the ILSVRC 2014 Developed by Karen Simonyan and Andrew Zisserman Showed that depth of network was a critical factor in good performance

- **ResNet**

  Trained on very deep networks (up to 1,200 layers) Won first in the ILSVRC 2015 classification task

Related Works

One of the more traditional approaches to image colorization consists of propagating colored user specific scribbles to the whole image. Levin et al. (2004) proposed an optimization based framework for colorizing a grayscale image using this approach. This was done by solving a quadratic cost function derived from differences of intensities between a pixel and its neighboring pixels. This method was improved by Hunang et al. (2005) to prevent the color bleeding over object boundaries. Yatziv and Sapiro (2004) proposed a fast colorization technique using chrominance blending based on weighted geodesic distances. Luan et al. (2007) employed texture similarity for more effective color propagation. Tex ture classification has also been used for cartoon colorization (Qu et al. 2006). Sykora et al. (2009) proposed a flexible colorization tool for hand drawn cartoons based on a graph cut based optimization framework that is easily applicable to various drawing styles. To enable long range propagation for image recolorization or tonal editing, various affinity based methods have also been proposed, such as global optimization with all pair constraints (An and Pellacini 2008; Xu et al. 2009), Radial Basis Function interpolation (Li et al. 2010), manifold learning (Chen et al. 2012), and stochastic modeling of appearance similarities between user specified pixels and other pixels (Xu et al. 2013). However, these methods heavily depend on user input and require trial and error to obtain an acceptable result.[6]

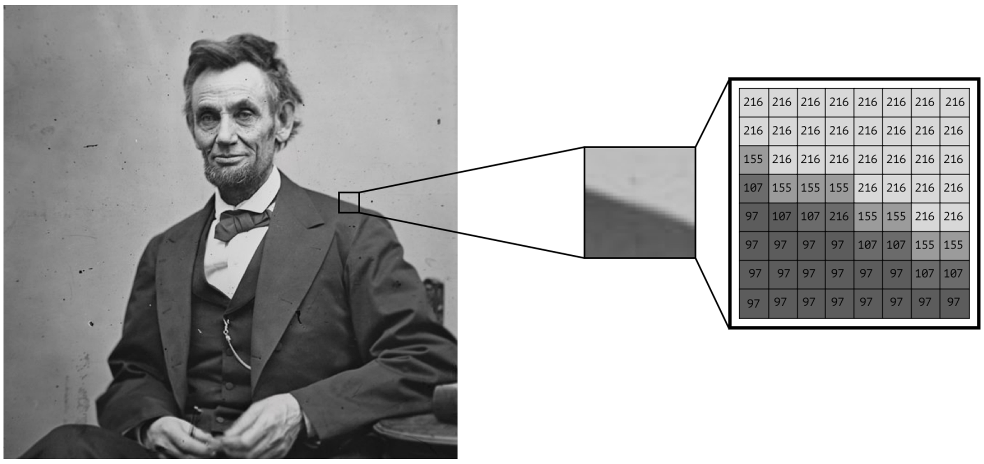
Unlike the scribble based methods that utilize user supplied colors, example based colorization techniques exploit the colors of a reference image that are similar to the input image. For recoloring a color image, color transfer techniques (Reinhard et al. 2001; Tai et al. 2005; Pitié et al. 2007; Wu et al. 2013) are widely used. These compute color statistics in both input and reference images and then establish mapping functions that map the color distribution of a reference image to the input image. Inspired by the color transfer, Welsh et al. (2002) proposed a general technique to colorize grayscale images by matching the luminance and texture information between images. This technique was improved by using a supervised classification scheme that analyzed low level features (Irony et al. 2005). Charpiat et al. (2008) proposed a global optimization framework that deals with multi modality to predict probability of possible colors at each pixel. Gupta et al. (2012) match superpixels between the input image and the reference image using feature matching and space voting to perform the colorization. However, these methods require the user to supply suit able reference images that are similar to the input image, which is a time consuming task. In comparison, our model does not require any user annotation at all.[6]

Instead of requiring the user to provide reference images, Liu et al. (2008) proposed an example based colorization robust to illumination differences between input and reference images that are obtained directly from web search. Its applicability is, however, limited to famous landmarks where exact matches can be found. Chia et al. (2011) extend this to general objects and scenes where exact matches are in general not available by filtering the reference images so that only the most appropriate parts of the images are used. However, they still require the user to input the search query to find the reference images.[6]

4 | Application Design

4.1 Application design overview

Grayscale images can be represented in grids of pixels that their values that corresponds to its brightness range from [0 - 255], from black to white, as shown in Figure 4.1



216	216	216	216	216	216	216	216
216	216	216	216	216	216	216	216
155	216	216	216	216	216	216	216
107	155	155	155	216	216	216	216
97	107	107	216	155	155	216	216
97	97	97	97	107	107	155	155
97	97	97	97	97	97	107	107
97	97	97	97	97	97	97	97

Figure 4.1: The numerical representation of an image

In other hand, colored images consist of three channels, red, green and blue, each channel has values that range from [0 - 255]. Figure 4.2 presents an image in all three channels.

