A Computational Model of Icons Appearance

a dissertation presented
by
Manuel Lagunas

advised
by
Elena Garcés and Diego Gutiérrez

Máster en Modelización e Investigación Matemática, Estadística y Computación

Universidad de Zaragoza
September 2017

A Computational Model of Icons Appearance

Abstract

Nowadays, visual design and interaction are playing an important role in traditional marketing pipelines. Within the context of visual design, the proper selection of icons can be the key of success for engaging and captivating the audience. In this work, we present a method based on a Convolutional Neural Network to obtain icon sets optimized for the properties of style and visual similarity and ease the process of selection to the users. We also obtain a new dataset of icons using an online database where the icons are uploaded by graphic designers in small collections that usually share same semantic meaning and appearance. Since the dataset is labelled by the designers, we cannot be completely sure that there is noise or non-reliable points in the collections. Thus, we use part of the raw data from this dataset to train the CNN, and the rest to collect human-rated information which used for testing. The final goal of the method is to train a Convolutional Neural Network to map each input image to a new Euclidean space where the properties of style and visual similarity are considered. To do so, we train the model using a triplet loss which ensures that images sharing style and visually similar remain closer in the new feature space, while keeps farther icons without similar appearance properties. At the end, we present several results and applications that can be helpful for designers and non-expert users while exploring large collections of icons.

To everyone that made that possible, particularly I would like to thank my wife, my family and, last but not less, my supervisors.

Contents

List of Figures

List of Tables

1 Introduction

Visual communication is one of the most important ways to share and transmit information. In the same way as words are used for verbal communication, symbols or icons are the elements used to convey information in a universal and ubiquitous language. Within the context of visual design, icons are a simple and effective way to captivate users to certain contents: it does not matter how exciting the information you are sharing is if you do not succeed in engaging your observers.

Icons are key elements to structure visual content and make it more appealing and comprehensible. Thus, finding the optimal set of icons is a very delicate task involving semantic, aesthetic and usability criteria. Recent trends aim at automatizing this task and make it more accessible to the general public, either by providing a unified icon representation, such as Google Materials[1], or with online datasets such as The Noun Project[2] with more than one million elements.

While these datasets are undoubtedly useful, they can be hard to explore due to their magnitude. In the Noun Project database, for example, it is common to find small icon collections that usually share similar semantic and style properties, as can be seen in Figure 1.0.1. However, the collections in the dataset do not have unique visual appearance and two or more different collections might share style and semantic properties, making those databases even harder to explore especially when there is no available tool that eases the exploration task or helps with the selection process.

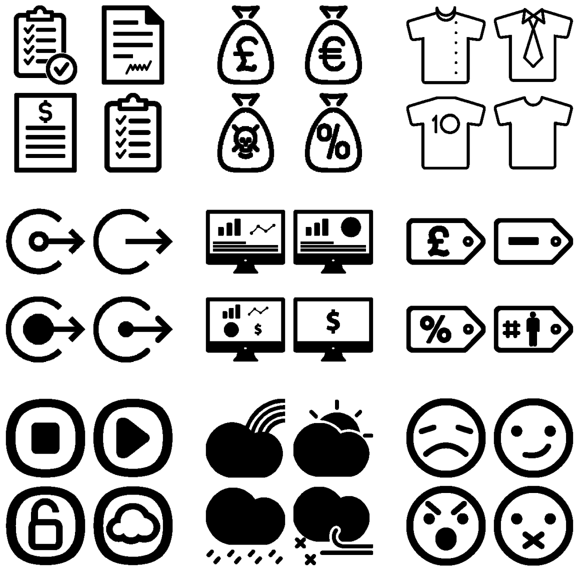


Figure 1.0.1: Example of nine different collections of the dataset. We can notice how style and semantic meaning are preserved for each collections. In reading order we see the collections: "notebook", "faces", "t-shirt", "circle-arrow", "monitor", "label" "round-icon", "weather" and "faces".

An effective set of icons should be made considering the following properties: it should be appropriate for the meaning and be visually appealing. The first property can be leveraged using the **semantic** information of the icons. In general, the icon's designers upload and label small collections that share common semantic properties (i.e. a collection of arrows,

[1]https://material.google.com/
[2]https://thenounproject.com/

houses...), therefore, semantics can be considered in the model just by using the label structure of the dataset. Visual appealing properties like **style** and **distinctiveness** are necessary for an icon set to be effective [28]. Although humans can easily recognize it, style similarity is a difficult concept to define computationally. We will consider style similarity as a function that tries to mimic the human perception of style and that is dependent on the image features such as textures, strokes or lines among others [7]. On the other hand, distinctiveness, or visual similarity, measures the perceptual similarity when comparing icons that have a fixed style. In an ideal icon set, we want icons to be easily distinguishable so that the user does not face problems to discern between them. The Figure 1.0.2 exemplify both concepts.

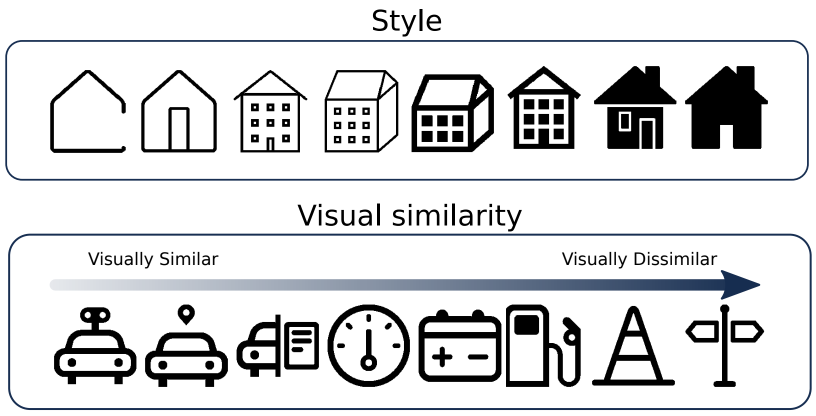


Figure 1.0.2: Graphical example of the style and visual similarity properties. The Style box shows a style interpolation from an icon with thin lines to a black solid icon. The Visual Similarity box shows a set of icons with similar style but varying visual similarity. The icons on the left are more visually similar while the icons on the right are less.

The semantic property is highly attached to the application, and thus, cannot be defined computationally. Therefore, from now on, we assume that each icon is labelled with a keyword that represents its concept properly. We will focus on visual appealing properties which, although hard to define computationally, can be approximated by analyzing the pictorial content of the images.

Problem Statement Ignoring semantics, an icon can be defined by its pictorial properties like strokes, curvature or textures. All these features conform the style of the icon

(Figure 1.0.2 shows examples of different styles). Although we can try to find categorical descriptions of different styles, it is really impractical given the subtle differences between them. Instead, it is more intuitive to find a metric space where the distances between the icons correspond to differences in the perceived style [6]. In such a case, the definition of the style comes from a combination of several elements of the icons whose distances in the new feature space are small. As we can observe in our collections, visual similarity is also a property of style (see, for example, in Figure 1.0.1 the t-shirts or the circle faces and Figure 1.0.2). Then, given our dataset, our goal will be to find a function $D(i, j)$ that measures the properties of style and visual similarity between icons: $D(i, j) = Ds(i, j) + Dv(i, j)$ where (i, j) is a pair of icons, the function $Ds(i, j) \in \mathbb{R}^+$ is the style similarity metric, and $Dv(i, j) \in \mathbb{R}^+$ is the visual similarity metric. The function is considered as a distance, therefore it should return small values for a pair of icons with similar style and visually similar, while it should return large values if the pair of icons have different style and are visually dissimilar. For icons with similar style and visual similarity: $Ds \simeq 0$ and $Dv \simeq 0$, then $D \simeq 0$. For icons with similar style but visually dissimilar the similarity function will be equal to the visual similarity $D = Dv$. Finally, for icons where both properties are very different, the similarity function will also have a high value; $D >> 0$ with $Dv >> 0$ and $Ds >> 0$. The Figure 1.0.3 *(A)* compares two rounded icons consisting on medium-thick lines. The style and visual similarity properties of both are really similar, therefore, the value of the similarity function will be small, $D_A = Ds_A + Dv_A \simeq 0$. Comparing Figure *(B)* we can observe how both have similar style with medium-thick lines $Ds_B \simeq 0$, however, they are visually dissimilar $Dv_B >> 0$, therefore, the value of the similarity function will be larger than it was for the Figure *(A)*, $D_B > D_A$. Figure *(C)* compares the rounded icon with medium-thick lines against a solid icon representing a flamingo. In that case, the style is different $Ds_C >> 0$ and the icons are visually dissimilar $Dv_C >> 0$, thus, the value of thesimilarity function will be the largest, $D_C > D_B > D_A$.

Previous works [5, 6] have proposed perceptual embeddings for predefined icon sets based on crowd-sourcing data. These techniques project the data into a new perceptual space without finding a new representation for the data itself. Thus, they require new user data for each new icon set, not being able to generalize. On the contrary, feature-based metric learning techniques aim to find, first, a representation of the data into a numerical feature space, second, a projection of this feature space where perceptual distances between the elements are maintained. Metric learning approaches generalize to any new data that

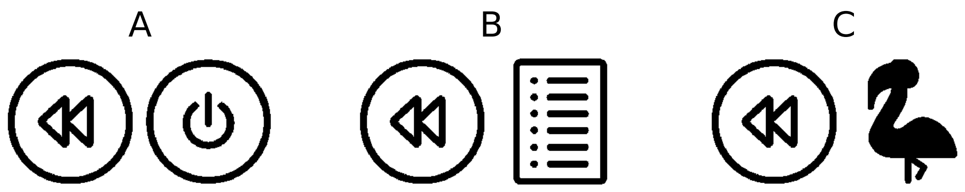


Figure 1.0.3: Examples of similarity between icons. The Figure A shows two icons with similar style and visually similar while Figure B compares icons with similar style yet visually dissimilar and C show two pairs of icons whose style is different and they are also visually dissimilar. In Figure A, we expect a value of $\mathcal{D}_A$ small, icons are visually similar and they share style. On the other hand, Figure B, although icons share style, they are visually dissimilar therefore, it will have a value $\mathcal{D}_B > \mathcal{D}_A$. Figure C $\mathcal{D}$ will have the largest $\mathcal{D}_C > \mathcal{D}_B$ as a result of the differences in style together with the explicit visual dissimilarity of its icons.

can be represented in the selected feature space. These methods are being a hot topic of research [7, 21, 26], particularly, since Convolutional neural networks make hand design of the feature space unnecessary.

In this work, we propose a deep learning-based similarity metric that captures the properties of style and visual similarity for iconography. We greedily gather an icon dataset from the Noun Project online database. The icons in this dataset are organized in collections, where each collection contains a set of icons that share the properties of style and visual similarity. The labels of each collection are given by each designer, so there is no unified and homogeneous label set which we can completely trust. Thus, we might find noise like images that do not share style within the same collections or labels that do no represent the content of a collection accurately. In the machine learning and vision community, this kind of datasets is named *weakly labelled* [30] since additional effort needs to be made to deal with spurious non-reliable points. In our case, the loss function and the data augmentation techniques applied while training help to make the model more robust and mitigate the errors produced by the weakly-labelled data.

In this project, we train a Siamese Neural Network (SNN) that maps the inputs into a new feature space where the properties of style and visual similarity are considered. Training data is sampled from the weakly labelled dataset, while testing data is collected from crowd sourcing experiments. The new feature space aims to keep icons that share style and are visually similar closer, while keeping the ones that do not share those properties farther.