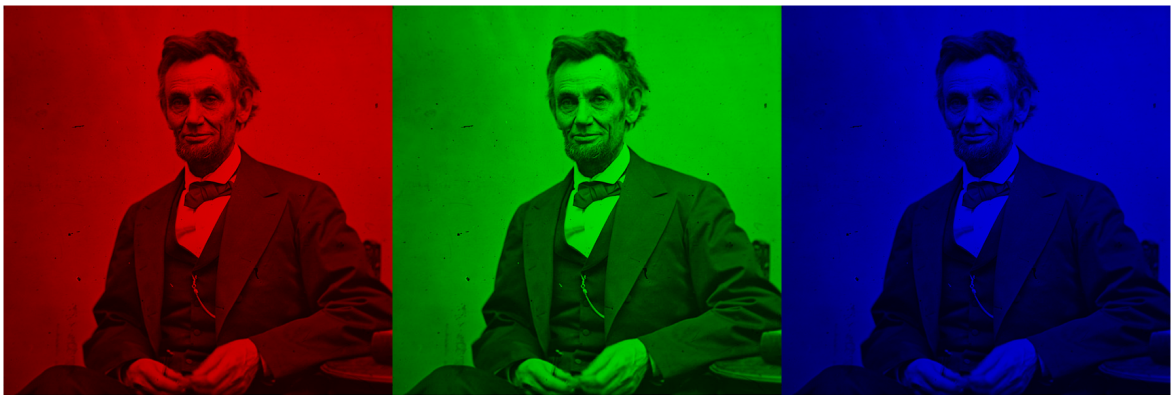


Figure 4.2: The three channels of an RGB image

To acquire the color white, we need to equally distribute all colors. By adding an equal amount of blue and red, it makes the green even bright, see Figure 4.3:

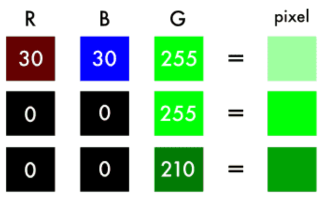


Figure 4.3: The color mixture

Just like grayscale images, each layer in a color image has a value from 0 - 255. If the value is 0 for all color channels, then the image pixel is black.

The neural network creates a relationship between an input value and output value. To be more accurate with our colorization task, the network needs to find the features that link grayscale images with colored ones.

In other word, we are searching for the features that link a grid of grayscale values to the three color grids, see Figure 4.4

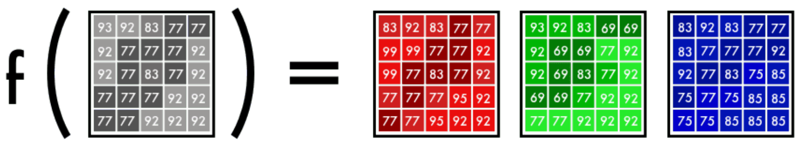


Figure 4.4: A function that takes a grayscale input image, and returns a colored RGB output image

First, we'll use an algorithm to convert the image from RGB to Lab. L stands for lightness, and a and b for the color spectrums green–red and blue–yellow.

As Figure 4.5 shows, a Lab encoded image has one layer for grayscale and have stacked three color layers into two. This means that we can use the very first grayscale image in the final prediction. And so, we have only two channels to predict.

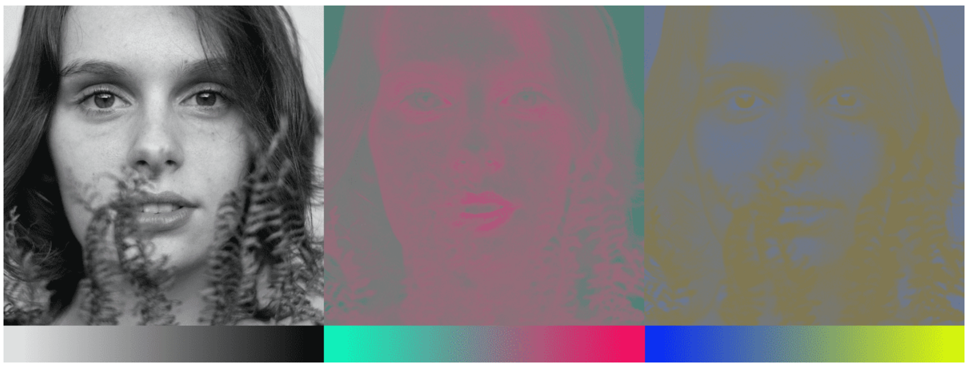


Figure 4.5: The three channels of a LAB image

Science fact - 94% of the cells in our eyes determine brightness. That leaves only 6% of our receptors to act as sensors for colors. As we can see in Figure 4.5, the grayscale image is much sharper than the rest of the two layers. This is why we have to keep the grayscale image in the final prediction. [6]

The final prediction is represented in Figure 4.6. We have a grayscale layer for input, and we want to predict the two color layers, the AB in Lab. To create the final color image we'll include the L(grayscale) image we used for the input, thus, creating a Lab image.[6]

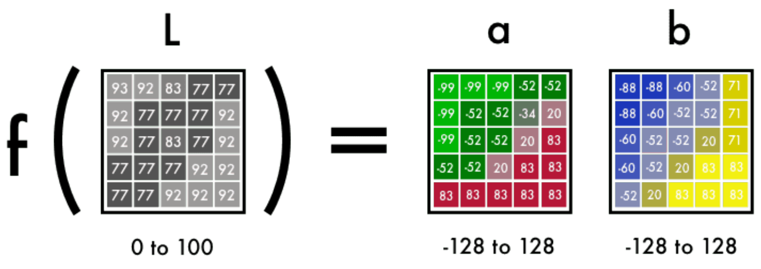


Figure 4.6: The three channels of a LAB image

To turn one layer into two layers, we use convolutional filters. Each filter determines what we see in a picture. They can highlight or remove something to extract information out of the image. The network can either create a whole new image from a filter or stack many filters into one image.[6]

For a convolutional neural network, each filter is automatically adjusted to help arriving to our target outcome.[6]

The input is a grid representing a grayscale image. It outputs two grids with color values (A and B). Between the input and output values, we create filters to link them together, a convolutional neural network.[6]

When we train the network, we use colored images. We convert RGB colors to the Lab color space. The geayscale layer is our input and the two colored layers are the output.[6]

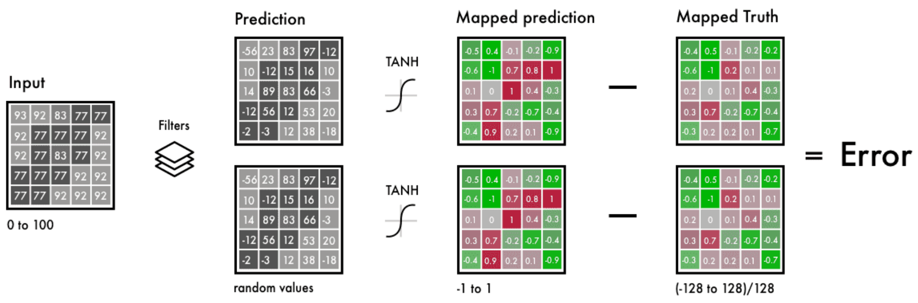


Figure 4.7: The flow in one epoch

In Figure 4.7 we have the grayscale input, the filters used, the mapped prediction adn the mapped truth.

We map the predicted values and the real values within the same interval. This way, we can compare the values. The interval ranges from [-1 to 1]. To get the predicted values we use a Tanh activation function to turn any value between the range of -1 to 1.[6]

The true color values balance from -128 to 128, this is the default interval in the Lab color space. By dividing them by 128, they will fall too within the range of -1 and 1. This enables us to compute the error from our prediction.[6]

After calculating the final error, the network adjusts the filters to reduce the error. The network will run for a certain number of epochs until we have the most minimal error value.[6]

4.2 Training phase

The table 4.1 represents the architecture of the network, along side with the activation function, kernel size, stride and the number of filters.

Type	Activation	Filter	Stride	Output
Conv	ReLU	3×3	2	64
Conv	ReLU	3×3	same	128
Conv	ReLU	3×3	2	128
Conv	ReLU	3×3	same	256
Conv	ReLU	3×3	2	256
Conv	ReLU	3×3	same	512
Conv	ReLU	3×3	2	512
Conv	ReLU	3×3	same	512
Conv	ReLU	3×3	2	512
Conv	ReLU	3×3	same	512
Conv	ReLU	3×3	2	512
Conv	ReLU	3×3	same	256
Conv	ReLU	3×3	same	128
UpSampling	–	–	same	64
Conv	ReLU	3×3	same	32
UpSampling	–	–	same	16
Conv	ReLU	3×3	same	16
UpSampling	–	–	same	6
Conv	Tanh	3×3	same	2

Table 4.1: The network architecture

We start to feed the dataset to the network that is composed of more than 100 images. The algorithm takes the images, turn them into an array of bytes, and then convert them to the LAB color space. The array will be split into two sets, the one for the input and the second for the target output.

We loop the flow for 3000 epochs. We generate randomly the weights, and let them flow in the network, and we calculate the probability with the loss function between our generated weights and the target output, this will determine how well the network is learning. We use the Mean Square Error (MSE) loss function to calculate the loss value.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(output - input)^2 \tag{4.1}$$

For each epoch, we alter the weights to minimize the loss value, for a better predicting result, using an optimizer; we use RMSProp optimizer to adjust the weights.

$$v(w,t) = \gamma v(w,t-1) + (1-\gamma)(\Delta Q_i(w))^2 \tag{4.2}$$

where γ is the forgetting factor, and the parameters are updated as follows:

$$w = w - \frac{\eta}{\sqrt{v(w,t)}}\Delta Q_i(w) \tag{4.3}$$

When the training is finished, we save the weights in '.h' file to use it later in the testing phase. The saved file contains the architecture of the network (the model) and the updated weights.

4.3 Testing phase

In this phase, we load the model to test it with an offset input image. The input image will go through the same processing, except there will be no weight adjusting.

We apply the fixed weights acquired in the training phase on the input image, to determine the predicted output, the AB channels.