Using the representation of the icons in the new feature space we are able to propose visually appealing icon sets. We assume that icons are labelled by a specific name which implicitly reflects its semantics and do not make any prior assumption about which geometrical or pictorial properties of the images influence in distinctiveness or style.

1.1 Dissertation Background

This Master's dissertation has been developed inside the research group *Graphics and Imaging Lab* within Departamento de Informática e Ingeniería de Sistemas (DIIS) at Escuela de Ingenería y Arquitectura de la Universidad de Zaragoza (EINA). The project has been supervised by Elena Garcés and Diego Gutiérrez. We also had the collaboration of José Tomás Alcalá.

1.2 Dissertation's Overview

The following dissertation is structured in six sections including the introduction. An overview of the project can be seen on Figure 1.2.1

Related Work 2 This chapter comments and explores previous research that is relevant for our work. It includes methods for icon design, previous metrics used to measure the similarity between images by considering the shape or style and using hand-crafted features, methods of kernel learning and finally Siamese Neural Networks for comparing similarity.

Mathematical Background 3 Here, we describe the concept of the Convolutional Neural Network (CNN), the layers that they use and the operations they compute. We introduce the optimization ideas behind the training algorithms of the CNN. Also, we describe meta-algorithms that help the models to converge.

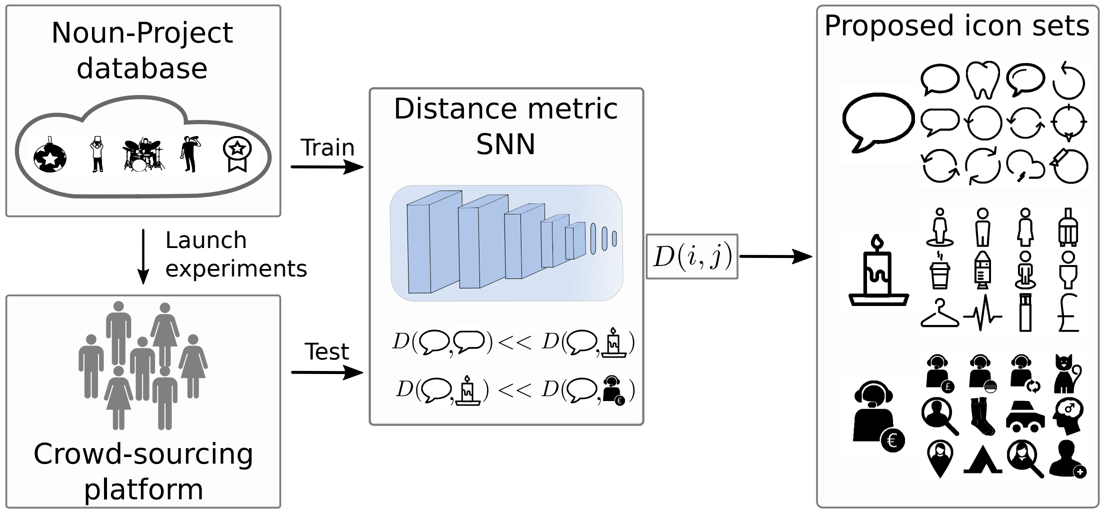


Figure 1.2.1: This figure shows an overview of the project. The leftmost part represents the data gathering part where we used a set of icons collected directly from the Noun Project website. Furthermore, 20% of the data was used to launch experiments in a crowd-sourcing platform, aiming to collect curated data to validate the proposed models. The middle rectangle represents the trained model fed with the previously collected data. Finally, once the model is trained we are able to collect results, visualize them and propose future applications.

Collecting Data 4 In this chapter we explain the method we followed to collect an icons dataset. Furthermore, it describes the process followed to launch experiments into a crowd-sourcing platform and how we obtained human-rated data that will be later used to validate the proposed models.

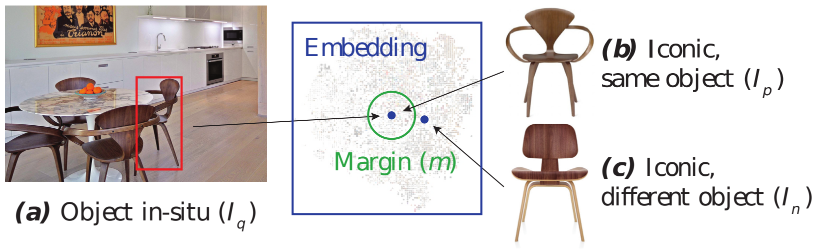
Modelling Visual Appearance of Icons 5 That chapter comments the method followed to solve the problem of finding an optimal set of icons. First, we describe the network type we use, explaining in depth the best architecture we found. Then we define a loss function to train the models and compare the results obtained in some of our models. Finally, we show some results obtained with the model.

Conclusion and Future Work 6 The last chapter summarizes the method, and comment possible future work. Moreover, this chapter includes a detailed diagram with the time spent in each part of the project. Also, there is a personal note explaining the experience I got during the development of the project.

2 Related Work

Icon Design Previous works have focused on generating semantically relevant icons to improve visualizations [28, 29]. In particular, Setlur et al. [28] develop a method for mapping categorical data to icons where they found out that users prefer stylistically similar icons within a set, as opposed to automatic sets that might differ in look-and-feel.

Style Similarity Style similarity metrics have been recently proposed for fonts [21] where they use crowd-sourced data to train models that predict categorical values for new fonts like: ”dramatic” or ”legible” proposing new interfaces instead of the traditional ones in alphabetical order; for infographics [26] by learning the similarity metric on human perception data and using color histograms and histograms-of-gradients (HoG) to characterize the style: in 3D models [19, 20], or interior designs [2] which use crowd-sourced data together with Convolutional Neural Networks to find similar furniture among thousands of images (see Figure 2.0.1). Closer to our goal, the work of Garces et al. [7] utilizes an hand-made



(a) Object in-situ (I_q)
Embedding
Margin (m)
(b) Iconic, same object (I_p)
(c) Iconic, different object (I_n)

Figure 2.0.1: Final goal of the work by Bell and Bala. They aim to find similar furniture among images. Their method receives an input image with a piece of furniture and returns candidate photographs where similar items are present. To do so they rely on a CNN that compute the embedding of the object in the input image and the pieces of furniture in the candidate photograph. If one embedding in the candidate photo has a distance to the reference image below a previously set margin it means that the input item is present in the photograph.

feature vector to measure style similarity for clip art. However, since the descriptors that make up the vector have been manually selected for that particular domain and to solve a style similarity problem, their distance metric does not generalize to our data and is not capable to model the visual similarity property, as we will show in Section 5. Instead, we automatically learn a distance metric that measures both style and visual similarity using a deep siamese neuronal network.

Visual Similarity The problem of visual similarity is related to shape similarity, which is a long-standing problem in computer graphics. Osada et al.[22] propose several descriptors that can be used for 2D and 3D shapes. However, they are expensive computationally, while our metric can be efficiently calculated. Other shape descriptors have been proposed, including Hu-moments [10] that try to find seven moments invariant to rotation, translation and scale; shape context [3] which aims to find a measure of shape similarity by sampling n points on the objects contour and using its relation with the other $n-1$ points as a detailed descriptor; the use of Zernike polynomials as shape descriptors [12] (Zernike moments) thanks to the orthogonality between them that allows them to represent image properties without information overlap; pyramid of arclength descriptors [16], or Fourier descriptors [35]. Kleiman et al. [14] focused on 3D shape similarity, using part-based models, while other works compare shapes using single closed contours [1, 17]. In contrast, our method does not need to explicitly model the geometrical properties of the given image and implicitly considers additional properties such as image abstraction and complexity that

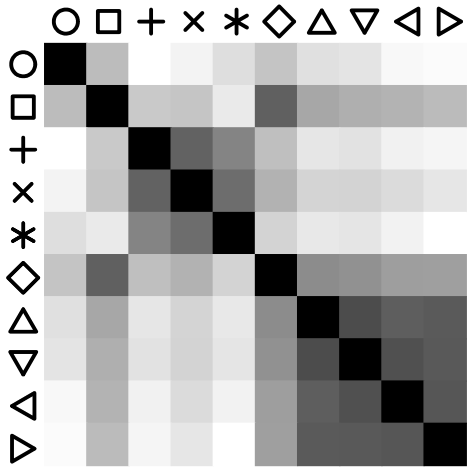


Figure 2.0.2: Example of the kernels computed by Demiralp et al. The figure represents the perceptual distance between the symbols computed on human comparisons. The axes of the figure represent each symbol, the color means the perceptual distance between the compared symbols: black means that symbols are perceptually similar while white means that they are perceptually dissimilar.

are recognized while training the Siamese Neural Network.

Kernel Learning In contrast to the previous works that rely on a feature-based representation of the data, kernel methods aim to obtain directly the similarity matrix for a fixed set of objects, thus such approaches do not generalize to objects outside the chosen set. The work of Laursen et al. [6] proposes an embedding of a small fixed set of icons optimized for comprehensibility and identifiability properties. On the other hand, we propose a general metric trained on a large set of icons and based on deep image representations learned by the Convolutional Neural Networks. The metrics are valid for any candidate, even outside the sample space. Demiralp et al. [5] re-order icon sets to maximize perceptual discriminability, an example can be seen in Figure 2.0.2. Unlike our work, they learn directly the distance kernel over the given set of objects relying on user judgments, on the other hand, we propose to measure perceptual discriminability from image properties and we rely on user judgments to validate the models.

Siamese Neural Networks The name *Siamese Neural Networks (SNN)* refers to a specific architecture that contains two or more identical Convolutional Neural Networks (CNN) that share their parameters. This kind of architecture is really powerful when used in tasks that involve finding relationships between two or more comparable objects (i.e. images [4] or sentences [34]). It has also been shown that SNNs can learn a mapping to a Euclidean space using a triplet loss [23, 27] that compares three inputs - the reference, a positive and a negative sample - making the distance in the new feature space between the positive and the reference really small while ensuring that it is large for the negative sample. In our work, we design our own Siamese Neural Network and follow same principles to design the triplet loss function similar, making sure that icons with same style and visually similar remain close in the new euclidean space learned by the model.

3 Mathematical Background

The following chapter defines the concept of Convolutional Neural Networks in Section 3.1 and explain the operations that are involved in some of the most common layers that are used in these models. Furthermore, the Section 3.2 includes a small introduction about gradient based optimization algorithms that are applied to train Convolutional Neural Networks. The section explains the stochastic gradient descent algorithm, modifications to improve its convergence and regularization together with other methods for stochastic optimization as Adaptative moments or meta-algorithms that might help during the training process.

3.1 Convolutional Neural Networks

Convolutional Neural Networks or CNNs [18] refers to a special type of traditional neural network that is useful when processing data with a grid-like topology like images (2D pixels grid) or time-series (1D grid of samples at regular time intervals) among others. Its name comes from the use of **convolutions** instead of the standard matrix multiplication within its layers. Due to the large number of layers and parameters that these models reach, they are usually named as deep models and its field of research is, generally, called deep learning.