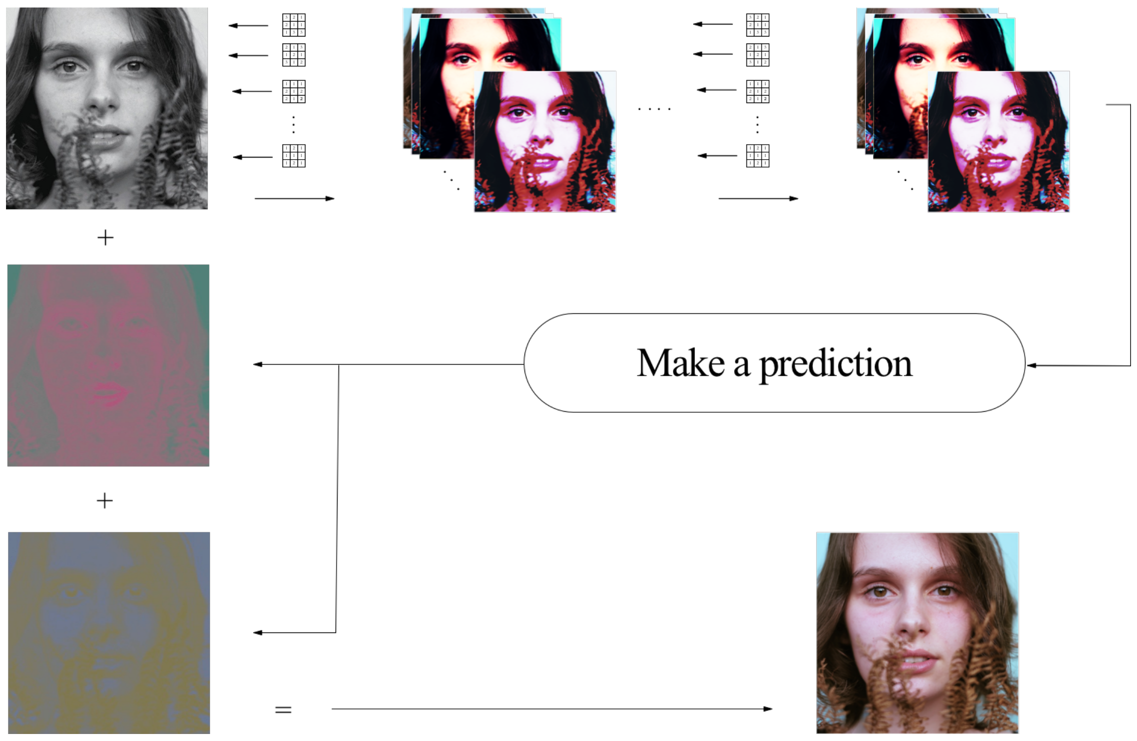


Figure 4.8: The testing process

We then use the input image (grayscale image) as the third channel L to combine them all to acquire the image on LAB color space, to be converted then to the RGB color space, see Figure 4.8.

4.4 The graphical user interface

The application treats different kind of images differently, we divide it into categories to make the prediction more accurate, because the model was trained on one category, and this is due to hardware limitation.

The application supports all different kind of resolution, it can colorize low quality images as it can do the same with high quality images.

The user can either colorize and image and save it, or he can colorize and edit it, the Figure 4.9 represents the use case diagram in which the user can interact with the application.

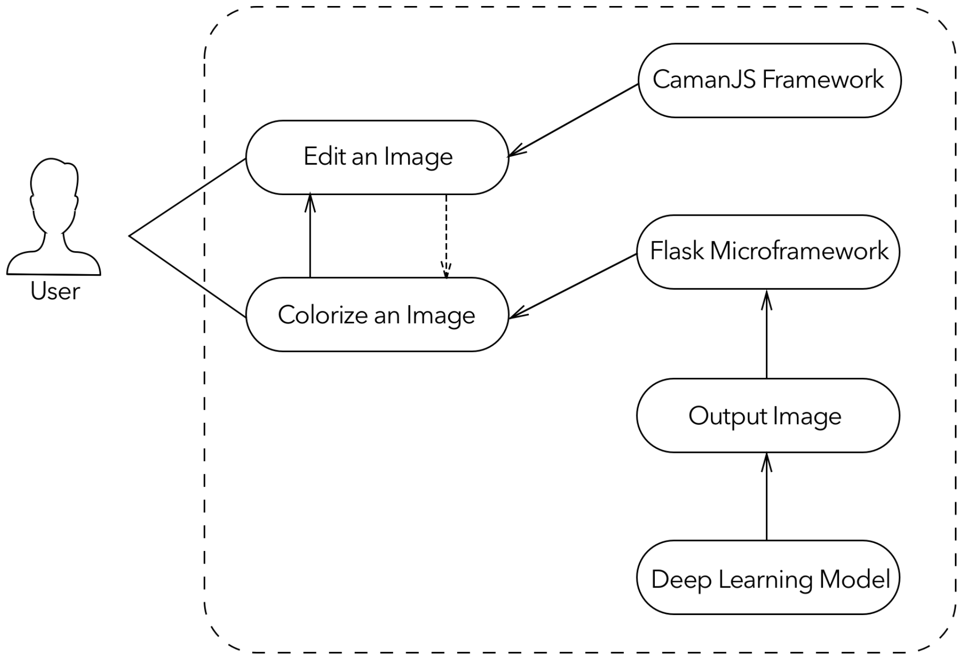


Figure 4.9: The use case diagram.

We provide an elegant interface for the user, starting from the home page, see Figure 4.10.

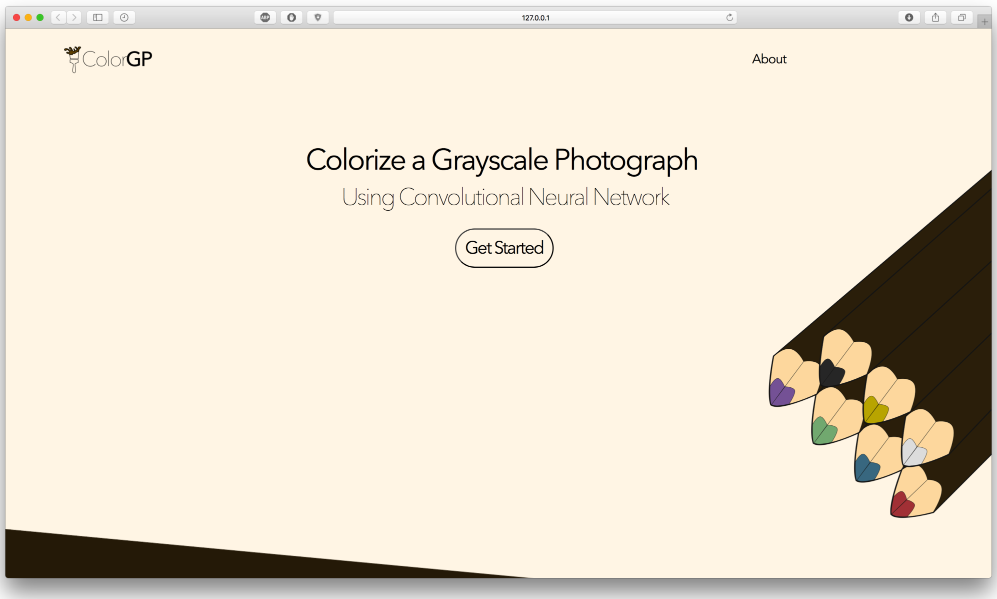

127.0.0.1
ColorGP
About
Colorize a Grayscale Photograph
Using Convolutional Neural Network
Get Started

Figure 4.10: Home page

Jumping up to the category page, in which the user can choose the appropriate category for his image, see Figure 4.11

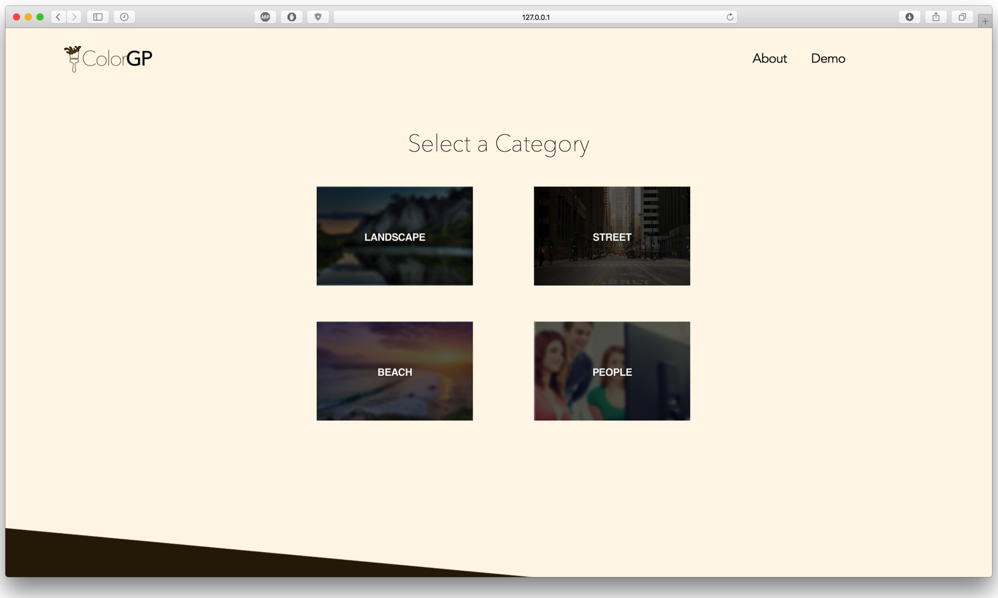


Figure 4.11: Category page

Let us choose a category (say beach), the page contains an indicator of that specific category, the user can upload an image, colorize it, adjust the colors, watch a comparison between the grayscaled and the colored images, and finally he can save it as well to his local disk, see Figure 4.12 & 4.13, 4.14, 4.15, 4.16, 4.17 respectively.