The convolution is an operation that uses two functions x and g to produce a new one s, typically viewed as a modification of one of the originals. It is defined as:

$$s(t) = (x * g)(t) = \int_{-\infty}^{\infty} x(a)g(t-a)da \tag{3.1}$$

Using the terminology of convolutional networks, the function x is referred as the **input**, g is the **kernel**, and the output of the operation s is usually called the **feature map**. Since we are working with a computer we cannot work need to discretize the time t:

$$s(t) = (x * g)(t) = \sum_{a=-\infty}^{\infty} x(a)g(t-a) \tag{3.2}$$

When working with Convolutional Neural Networks is common to use multidimensional arrays as the inputs x and kernels (also called **weights** w). Furthermore, a new term called **bias** (b) is added to the convolution result to get the output s. The bias together with the weights make the trainable parameters $\theta = (w, b)$ of the CNNs. When working with images the space is discretized within the two image dimensions (i, j). The convolution operation slides the kernel across the height and width of the input image and adds the bias term to its result.

$$s(i,j) = (x * w)(i,j) + b = \sum_{m}\sum_{n} x(m,n)w(i-m, j-n) + b \tag{3.3}$$

The parameters θ can be stacked in the convolution, generating then an output with a number of channels equal to the number of stacked parameters (i.e. if we use three kernels and three biases we will get an output with three channels in the form $s = widht \times height \times 3$).

In order to tweak the dimensions of the output and the way of computing the convolution, some variables can be set. Assuming that we are at the iteration t, we can set the values that (m, n) will take at $t+1$ by setting its horizontal and vertical steps, what is called

stride:

$$m_t = m_{t-1} + \underbrace{h_s}_{\text{horizontal stride}} \tag{3.4}$$

$$n_t = n_{t-1} + \underbrace{h_v}_{\text{vertical stride}} \tag{3.5}$$

Where h_s and h_v represent the horizontal and vertical steps that the sliding window takes over the input on each iteration. Finally, the **padding** is a variable that modifies the image dimensions by adding zeros. Let's say we have a 32×32 image, if we set the $padding = 1$ then we get a new image with size 33×33 with the new pixels filled with zeros. This can be useful to preserve image dimensions while applying the convolution. The process followed in a convolutional layer can be visualized in the Figure 3.1.1.

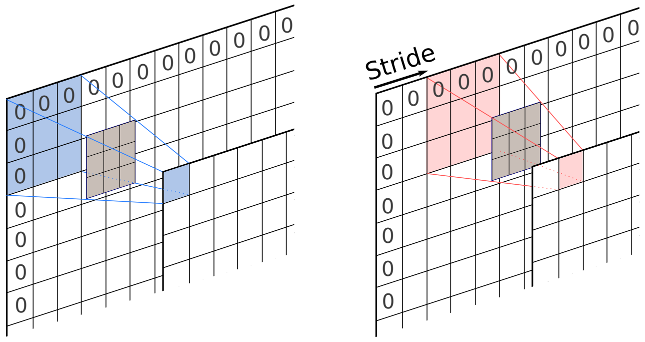


Figure 3.1.1: Example of the convolution over the pixels of an image. The input has only one colour channel, and the convolution is using only one kernel (the matrix with brown colour), thus generating an output with one channel. Padding is set to one pixel and stride is set to two pixels. The bias term in this example is 0. The blue coloured zone and red coloured 3×3 matrix on the input image represent the receptive field multiplied by the kernel and later added to the bias to generate the output pixel, in this case, the blue and red 1×1 matrix coloured on the feature maps.

The convolutional layers are not usually used alone; instead, they are used simultaneously with other functions (or layers) like **rectified linear units (ReLUs)** or **poolings**.

The rectified linear unit or ReLU, is an activation function defined as $f(x) = max(0, x)$ where x is the input. Usually the ReLU activation is preferred over sigmoid or tanh in CNNs due to its reduced likelihood to vanish the gradient (the value of x is constant within the function) and the sparsity produced in the feature vectors (if $x < 0$ then it will be clamped to $x = 0$ promoting sparsity). Furthermore, it has been proved that ReLUs greatly accelerate the convergence of SGD compared to sigmoid or tanh functions [15].

The pooling layers are used to reduce the dimensionality of the network's output. It computes a summary statistic of neighbouring pixels to produce an output. In particular, the max-pooling outputs the maximum pixel value within a rectangular neighbourhood. Other popular pooling functions average the neighbourhood or calculate its L^2 norm. The neighbourhood summarization helps to make the network invariant to small local variations in the inputs.

Finally, after applying convolutions, ReLUs and poolings we obtain a high-level representation of the features in the input data. This new feature space could be directly connected to the output layer, instead, adding a new layers that compute a standard multiplication $s = x \cdot w + b$, where w and b are the parameters, x the input and s the output, could help learning a new linear function in that space and improve the model performance. This standard matrix multiplication layer is called **fully-connected**. In deep models, while the convolutional part provides a meaningful and low dimensional feature space, the fully connected layers learn better functions that represent this space before connecting it to the output.

Creating a CNN A standard Convolutional Neural Network architecture can be seen in Figure 3.1.2. The model has 16 convolutional layers with the depth of its kernels written inside. After the convolution there are 3 fully-connected layers with its number of features written. After every convolution and fully-connected layer there is a ReLU layer (although they are not represented in the figure). Moreover, between some of the convolutions we can find max-pooling layers. The output of the network is a feature map with 1000 feature which can be connected to a new layer regarding the nature of our problem (i.e. softmax function to return the probabilities of the input image to belong to each of the 1000 predefined classes).

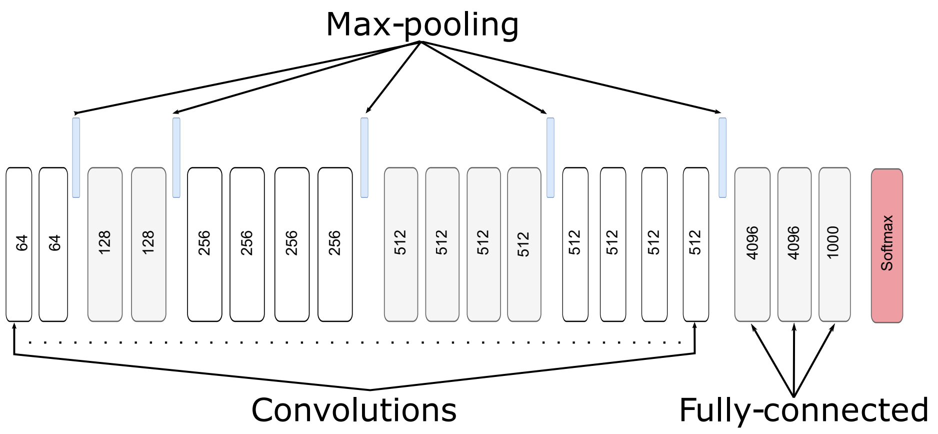


Figure 3.1.2: Example of a traditional Convolutional Neural Network formed by 19 layers with weights: 16 convolutions and 3 fully-connected (FC) layers, All the convolutions and FC layers are followed by ReLU non-linearities. Furthermore, between some of the convolutions there are max-pooling layers.

3.2 Gradient Based Optimization

The performance of deep learning algorithms and models is measured by the **loss function ($\mathcal{L}$)**. The loss function is a measurement of how bad the model is performing a specific task, therefore it will be important to optimize the function searching for its global minimum (finding the global minimum can be an ill-defined problem, we will settle to find a local minimum small enough to work for the problem).

If we denote the loss function as $y = f(x)$, the derivative of the function is $f'(x) = \frac{dy}{dx}$ and it gives us the slope of $f(x)$ at x. Since $f'(x)$ is telling us the slope, we can use it to minimize the function by taking small steps on the opposite sign of the derivative, in other words, descend the slope of the function. When $f'(x) = 0$, the derivative does not provide information about the direction we should take to reduce $f(x)$. In case the function has several inputs $x : \{x_1, ..., x_n\}$ we should use the partial derivative $\frac{\partial f(x)}{\partial x_i}$ to measure how the function is changing when x_i is modified. The vector containing all the partial derivatives is called the gradient $\nabla f(x)$ and its i-th element will be the partial derivative of f with respect to (w.r.t) the variable x_i.

If we compute the derivative of $f(x+\alpha u)$ w.r.t. α, being u the direction of the slope, and evaluating it at $\alpha = 0$, we get that $\frac{\partial f(x+\alpha u)}{\partial \alpha}$ evaluates to $u^T \nabla f(x)$. Therefore to minimize

f we need the direction that decreases the function the fastest.

$$\begin{aligned} min_u \quad & u^T \nabla f(x) \ = & (3.6) \\ min_u \quad & ||u||_2 \cdot ||\nabla f(x)||_2 \cdot \cos(\phi) & (3.7) \end{aligned}$$

The variable ϕ is the angle between the slope direction u and the gradient. Knowing $||u||_2 = 1$, and ignoring other variables that do not depend on u, we can simplify it to $min_u \ \cos(\phi)$. This equation has its minimum at $\phi = 180$, hence u points in the opposite direction of the gradient. Accordingly, we can decrease the value of f by moving from x in the direction of the negative gradient to get a new point x', which will be computed as follows:

$$x_t = x_{t-1} - \epsilon \nabla f(x) \tag{3.8}$$

where ϵ is a positive scalar called the **learning rate (lr)**. It sets how big is the step we give alongside the negative gradient to update the value of x and minimize $f(x)$. If it is really big we might "jump" the global minimum and never reach it, on the other hand, if it is really small it will take us lots of time to find the global minimum. The gradient descent converges when every element of the gradient is zero. In the case of neural networks, we can change the value of x by its parameters.

3.2.1 Stochastic Gradient Descent

Applied to Convolutional Neural Networks (CNN), the loss function $\mathcal{L}$ can be seen as the sum over the training examples of the per-example loss function. Taking the negative conditional log-likelihood as the loss function, and considering as θ the **parameters** (*weights* and *bias*) of the model :

$$\mathcal{L}(\theta) = \mathbb{E}\big[L(x, y, \theta)\big] = \frac{\sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta)}{n} \tag{3.9}$$

where L is the per-example loss function $L(x, y, \theta) = -log \cdot P(y|x, \theta)$. To apply the gradient descent method we need to compute the gradients w.r.t. the parameters θ

$$\nabla_\theta \mathcal{L}(\theta) = \frac{\sum_{i=1}^{n} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)}{n} \tag{3.10}$$

analyzing the sum, we have that its computational cost is $\mathcal{O}(n)$. Considering that the sizes of the training sets for deep models can grow bigger than millions of elements, the time to compute and execute one step of the gradient can be really long. As the gradient is an expectation, we can estimate it using a subset of samples what is called a *minibatch* $\mathbb{B} = \{x^{(1)}, ..., x^{(n')}\}$, being n' not bigger than few hundreds. Now we can fit a training set with millions of items by using updates that have been calculated on few hundred examples, having an estimation from the samples in $\mathbb{B}$:

$$\nabla_\theta \mathcal{L}^{\mathbb{B}}(\theta) = \frac{\nabla_\theta \sum_{i=1}^{n'} L(x^{(i)}, y^{(i)}, \theta)}{n'} \tag{3.11}$$

and applying it to the gradient descent algorithm we get that the parameters update can be computed as:

$$\theta_t = \theta_{t-1} - \epsilon \nabla_\theta \mathcal{L}^{\mathbb{B}}(\theta_{t-1}) \tag{3.12}$$

This algorithm which uses the estimation of the gradient $\mathcal{G}$ in a subset of data, or *mini-batch*, $\mathbb{B}$ is called **Stochastic Gradient Descent (SGD)** and it is widely used to train deep models. It is an optimization algorithm that does not guarantee to reach a global minimum, instead, it often finds great values of the loss function not requiring really long times, which makes it useful.

Regularization

Traditional machine learning algorithms have been using regularization techniques before deep learning appeared. Most of those methods are aim to limit the freedom of the model parameters by adding a parameter norm penalization $\mathcal{R}(\theta)$ to the loss function $\mathcal{L}$

$$\mathcal{L}' = \mathcal{L}(x, y, \theta) + \lambda \mathcal{R}(\theta) \tag{3.13}$$

with $\lambda \geq 0$ a hyper-parameter that sets the weight of the regularization in the loss function. In the case of neural networks, the norm penalty ($\mathcal{R}$) is only applied to the *weights (w)* while leaving the *bias* with no regularization. Also, to introduce the regularization term in the bias could lead to under-fitting. It would be desirable to tune each regularization coefficient λ_i over each layer i of the neural network but it can be computationally expensive, hence it is reasonable to use the same regularization coefficient λ over each layer to constrain the search space.

The L^2 parameter norm penalty is one of the most common ones when working with neural networks, it is known as *weight decay* (also called ridge regression). It adds a regularization term $\mathcal{R}(\theta) = \frac{||w||_2^2}{2}$. If we include the regularization term $\mathcal{R}$ in the new loss function $\mathcal{L}'$ and assume that there is no bias parameter $\theta = w$ we get:

$$\mathcal{L}'(x, y, w) = \mathcal{L}(x, y, w) + \lambda \frac{||w||_2^2}{2} \tag{3.14}$$

Therefore, the gradient is computed as

$$\nabla_w \mathcal{L}'(x, y, w) = \nabla_w \mathcal{L}(x, y, w) + \lambda w \tag{3.15}$$

Now, the parameters update in the gradient descent algorithm is

$$w_t = w_{t-1} - \epsilon \lambda w_{t-1} - \nabla_w \mathcal{L}(x, y, w_{t-1}) \tag{3.16}$$

The regularization term modifies the gradient descent step and shrinks the coefficients towards zero, thus ensuring that we find a minimum with smaller parameters and avoiding over-fitting.

Momentum

Some methods have been developed to speed up the learning focusing in particular scenarios where the process could get stuck. The method of **momentum** [24] stores an exponentially decaying moving average of the previous gradients and it makes sure that it continues moving towards their direction helping to the training convergence.

The momentum algorithm introduces the variable v, or velocity, which accumulates the direction at which parameters are moving through the search space. We denote v' the velocity term we are going to compute while v is the velocity already computed in the previous iteration.

$$v_t = \alpha v_{t-1} - \epsilon \nabla_\theta \mathcal{L}(\theta_{t-1}) \tag{3.17}$$

$$\theta_t = \theta_{t-1} + v_t \tag{3.18}$$

The hyperparameter α is a real number between 0 and 1, and it defines how fast the previous gradients exponentially decay.

Nesterov Momentum Recently, Sutskever et al. [32], inspired by Nesterov's accelerated gradient method, introduced a new feature to the original momentum algorithm. They propose to evaluate the gradient after the current velocity (v) is applied.

$$v_t = \alpha v_{t-1} - \epsilon \nabla_\theta \left(\frac{\sum_{i=1}^{n} L\big(x^{(i)}, y^{(i)}, \theta_{t-1} + \alpha v_{t-1}\big)}{n} \right) \tag{3.19}$$

$$\theta_t = \theta_{t-1} + v_t \tag{3.20}$$

The momentum method modifies the direction that the original gradient descent algorithm would take to the optimal value by avoiding fluctuations in the path.

We can add both momentum and weight decay as constraints to our loss function $\mathcal{L}$. The new parameters update after the gradient adding both constraints would be:

$$\theta_t = \theta_{t-1} + \underbrace{\alpha v_t}_{\text{Momentum}} - \underbrace{\epsilon \lambda \theta_{t-1}}_{\text{Weight decay}} - \underbrace{\epsilon \nabla_\theta \mathcal{L}(\theta_{t-1})}_{\text{Gradient update}} \tag{3.21}$$

where α is the momentum parameter, λ is the weight decay coefficient and ϵ is the learning rate.

3.2.2 Adaptative Moments: ADAM

The learning rate (ϵ) is a hyper-parameter that we have to tune during training to achieve a better training convergence. The Adaptative Moment Estimation (ADAM) [13, 25] method tries to update the learning rate for each parameter. It stores an exponentially decaying average of squared gradients t_t and an exponentially decaying average of past gradients m_t which are estimates of the first and second moment of the gradients (mean and variance). The method requires also two hyper-parameters β_1, $\beta_2 \in [0, 1)$ that control the exponential decay rates of the moving averages t_t and m_t.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta \mathcal{L}(\theta_{t-1}) \\ t_t &= \beta_2 t_{t-1} + (1 - \beta_2)\nabla_\theta \mathcal{L}(\theta_{t-1})^2 \end{aligned} \tag{3.22}$$

The parameters t_t and m_t are initialized as vectors of zeros. Authors perceived that this initialization leads to moment estimations that are biased towards zero during initialization (first time-steps) and when the hyper-parameters β_1, β_2 are close to 1. They propose a bias correction in the estimation.

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{t}_t &= \frac{t_t}{1 - \beta_2^t} \end{aligned} \tag{3.23}$$

Then they use this estimation to update the learning rule:

$$\theta_t = \theta_t - 1 - \frac{\epsilon m_t}{\sqrt{\hat{t}_t} + \gamma} \tag{3.24}$$

where the proposed values of the hyper-parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\gamma = 10^{-8}$. They also empirically show that ADAM is able to achieve faster convergence that other adaptative-learning methods. While this method is still based on SGD, adding constraints like weight decay or momentum does not improve the performance. A comparison[1] of the previously commented methods can be seen in the Figure 3.2.1.

[1] Code: `https://gist.github.com/mlagunas/920d00d0436e12587ed9b533b873ce07`

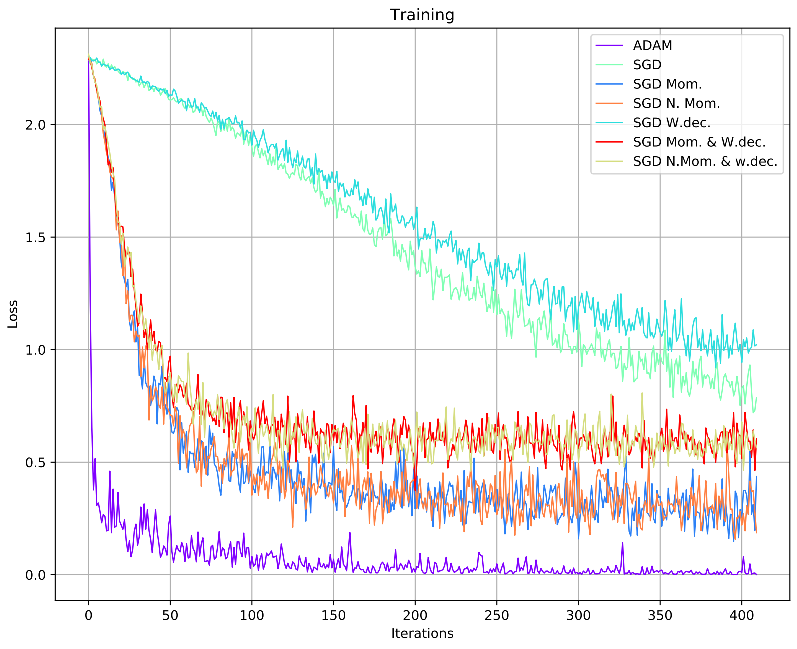


Figure 3.2.1: The graph shows a comparison in convergence time (counting number of iterations) of different optimization algorithms using the same standard neural network with just a hidden layer, training it to classify digits.

3.2.3 Meta-Algorithms to improve convergence

Dropout Dropout [31] can be seen as a method that makes bagging practical for ensembles of exponentially many neural networks (or Convolutional Neural Networks) that are sharing their parameters. It helps regularizing a large number of different models and it is cheap in computational terms.

The Dropout method trains an ensemble of the models that can be built by removing connections between layers that are not in the output. The model has the same architecture but some of the connections between layers are removed, furthermore, these models share their parameters. The main goal of this method is to approximate the bagging process. To include Dropout during training we need to sample a binary mask, that will be applied to the connections between the hidden units. If the mask is zero the connection will be *stopped* and its value will be 0, if the mask is one the connection continue and propagate its value through the model. The probability of sampling a value of one is a hyper-parameter set

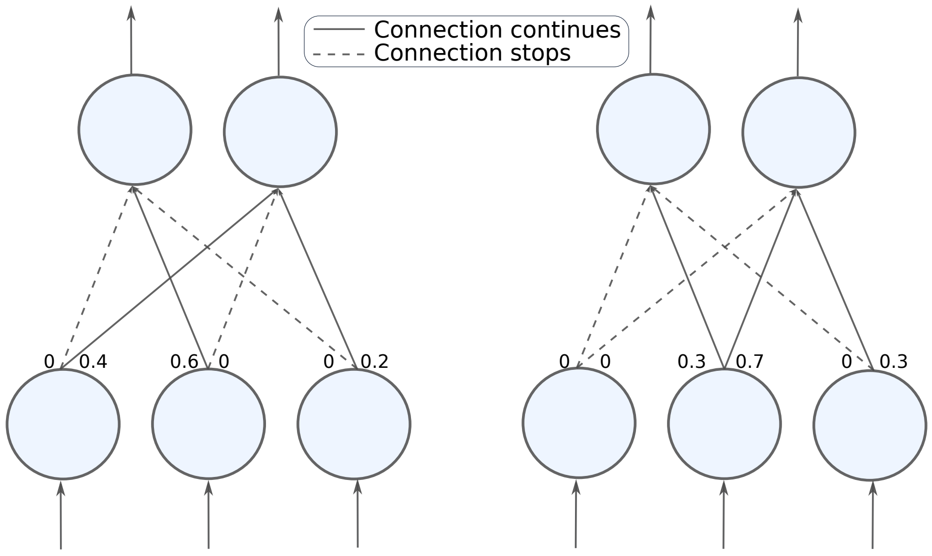


Figure 3.2.2: Graphical example of Dropout method in a simple neural network. The Dropout value is fixed to 0.5, therefore half of the connections are stopped. The value of each connection is represented next to it, being zero when they are stopped.

before training starts. The way to compute the error, gradients and the learning updates stay as usual. During testing, Dropout is not applied, therefore the mask has a value of 1 in all its elements. Unlike bagging, Dropout shares the parameters between models which make possible to represent an exponential number in a contained amount of memory. At the end, it is just one model with different connections being zero at each step of the learning process. The Figure 3.2.2 shows an example of how dropout works for a simple neural network.

Batch-Normalization Due to the complexity training deep architectures, some meta-algorithms and heuristics have appeared that aim to improve the optimization process even though they are not actually optimization algorithms at all. Deep neural networks are formed by several layers with parameters that are updated using its gradient. When a layer is updated, we make the assumption that the other layers do not change, but in practice, all the layers are updated *simultaneously* which could yield into errors and unexpected results.