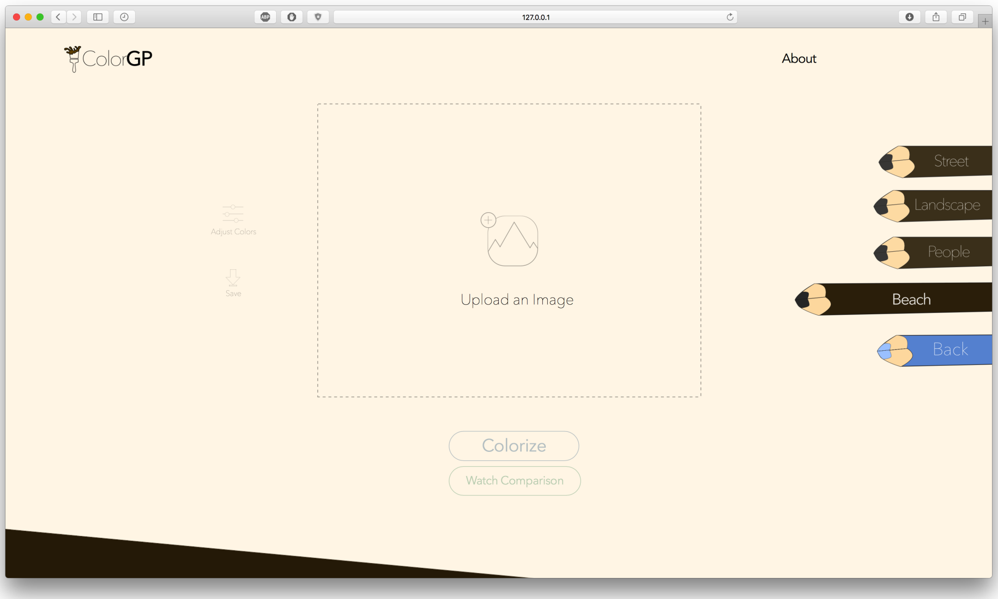


Figure 4.12: Upload an image -1-

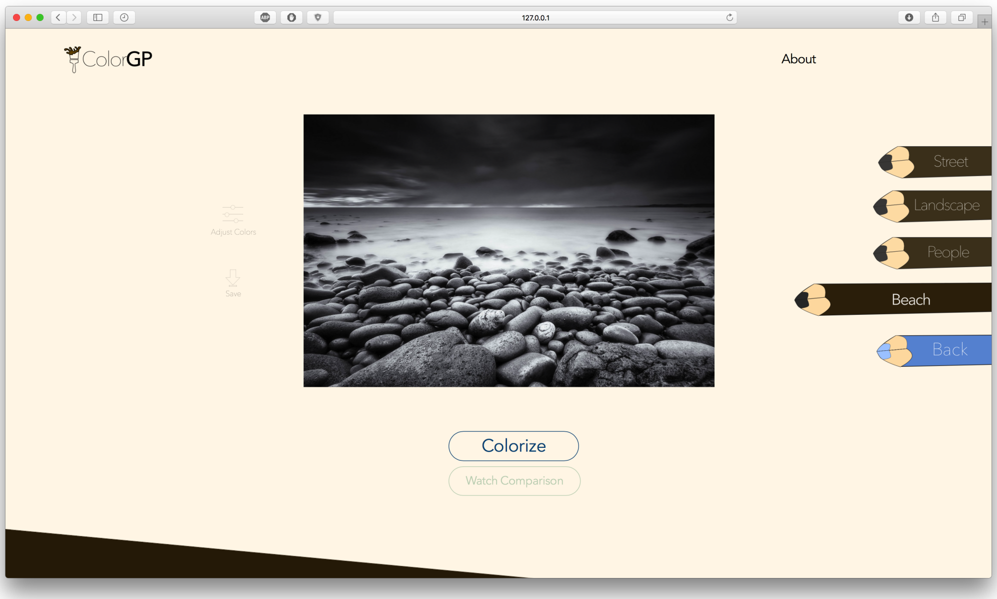


Figure 4.13: Upload an image -2-

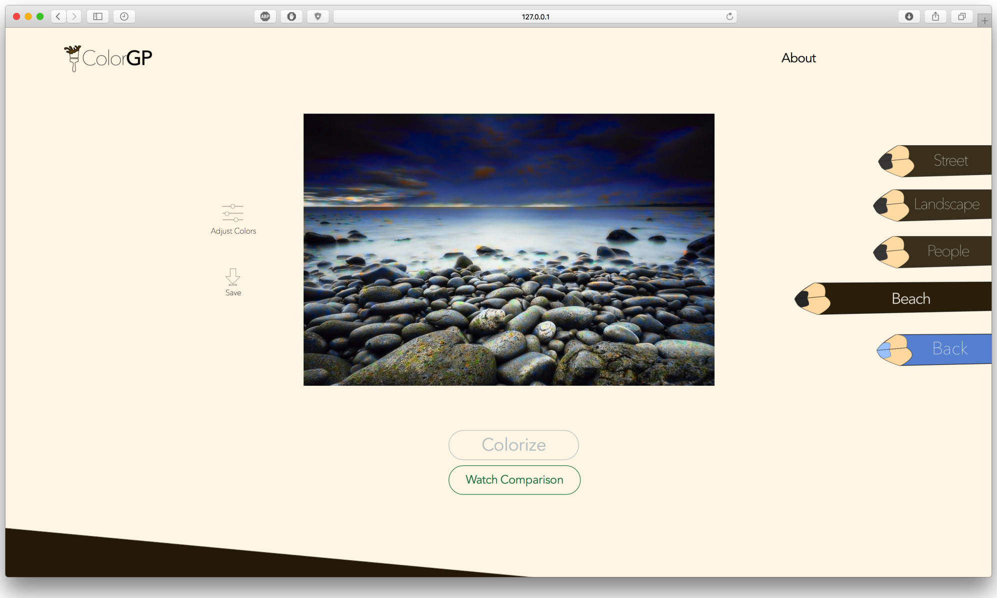
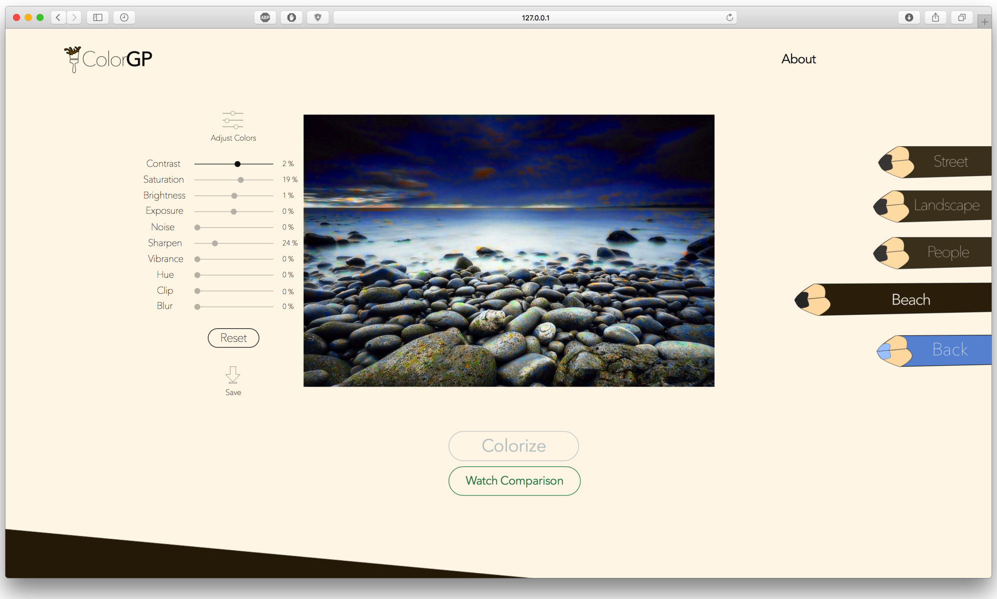


Figure 4.14: Colorize the image



Contrast	2 %
Saturation	19 %
Brightness	1 %
Exposure	0 %
Noise	0 %
Sharpen	24 %
Vibrance	0 %
Hue	0 %
Clip	0 %
Blur	0 %

Figure 4.15: Adjust the image colors

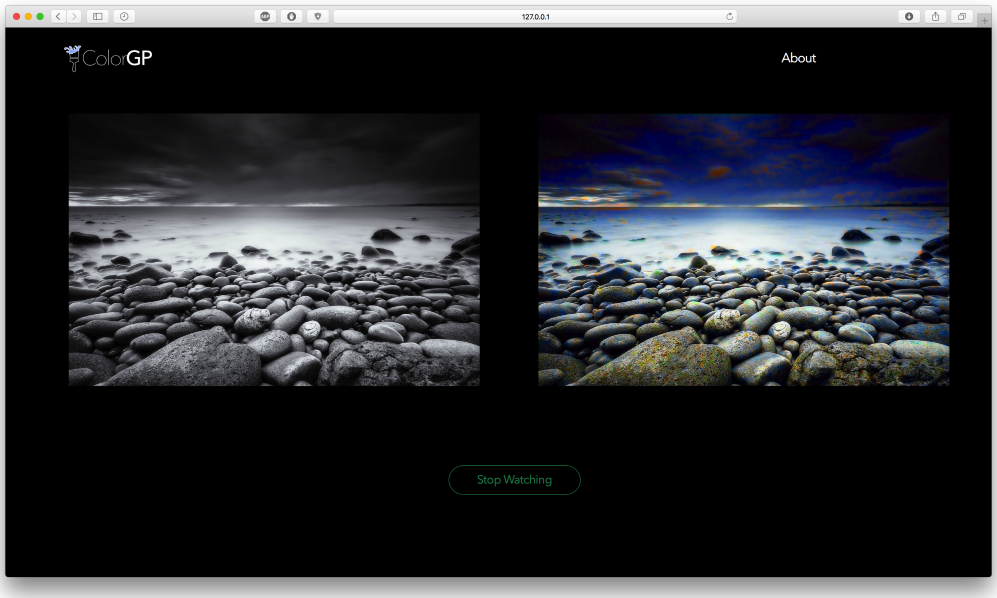


Figure 4.16: Watch the comparison

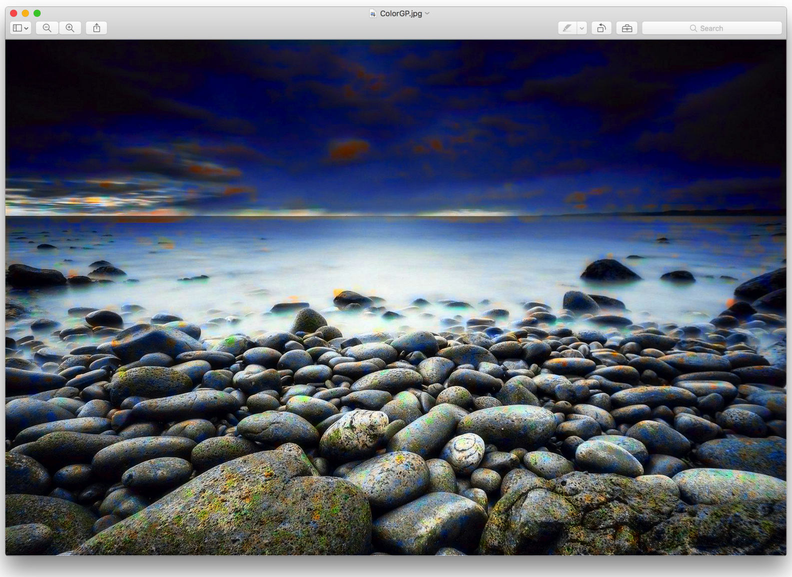


Figure 4.17: Save the image

4.5 Limitations

- Our application uses deep learning to predict the input image colors, it has to be trained on a large dataset to involve all world objects, and this is a heavy computation process, and due to hardware limitation, it was decided to divide it by categories and use a limited dataset for each category.
- Our neural network was trained on a small limited dataset due to hardware limitation, the dataset for each category may not involve every object it contains, so the colorization may not be accurate all the time.
- For each category there's a model that expect a certain kind of object, if the user chose the wrong category for his image, the colorization will not be as expected.

Conclusion

So many researchers take the nature as their source of inspiration, and the artificial neural network is no exception, that admiring human biology and brain functionality provided a powerful tool to change how algorithms work.

By the time, improving the shallow neural network led to a revolutionary approach in which allowing the computer to act like a human even powerful than it ever was, and founding deep learning opened a vast opportunities to think out of the box, and make too many services possible to achieve.

The vision is our daily driver, and images are the way we enjoy life, and adapting deep learning into images made the computer vision an interesting field that so many applications nowadays are relying on, and that's where convolutional neural networks appeared, to have a much more understanding of images, and recognize object.

To achieve our objective to colorize a grayscale photos, we use convolutional neural network to learn from photos and teach the algorithm to understand their content, with training the model with a wide dataset to improve accuracy.

Using a colored images as a dataset, the LAB color space provides a way to split the datset into two sets , one for the input and the other for the target output, to create a relationship between the grayscale images and the output colored images.

The result acquired is decent but needed more accuracy, and that's by using a large dataset that involve every world object, and changing the network architecture.

Used Tools

Hardware

- **Computer Brand :**
  Macbook Pro (Retina, 15-inch, mid 2014)
- **Processor :**
  2.2 GHz Intel Core i7
- **Memory :**
  16 GB 1600 MHz DDR3
- **Startup Disk :**
  256 GB SSD
- **Graphics :**
  Intel Iris Pro 1536 MB

Software

- **Operating System :**
  MacOS Sierra Version 10.12.6
- **Back-end :**
  - **Programming language :**
    Python
  - **Libraries :**
    SKimage, Numpy
  - **API :**
    Tensorflow, Keras
  - **Framework :**
    Flask microframework
- **Front-end :**
  - **Programming language :**
    Javascript
  - **Markup language :**
    HTML5
  - **Style Sheet :**
    CSS3
  - **Libraries :**
    CamanJS
  - **Framework :**
    JQuery

- **IDE, code editor & tested browsers :**
  - **IDE :**
    PyCharm
  - **Editor :**
    Brackets
  - **Browsers :**
    Google Chrome, Safari
- **UI Design :**
  Sketch
- **Document editor :**
  ShareLatex (Latex editor)