Batch normalization [11] defines a new way of re-parametrizing the deep models to ease the problem of simultaneous updates over the layers. Considering the matrix A as the minibatch of activations (outputs) of the layer to normalize, and A' the activations already normalized:

$$A' = \frac{A - \mu}{\sigma} \tag{3.25}$$

where μ is the mean and σ is the standard deviation of each unit. The variables μ and σ are calculated at training time:

$$\mu = \frac{1}{m} \sum_i A_i \tag{3.26}$$

$$\sigma = \sqrt{\rho + \frac{1}{m} \sum_i (A_i - \mu)_i^2} \tag{3.27}$$

the constant $\rho = 10^{-8}$ is a small value to avoid encountering undefined gradients like $(\sqrt{z})' = 1/2\sqrt{z}$ when $z = 0$. At test time μ and σ are set as the previous values calculated during training time, we do not re-evaluate again over the whole minibatch. However, normalizing the layers can reduce the diversity of the network and its performance. In order to keep it the activation $A = \gamma A' + \beta$ where γ and β are new parameters that allow different learning dynamics. The input has been normalized to zero mean and unit variance A' but we are letting it learn any other distribution that could be better. Even if that method does not work, the network could learn during the gradient descent $\gamma = \sigma$ and $\beta = \mu$, which means that it can unroll those changes.

4 Collecting Data

We obtain our icon dataset from the *Noun Project* website[1], which contains thousands of black and white icons uploaded by graphic designers. Using the provided API we greedily downloaded a total of 26027 different icons, grouped in 1212 collections or classes each one sharing a semantic label decided by the author (see Figure 4.0.1 for a few examples). Since we cannot assume that the labelling was done precisely and some classes might contain errors or be noisy, we say that our dataset was weakly labelled [30]. Each icon belongs to just one class and most of the icons per class share also similar style and semantic properties (i.e. the author uploads icons of houses he made with similar textures, strokes, etc.) their visual similarity is also really high. As we show later, we leverage this information to train our similarity metric. For our experiments, by means of stratified sampling, we split the dataset into three subsets: training (70%), validation (10%), and test (20%). This method samples the subsets from each class, ensuring that all of them contain elements in the three

[1]https://thenounproject.com/

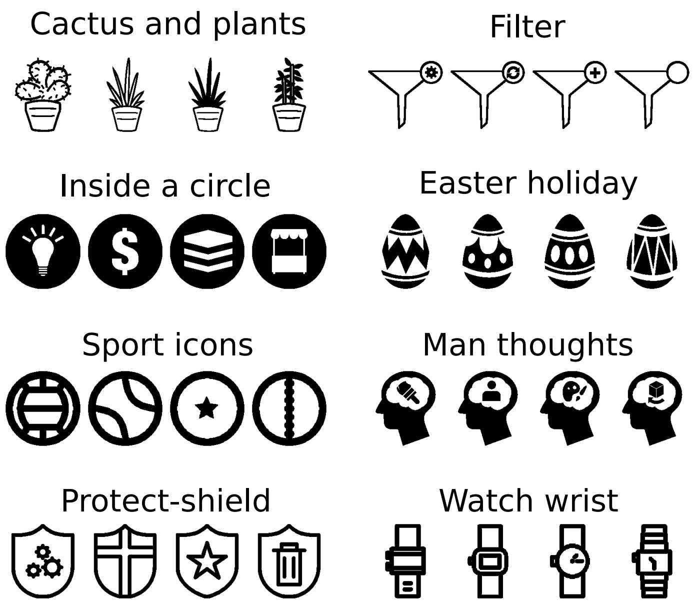


Figure 4.0.1: Some representative icons belonging to a few of the 1212 collections that the dataset contains. The name of each collection is written on the top of each set of icons.

categories.

4.1 Collecting Curated Data

The dataset we collected is weakly-labelled, meaning that few classes might contain errors. Although we can train the model parameters with this data, we cannot use it at testing time. The few errors that the classes may contain could make the results of our metric not accurate enough, yielding to problems like not detecting if the model has overfitting or not allowing for a fair comparison with other methods. That's why we collect valid data on the perception of icon's similarity that will be used to test the accuracy of the deep models that are proposed. We used *Amazon Mechanical Turk (MTurk)*[2] - a crowd-sourcing platform that connects human raters with the tasks published by the requesters - to launch the tests (or Human Intelligence Task, HIT) and collect valid data for testing. Similar to previous works [2, 7, 19], we gathered data in the form of *relative comparisons*, since they

[2]https://www.mturk.com/mturk/welcome

are more robust and easier for human raters than likert ratings [5]. The structure of each HIT consisted on:

- First, a clear description of the task that the rater had to perform, also, we show some examples and thoroughly explaining to the user how to perform the task using the examples. At the end of this part, we ask the user for anonymous information like gender and previous experience in related topics to user interfaces interaction and design.
- After the user reads the description, it begins the *training phase* where a small set of four relative comparisons is shown, each one showing three icons, namely the Reference, icon A, and icon B. The set of relative comparisons in the training phase was manually picked up and the rater was asked: "*Is the Reference more similar to icon A, or to icon B?*", in case the user answers incorrectly a guidance message appears pointing out how to perform the task and letting the user try again.
- After all the questions in the training phase has been answered correctly the user proceeds to the *test phase*. The question made to the human rater is the same as in the training phase. The rater has to answer a total of 60 comparisons - where seven questions belong to control questions manually selected with an obvious answer - sampled from the test set. We include the control questions to detect raters that randomly select the answers, thus rejecting the HITs without enough quality in their answers. The duration of each HIT was approximately 7 minutes, and we paid an average of 0.15$ per HIT. An example of the HIT configuration can be seen in the Figure 4.1.1

Given that one answer would not be enough to assume that it was the correct one. We launched 1000 HITs, 100 different tests each of them answered by ten users, and collected 5300 different relative comparisons (since they are answered by ten users, a total of 53000 answers). The relative comparisons tell us which of the two images is more similar to the reference. Furthermore, they are suitable to train a similarity metric with a triplet loss function.

To create the triplets for each question, we randomly selected one icon per class from

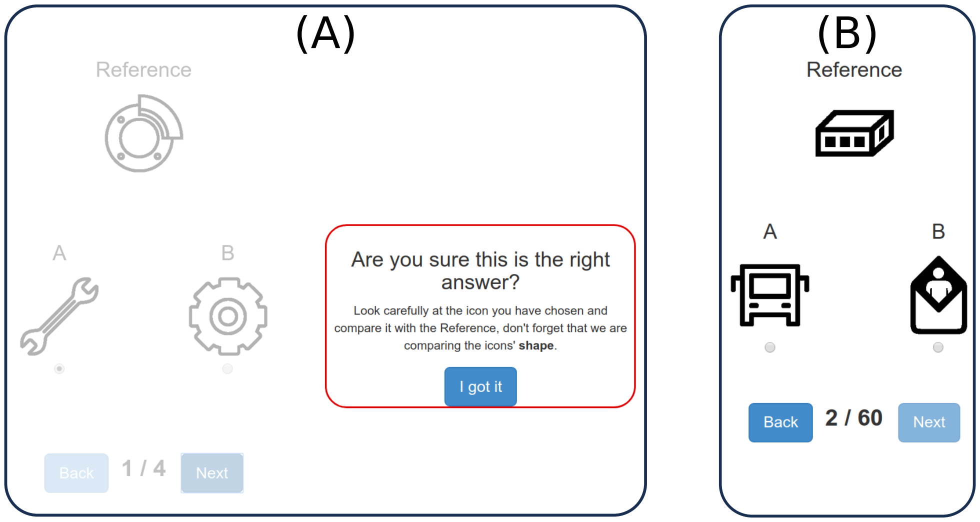

(A)
Reference
A
B
Are you sure this is the right answer?
Look carefully at the icon you have chosen and compare it with the Reference, don't forget that we are comparing the icons' **shape**.
I got it
Back
1 / 4
Next
(B)
Reference
A
B
Back
2 / 60
Next

Figure 4.1.1: The figure shows the configuration of a random HIT during the test phase. The Figure *(A)* shows the training phase where a guidance message appears if the user fails to answer. Since the relative comparisons have been manually created we know in advance which one is the correct answer. The Figure *(B)* corresponds to the test phase. In this case, since the relative comparisons are randomly created to obtain test data and validate the proposed models no guidance message will appear (we do not know the correct answer, indeed).

three different random classes. We rejected all the human raters that had more than one error out of seven in the control questions. In the end, 962 HITs were approved and 38 rejected. We allowed participants to do as many HITs as they wanted without repetition. A total of 213 users took part in the survey, 43% female. Among raters, 5.95% claimed some professional experience in user interface and interaction design, while 6.43% have had some professional experience with graphic design.

5 Modelling Visual Appearance of Icons

The following chapter explains, first, the model we are using: Siamese Neural Network and the best architecture we found, in Section 5.1. Then, the Section 5.2 define the loss function used to train the models together with the sampling method we used during training. Then, in Section 5.3, we comment how the Siamese Neural Networks were trained and their hyper-parameters configuration. After that, in Section 5.4 we explain how we tested our models and point out some architectures that did not work. Finally, the Section 5.5, shows some results obtained using our best model and presents an idea of real applications using it.

5.1 The Siamese Neural Network

Existing style similarity metrics [7, 26] use a handcrafted feature space to measure similarity in their respective datasets that do not generalize for our kind of data, consisting of plain, black and white icons. Thus, we will use a Siamese Neural Network (SNN) [4], avoiding to create the feature space by hand, to measure the similarity between icons, as it has been proved to work before in other image domains like interior design [2] and face similarity [27] among others.

The siamese architecture we introduce consists of three CNN sharing their parameters, each of them has four convolutional layers that are followed by a batch-normalization [11] layer and a max-pooling layer. The last pooling layer is connected to the linear classifier. The linear classifier contains three fully-connected layers where the first two have 4096 and 1024 features respectively. The last layer represents the final embedding $f(x)$ of the image x into the new feature space $\mathbb{R}^d$, where the value of d has been set to 256. We also included 2 dropout [31] layers between the fully-connected ones with a probability of 0.3 to stop the activations. An example of the architecture we described is shown in the Figure 5.1.1 inside the *CNN* container.

This architecture is trained using triplets of images which are called: a Reference, a Positive (similar style) and a Negative (different style) icons. Each image passes through the network to obtain an embedding, i.e. a high-level representation in a Euclidean space. The advantage of this new space is that it allows us to compare images and calculate the distances between them. To train the network we design a specific loss function which is explained in Section 5.2.

We train and test the network using the icons dataset gathered from *Noun Project*. We used 70% of the data to train the models, 10% to validate the hyper-parameters chosen and 20% to collect human-rated data to test the architectures. More details about the training and testing procedures are given in Section 5.3 and Section 5.4.

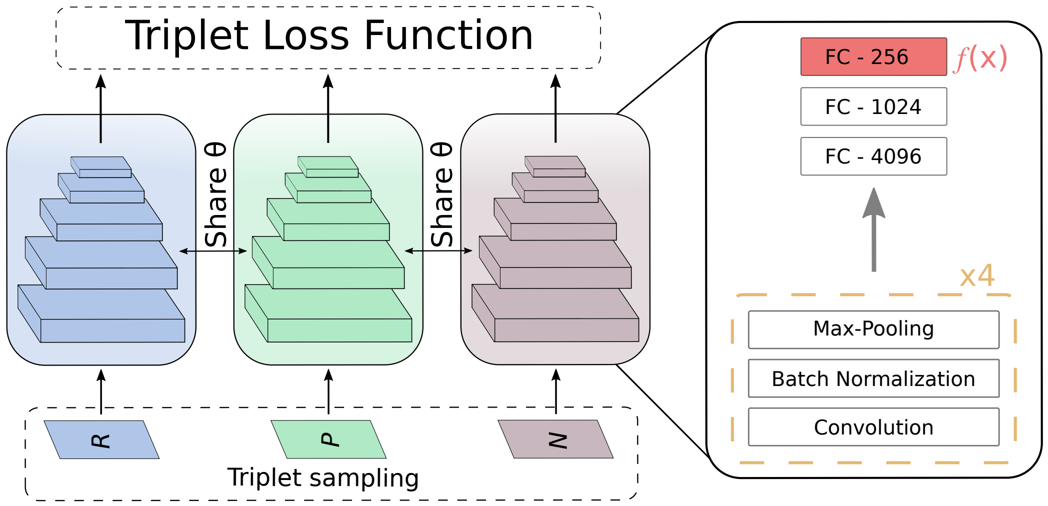


Figure 5.1.1: Architecture proposed to measure style similarity. The Siamese Network has three inputs: Reference (R), Positive (P) and Negative (N); and three Convolutional Neural Networks (CNN) to obtain its embeddings ($f(x)$). With these three embeddings we can compute the error of the network (L) using the triplet loss function described in Equation 5.1. The CNNs share the same structure and parameters. Each of them has 4 convolutional layers, that are followed by a batch-normalization layer and a max-pooling layer. The last pooling layer is connected to a linear classifier with 3 fully connected layers (FC). First FC has 4096 features, second one has 1024, while the last FC has only 256, furthermore, last FC of each CNN corresponds to the embedding $f(x^i)$, $i \in \{R, P, N\}$ of the input triplet $[x^R, x^P, x^N]$.

5.2 The Loss Function

Let's consider the output of the last fully-connected layer of the Convolutional Neural Network as an **embedding** $f(x) \in \mathbb{R}^d$ with input x (see Section 3.1). The embedding represents x in a new d-dimensional Euclidean space. Since we have a Siamese Neural Network formed by three CNNs that are identical with three inputs $[x^R, x^P, x^N]$, we get three embeddings as the output $[f(x^R), f(x^P), f(x^N)]$ where $f(x^R)$ corresponds to the embedding of a reference input while $f(x^P)$ is the embedding of an input of the same class as the reference and $f(x^N)$ is the input of an image that does not belong to the same class as the reference. Now we want to ensure that a reference icon x^R is closer to every icon of the same perceptual similarity (style and visual similarity) x^P than to the rest of icons with different image properties x^N. Thus our triplet loss function [23, 27] has to ensure that the distance in the d-dimensional Euclidean space between the reference and the positive icon is minimum while it is large between the reference and the negative icon.

$$\mathcal{L} = \left[\sum_{i}^{M} \|f(x_i^R) - f(x_i^P)\|_2^2 - \|f(x_i^R) - f(x_i^N)\|_2^2 + \alpha \right]_+ \tag{5.1}$$

Where M represents the training set of triplets and α is a margin enforced between negative and positive pairs which was empirically set to 0.2. The value α avoid the function to be 0 in some cases where the distance between the reference and the negative sample is larger than the reference and the positive sample, thus letting it find larger margins while training.

To apply the algorithm of Stochastic Gradient Descent to $\mathcal{L}$ we need to find the partial derivatives of each parameter of the function, keeping in mind that this applies only if $(||f(x_i^R) - f(x_i^P)||_2^2 - ||f(x_i^R) - f(x_i^N)||_2^2) \geq 0$, otherwise the value of the partial derivatives is zero:

$$\frac{\partial \mathcal{L}}{\partial f(x^R)} = \sum_{i}^{N} \Big[2(f(x_i^N) - f(x_i^P)) \Big] \tag{5.2}$$

$$\frac{\partial \mathcal{L}}{\partial f(x^P)} = \sum_{i}^{N} \Big[-2(f(x_i^P) - f(x_i^R)) \Big] \tag{5.3}$$

$$\frac{\partial \mathcal{L}}{\partial f(x^N)} = \sum_{i}^{N} \Big[2(f(x_i^R) - f(x_i^N)) \Big] \tag{5.4}$$

5.2.1 Adaptive sampling

If we would like to create all the possible triplets from the, approximately, 18200 icons in the training set we would have $\binom{18200}{3} \simeq 6.027 \cdot 10^{12}$ possible combinations, an unmanageable number using a standard desktop configuration. Furthermore, most of the generated triplets would easily satisfy the constraints of the loss function and not contribute to the training process at all, thus slowing it. For that reason, following the approach of Schroff et al. [27], we generate the triplets on the fly during the training process, selecting the ones that are active and help in the convergence. We generate triplets that violate the most the constraints imposed by the loss function which requires getting the embedding of the icons.

To do so, we randomly select one icon from the training set as the *reference*, then, we select the *positive* sample as the icon from the same class with the maximum distance in the Euclidean space to the reference, finally, we fix a different class and select the icon that has the minimum distance to the reference, obtaining the *negative* icon. We repeat this approach until a considerable number of triplets without repetition has been obtained. This process is applied before every epoch and it requires to compute the embeddings for every icon using the Siamese Neural Network. Although it increases the training time, it also ensures that the input triplets are meaningful. The Figure 5.2.1 shows an example of the triplets sampled during training in the first iteration.

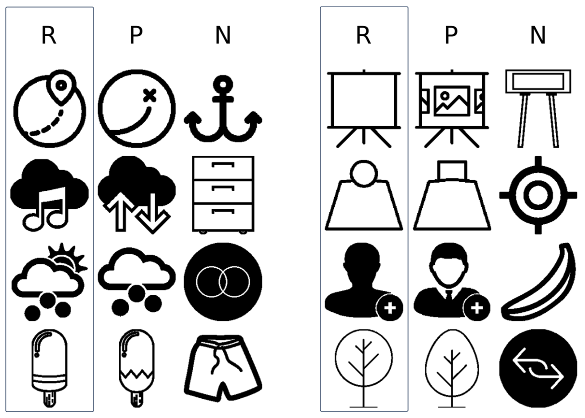


Figure 5.2.1: Examples of the triplets sampled during training. The letters R, P and N refers to the reference, positive and negative icon respectively. The positive icon and the reference are selected from the same class yet the positive icon is the one with the larger Euclidean distance to the reference within the icons in that class. The negative icon has the shorter euclidean distance among the icons within a randomly selected class.

5.3 Train the Models

The train deep models is a really complicated task due to the big amount of hyper-parameters that each model has - the size of the layers, dropout probability, learning rate, optimization parameters, etc.- and the vast search space that the combination of those yields. Furthermore, the stacked non-linearities of these models reduce the interpretability of their parameters. As a result of those problems, the training process requires a lot of empirical testing and comparisons between different architectures, as well as, using computationally expensive grid or random searches to find the best set of hyper-parameters. We followed an incremental approach while designing the models, we were stacking layers in the network until we found that using more than four convolutions the error rates given by the loss function were not decreasing.

During training, we used ADAM optimization [13] and the triplet sampling explained in the Section 5.2.1. The mini-batch had a size of 16 images and to update the parameters of the network we use standard back-propagation [9, 18] In training time, we perform two sequential operations with each image before feeding it to the network: first, *data augmentation* and second, *random crops*. For the data augmentation part, we randomly flip, rotate or leave the image as it is. For the crops, we randomly perform a crop of size 180x180 aligned to the corners in the original image, with size 200x200. We started the training with a learning rate of 10^{-4} that was reduced to let the model converge.

The following list summarizes the architectures of few of the models we tried until finding the one that gave us the best result:

– **Model-A** Add *max-pooling* layers between the convolutions and with two *fully-connected* layers

– **Model-B** Similar to Model-A, adding *dropout* between the two fully-connected layers

– **Model-C** Similar to Model-B, adding *Rectified Linear Units (ReLU)* between convolutions and also dropout between the two fully-connected layers

– **Model-D** Adding only max-pooling between the convolutions and using three fully-connected layers with dropout between them

– **Model-E** Add *batch-normalization* layers, max-pooling and ReLUs between convolutions, and use three fully-connected layers with dropout between them

The Table 5.4.1 shows a comparison on the models with the parameters that gave us best performance against human-raters and other the method proposed by Garces et al [7].

5.4 Test the Models

Once the models are trained, we evaluate their performance by comparing their precision on the gathered data from the MTurk HITs. At testing time, no data augmentation is applied and the inputs are directly scaled to 180×180 without cropping. Since the data gathered from MTurk to test the models only tell us which icon the user perceives more similar to the reference and with the Siamese Neural Network what we are obtaining is an embedding $f(x)$, we need to transform it to make it comparable. First, we obtain the embeddings for the three inputs of the triplet $[f(x^R), f(x^P), f(x^N)]$, since they are in a 256-dimensional Euclidean space, we can calculate the distances with respect to the reference $d(R, P)$ and $d(R, N)$. Actually, what we are aiming to obtain is a function of similarity instead of a distance, thus we define the similarity between two icons $s(R, P)$ as:

$$s(R,P) = \frac{1}{1 + d(R,P)} \tag{5.5}$$

when the positive P and reference R icon are really similar $d(R, P) = 0$ their similarity will be $s(R, P) = 1$. In the opposite case, if $d(R, P) = \infty$ we have that $s(R, P) = 0$. Knowing that $d(R, P)$ cannot be ∞ we can define the probability of choosing the icon P against N as:

$$\mathbb{P}(P) = \frac{s(R,P)}{s(R,P) + s(R,N)} \tag{5.6}$$

$$\mathbb{P}(N) = \frac{s(R,N)}{s(R,P) + s(R,N)} \tag{5.7}$$

After we obtain the probability of choosing the positive icon over the negative we compare two properties: precision and perplexity. We calculate **precision** in two ways: assuming the correct answer relies on each opinion separately (*raw*) or assuming the majority opinion is the correct one (*majority*). We also compare our results with 2 baselines previously calculated: the Humans and the Oracle precision. To compute Humans baseline, we count the rater's opinion and compare it to the majority. For the Oracle baseline, we count the opinion of the majority on each relative comparison, that's why its precision computing the majority is always 1.

The **perplexity** $\mathbb{Q}$ is often used for measuring the usefulness of a model when predicting a sample. Its value is 1 when the model makes perfect predictions on every sample, while its value is 2 when the output is 0.5 for every sample, meaning total uncertainty. We define the perplexity of our model as

$$\mathbb{Q} = 2^{\left(-1/N \sum_{i=1}^{N} \log_2 \mathbb{P}(P_i)\right)} \tag{5.8}$$

To know which one is the positive sample P in the relative comparison we rely on the user's ratings. The value $\mathbb{P}(P)$ will be the probability given by the model using Equation 5.6, N corresponds to the number of entries we use to compute the perplexity which can also be computed using raw and majority data. The Table 5.4.1 shows the precision and perplexity of the described architecture.

	Precision		**Perplexity**	
Model	**Raw**	**Majority**	**Raw**	**Majority**
Humans	0.771	0.842	-	-
Oracle	0.859	1	-	-
Garces et al.	0.609	0.627	1.578	1.591
Model-A	0.519	0.521	1.603	1.617
Model-B	0.508	0.507	1.608	1.622
Model-C	0.693	0.726	1.500	1.513
Model-D	0.671	0.702	1.543	1.556
Model-E	0.667	0.699	1.515	1.527
Best model	0.706	0.738	1.555	1.568

Table 5.4.1: Comparison of the precision and perplexity of different models and methods. We can observe how model-F outperforms the rest comparing the precision and is the closest one to the human ratings. On the other hand, its perplexity values are not the best but there are not significant differences between methods.