Annex

- **Scalars**
  A scalar is a real number and an element of a field used to define a vector space.[8]
- **Vectors**
  For a positive integer n, a vector is an n-tuple, ordered (multi)set or array of n numbers, called elements or scalars.[8]
- **Matrices**
  A matrix is a two-dimensional array for which we have rows and columns.[8]
- **Tensors**
  A tensor is a multidimensional array at the most fundamental level. It is a more general mathematical structure than a vector. We can look at a vector as simply a subclass of tensors.[8]
- **Dot product**
  The dot product takes two vectors of the same length and returns a single number. This is done by matching up the entries in the two vectors, multiplying them, and then summing up the products thus obtained.[8]
- **Element-wise product**
  This operation takes two vectors of the same length and produces a vector of the same length with each corresponding element multiplied together from the two source vectors.[8]
- **Outer product**
  We take each element of a column vector and multiply it by all of the elements in a row vector creating a new row in the resultant matrix.[8]
- **Accuracy**
  Accuracy is the degree of closeness of measurements of a quantity to that quantity's true value.[8]

  $$Accuracy = \frac{(TruePositive + TrueNegative)}{TruePositive + FalsePositive + FalseNegative + TrueNegative} \tag{4.4}$$
- **Precision**
  The degree to which repeated measurements under the same conditions give us the same results is called precision in the context of science and statistics.[8]

  $$Precision = \frac{TruePositive}{(TruePositive + FalsePositive)} \tag{4.5}$$
- **Sigmoid**
  A *sigmoid* function is a machine that converts independent variables of near infinite range into simple probabilities between 0 and 1, and most of its output will be very close to 0 or 1.[8]
- **Tanh**
  Tanh is a hyperbolic trigonometric function . Just as the tangent represents a ratio between the opposite and adjacent sides of a right triangle, *tanh* represents the ratio of the hyperbolic sine to the hyperbolic cosine:

  $$tanh(x) = \frac{sinh(x)}{cosh(x)} \tag{4.6}$$

  Unlike the *sigmoid* function, the normalized range of *tanh* is –1 to 1. The advantage of *tanh* is that it can deal more easily with negative numbers.[8]

- **Optimization**
  The aforementioned process of adjusting weights to produce more and more accurate guesses about the data is known as parameter optimization. You can think of this process as a scientific method. You formulate a hypothesis, test it against reality, and refine or replace that hypothesis again and again to better describe events in the world.[8]

- **Stochastic Gradient Decent (SGD)**
  In SGD, we compute the gradient and parameter vector update after every training sample. This has been shown to speed up learning and also parallelizes. SGD is an approximation of "full batch" gradient descent.[8]

- **AdaGrad**
  Is the square root of the sum of squares of the history of gradient computations. AdaGrad speeds our training in the beginning and slows it appropriately toward convergence, allowing for a smoother training process.[8]

- **AdaDelta**
  AdaDelta is a variant of AdaGrad that keeps only the most recent history rather than accumulating it like AdaGrad does.[8]

- **ADAM**
  ADAM derives learning rates from estimates of first and second moments of the gradients.[8]

- **Dropout**
  Dropout is a mechanism used to improve the training of neural networks by omitting a hidden unit. It also speeds training. Dropout is driven by randomly dropping a neuron so that it will not contribute to the forward pass and backpropagation.[8]

- **Stride**
  Stride configures how far our sliding filter window will move per application of the filter function. Each time we apply the filter function to the input column, we create a new depth column in the output volume. Lower settings for stride will allocate more depth columns in the output volume. This also will yield more heavily overlapping receptive fields between the columns, leading to larger output volumes. The opposite is true when we specify higher stride values. These higher stride values give us less overlap and smaller output volumes spatially.[8]

- **Dataset**
  The dataset represents a mapping of input features to an output vector (e.g., outcomes). The outcomes are specifically for neural network encoding such that any labels that are considered true are 1s. The rest are zeros.[8]

Bibliography

[1] et al. Ahmed Menshawy. *Deep Learning By Example: A hands-on guide to implementing advanced machine learning algorithms and neural networks.* Packt Publishing, 2018. ISBN: 9781788395762.

[2] Koray Kavukcuoglu Clément Farabet Yann LeCun. *Convolutional networks and applications in vision.* IEEE, May 2010, p. 256.

[3] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* Adaptive computation and machine learning. MIT Press, 2016. ISBN: 9780262035613.

[4] K. Gurney. *An Introduction to Neural Networks.* Taylor & Francis, 2003. ISBN: 9780203451519.

[5] S. Simon Haykin. *Neural networks and learning machines.* New York: Prentice Hall/Pearson, 2009.

[6] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. "Let there be Color!: Joint End-to-end Learning of Global and Local Image Priors for Automatic Image Colorization with Simultaneous Classification". In: *ACM Transactions on Graphics (Proc. of SIGGRAPH 2016)* 35.4 (2016).

[7] David Kriesel. *A Brief Introduction to Neural Networks.* first edition. 2007, p. 244.

[8] J. Patterson and A. Gibson. *Deep Learning: A Practitioner's Approach.* first edition. O'Reilly Media,Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2017, p. 507.

[9] Raúl Rojas. *Neural Networks: A Systematic Introduction.* first edition. Berlin, Heidelberg: Springer-Verlag, 1996, p. 453.

[10] Geoffrey Hinton Yann LeCun Yoshua Bengio. *Deep learning.* Vol. 521. Nature Publishing Group, May 2015, p. 436.

[11] Yann LeCun Yoshua Bengio. *Convolutional networks for images, speech, and time series.* Vol. 3361. The handbook of brain theory and neural networks, Apr. 1995, p. 1995.