5.5 Results and Applications

2D Visualization To visualize what the Siamese Neural Network has learned we use a non-linear dimensionality reduction technique, called t-Distributed Stochastic Neighbor Embedding (t-SNE) [33]. This method aims to preserve the original structure of the data while it reduces the dimensionality of the original dataset while preserving the distances between items. Using it we transform our 256-euclidean embeddings into a two-dimensional array that represents the Cartesian coordinates of each image. The Figure 5.5.1 shows the result of applying t-SNE to all the embeddings of our test set. We can see how the icons that have similar style and distinctiveness properties are grouping together in the new two-dimensional Cartesian space.

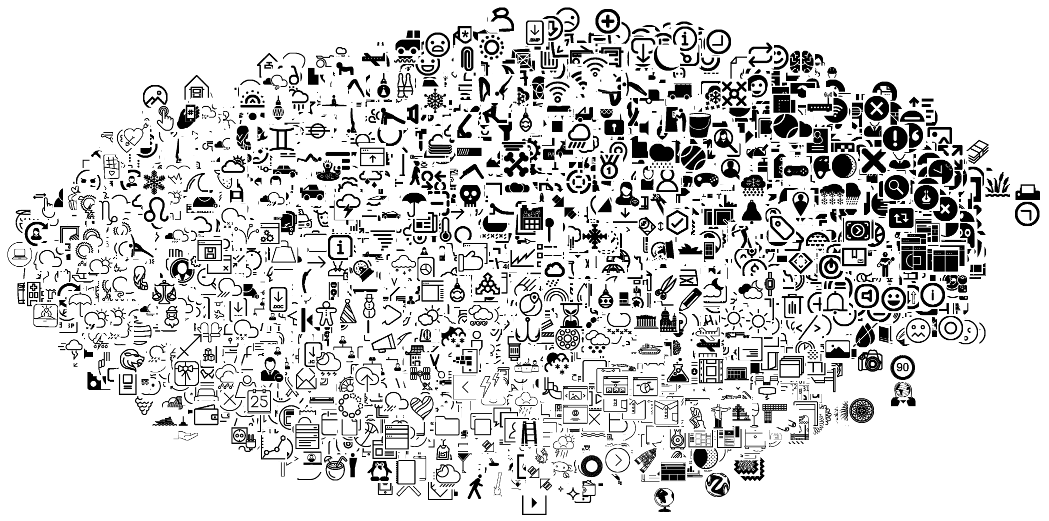


Figure 5.5.1: Visualization of the embeddings $f(x)$ of every image of the test set using the algorithm t-SNE. We can observe how groups of icons that are easily identifiable are placed with a big distance between them, while icons that are less identifiable are placed together. Also, we can observe how sub-groups of icons with shame shape are also placed together.

Search by Similarity We also obtained sets of icons optimized for a given reference. Results can be visualized in Table 5.4.1. We compare ourselves with the output given by the method presented by Garces et al. [7]. While Garces et al. method focus on modelling style similarity using a hand-crafted feature space, ours aim to obtain a new representation of the icons in a feature space learned by the Siamese Neural Network where the properties of style and visual similarity are considered.

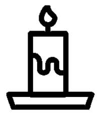
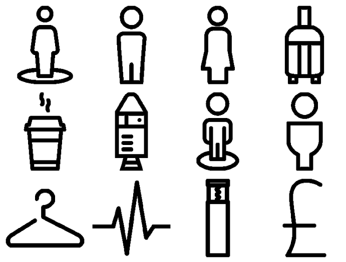
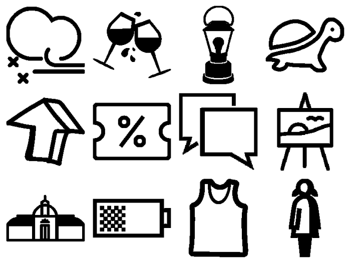
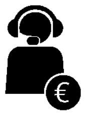
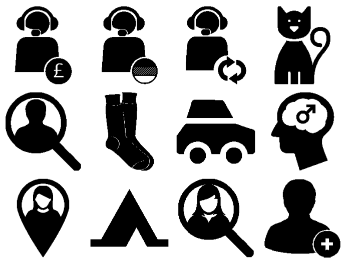
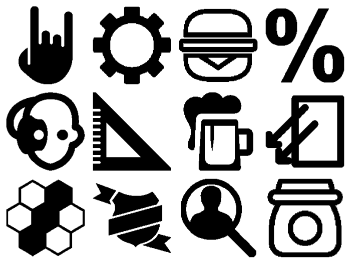
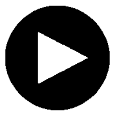
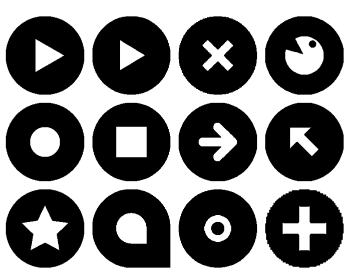
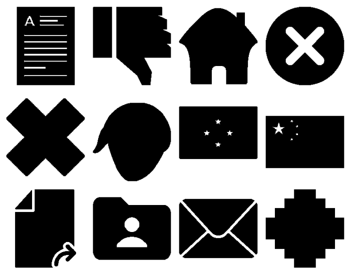
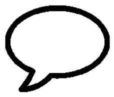
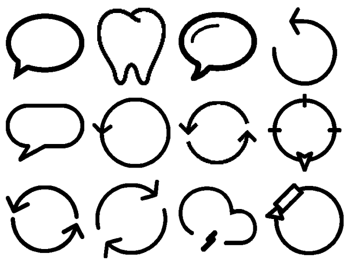
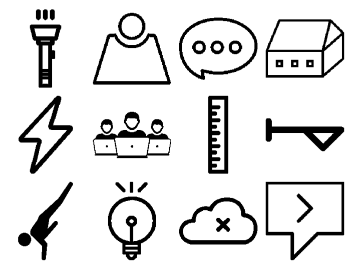
Reference	Our method	Garces et al.
		
		
		
		

Table 5.5.1: The following figure shows the most similar images given a reference and compares the output given by our method with the output given by Garces et al. [7]. We can see how our method returns a set of icons that share similar visual similarity and style properties.

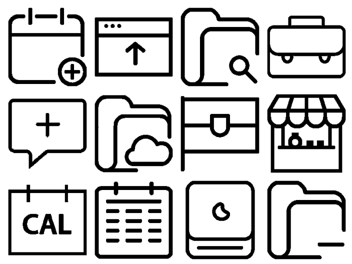
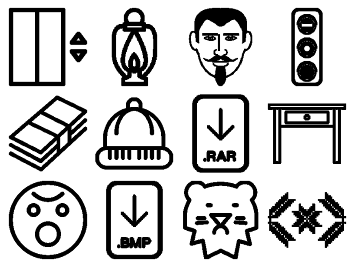
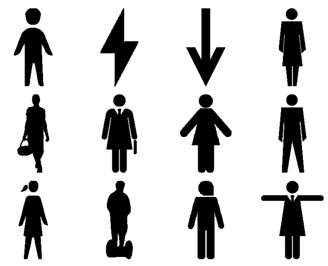
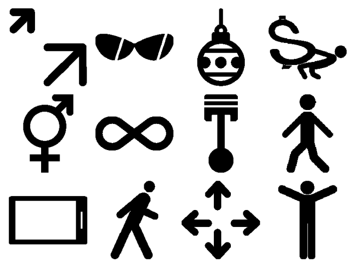
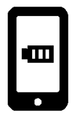
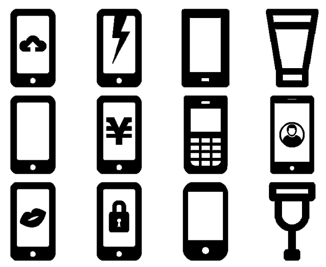
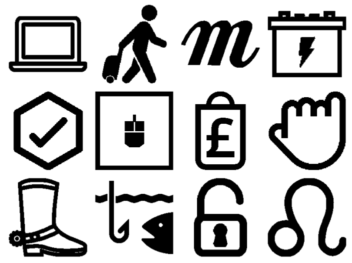
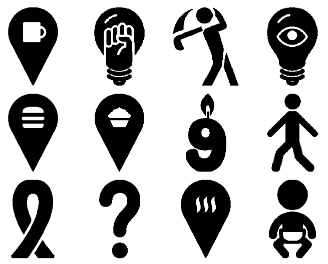
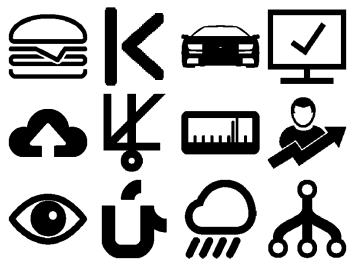
Reference	Our method	Garces et al.
		
		
		
		

Table 5.5.2: The following figure shows the most similar images given a reference and compares the output given by our method with the output given by Garces et al. [7]. We can see how our method returns a set of icons that share similar visual similarity and style properties.

Optimized Icon Sets A direct application of our method would be to use it to help designers creating applications or graphical user interfaces. The idea relies on obtaining icon sets optimized for the properties of style and visual similarity. We show an example that returns optimized icons sets for the collections of *animals (A)*, *arrows (B)* and *buildings (C)*. To obtain a set we need to consider the distance between all the possible candidate icons. In this case, since we are using three collections, we will obtain sets of three icons (each belongs to a different class). We define each candidate set as $(A, B, C) \in T$, where T is the set containing all the possible combinations of icons, which in this case contains 203 elements. Once we have all the combinations we can compute the distance of a collection as $D_{collection}(A, B, C) = D(A, B) + D(B, C) + D(A, C)$. After we get all the possible distances within the three collections we just need to find $argmin_{i,j,k} D_{collection}(T(A_i, A_j, B_k)$. At the end, the set that has the minimum distance between its icons will be the candidate. An example of candidate sets obtained following the explained method can be seen in Figure 5.5.2.

Perceptual Similarity We also applied our model to the icons used by Demiralp et al. [5] (see Chapter 2) to gather human-rated data and to generate their kernels and compare with their perceptual kernel (see Figure 2.0.2) and their pairs of icons with maximum perceptual distance (Figure 5.5.3 *(B)*). We utilized our method to create a perceptual kernel (Figure 5.5.3 *(A)*) find pairs of icons with the maximum distance in the new 256-dimensional Euclidean space created by the model (Figure 5.5.3 *(C)*). Overall, we can see how most of the icons have a coherent result. However, in comparison with Demiralp's results, we notice that the maximum distance to the square is the diamond icon (in the figure the first column and second row). This behaviour comes from the features obtained by the convolutional networks which are not invariant to rotation or scale. We also use our metric to show the icons of our dataset with the maximum distance in the Euclidean space (Figure 5.5.3 *(D)*)

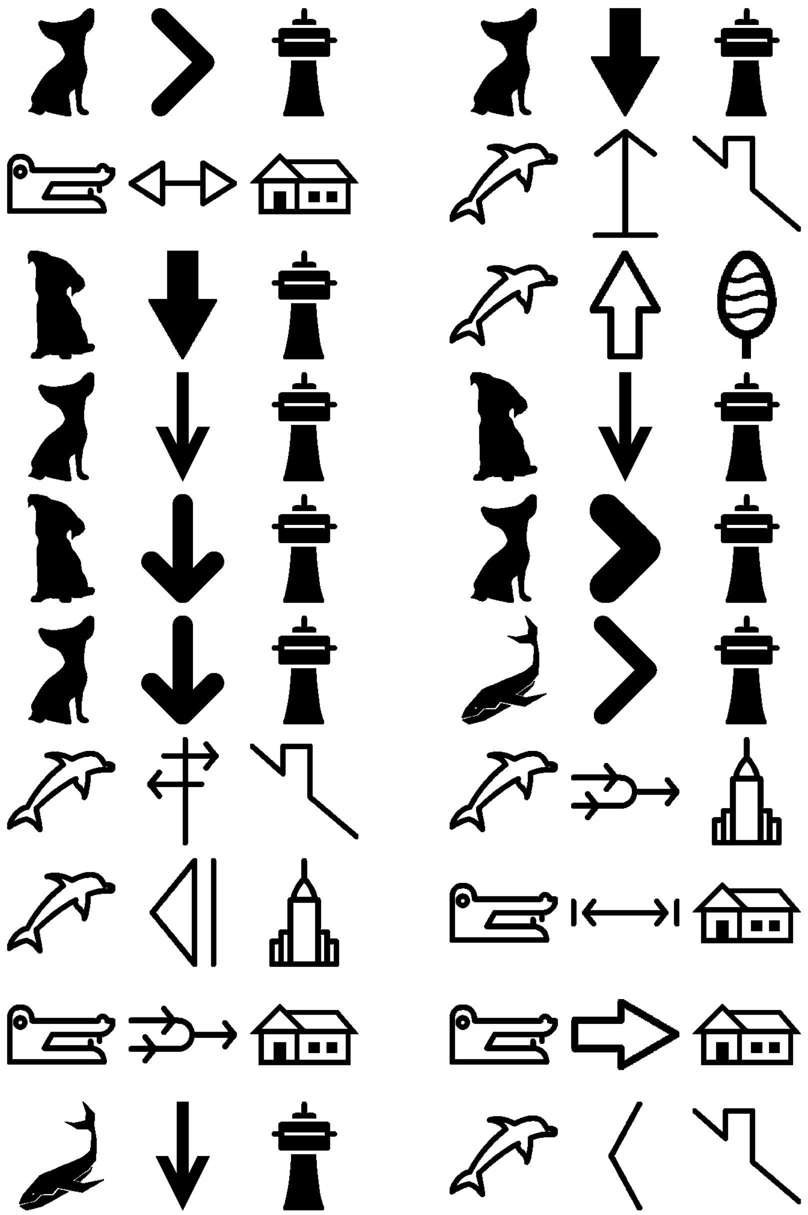


Figure 5.5.2: Example of optimized sets of icons from collections: animals, arrows and buildings.

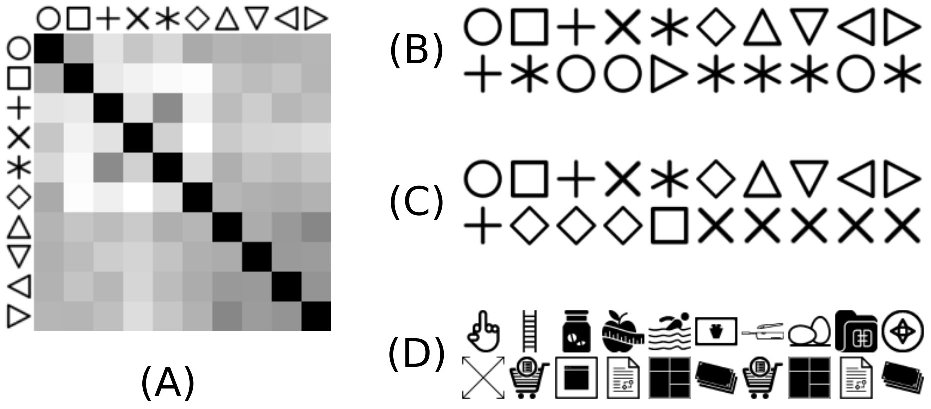


Figure 5.5.3: *(A)* kernel obtained using the same icons as Demiralp et al. with our similarity metric, we observe in the triangles that our neural network is not invariant to translation and rotation contrarily to Demiralp's metric. In *(B)* we show the results from Demiralp et al. for each icon on the left, we find the icons with the largest distance according to their perceptual metric. Figure *(C)* shows a pair of icons with the maximum distance obtained using our method. The example of the square and the diamond seems unintuitive. This failure case of our model is produced due to the behaviour of CNNs which are not invariant to rotation or scale. Finally, in *(D)* we show pair of icons with maximum distances using our whole dataset in the Euclidean space created by the CNN.

6 Conclusion and Future Work

In this work we have presented a model for measuring the properties of style and visual similarity. Our approach is based on a siamese CNN that produces meaningful embeddings representing both properties in a 256-dimensional Euclidean space. To train the models we rely on a weakly-labelled icons dataset gathered from an online database, while we launched crowd-sourced experiments to collect human-rated data and validate the architectures. We show the suitability of the embeddings in a t-SNE visualization and proposing sets of icons optimized for style and visual similarity of a given reference.

There are many lines of future work within the context of our project. First, we could try to decouple style and shape by learning an additional Siamese Neural Network trained using more curated data like the one shown in Figure 6.0.1, which could be used to learn a represenation of the *arrow shape* invariant to the style of the input. This kind of data would be useful to disambiguate both properties although it is very infrequent in the dataset, and

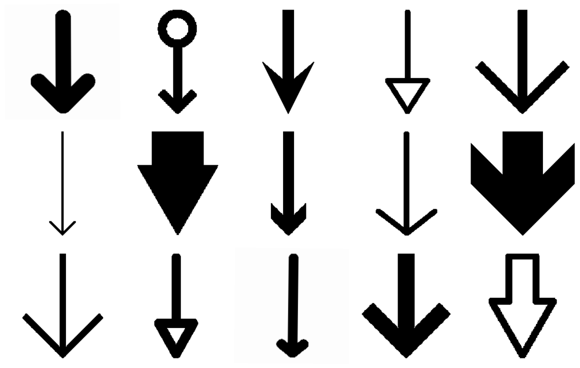


Figure 6.0.1: Example of data that could be useful to disambiguate style and shape. Figure shows a group of arrows pointing down with a wide variety of styles.

it would be necessary to generate it. Another option of future work, following the work of Laursen et al [6] would be to evaluate our metric to propose icon sets optimized for the properties of identifiability and distinguishability. This would require to gather new data from user studies.

Finally, a more challenging idea would be to use Varational Autoencoders or Generative Adversarial Networks (GAN) [8] - algorithms that encode images to a latent space and learn a mapping from it to the distribution of the original data - to generate new synthetic icons that follow the distribution of the original dataset of icons. Indeed, we have developed a small VAE that re-builds icons using a latent space with 128 features. The structure of the network is also based on convolutional layers and in the Figure 6.0.2 we show examples of the reconstructions obtained. This approach would allow us to synthesize new icons given a input image (i.e. a photograph of a face) and a style image (i.e. solid icon) obtaining a new icon with a solid face that keeps the content of the input image while it has the style of the style image, an example can is shown in Figure 6.0.3.

6.1 Personal Note

Summarizing my experience in few words would sound like *completely amazing.* During this work I was able to learn many new things, to push my boundaries and, the most important

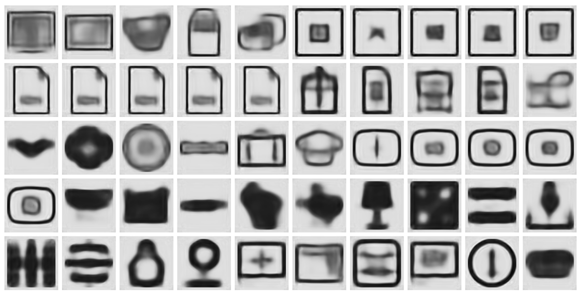


Figure 6.0.2: Examples of the synthetic images build by the Varational Autoencoder. These are is preliminary results and we are currently working on refining the architecture.

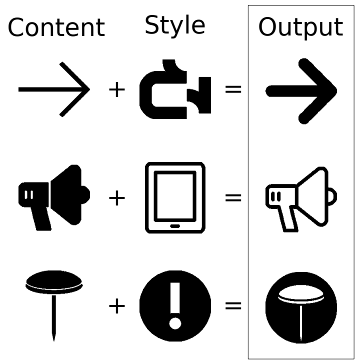


Figure 6.0.3: Example of the process of style transfer between icons. The ideal goal of the new work would be to synthesize new images given a pair of icons representing the content and the style of the output respectively.

thing, work in really exciting projects and topics I really enjoy. It was also another chance to be in contact with the research community and the people from the Graphics and Imaging Lab with whom I have shared lots of comments about papers, opinions and talks. To talk with people that love what they are doing makes you feel realized and wanting to learn more and more. Thanks to the experience I have had in the group while doing this project I decided to continue by doing a PhD with them. Finally, I would like to thank Elena and Diego for giving me the chance to do this exciting project.

Bibliography

[1] Xiang Bai, Xingwei Yang, Longin Jan Latecki, Wenyu Liu, and Zhuowen Tu. Learning context-sensitive shape similarity by graph transduction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(5):861–874, 2010.

[2] Sean Bell and Kavita Bala. Learning Visual Similarity for Product Design with Convolutional Neural Networks. *ACM Trans. Graphics (Proc. SIGGRAPH)*, 34(4), 2015.

[3] Serge Belongie, Jitendra Malik, and Jan Puzicha. Shape matching and object recognition using shape contexts. *IEEE transactions on pattern analysis and machine intelligence*, 24(4):509–522, 2002.

[4] Jane Bromley, Isabelle Guyon, Yann Lecun, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In *In NIPS Proc*, 1994.

[5] Çağatay Demiralp, Michael S Bernstein, and Jeffrey Heer. Learning perceptual kernels for visualization design. *IEEE transactions on visualization and computer graphics*, 20 (12):1933–1942, 2014.

[6] Lasse Farnung-Laursen, Yuki Koyama, Hsiang-Ting Chen, Elena Garces, Diego Gutierrez, Richard Harper, and Takeo Igarashi. Icon Set Selection via Human Computation. In *Pacific Graphics Short Papers*, 2016.

[7] Elena Garces, Aseem Agarwala, Diego Gutierrez, and Aaron Hertzmann. A Similarity Measure for Illustration Style. *ACM Trans. Graphics (Proc. SIGGRAPH)*, 33(4), 2014.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[10] Ming-Kuei Hu. Visual pattern recognition by moment invariants. *IRE transactions on information theory*, 8(2):179–187, 1962.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[12] Alireza Khotanzad and Yaw Hua Hong. Invariant image recognition by zernike moments. *IEEE Transactions on pattern analysis and machine intelligence*, 12(5):489–497, 1990.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[14] Yanir Kleiman, Oliver van Kaick, Olga Sorkine-Hornung, and Daniel Cohen-Or. Shed: shape edit distance for fine-grained shape similarity. *ACM Transactions on Graphics (TOG)*, 34(6):235, 2015.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[16] Kin Chung Kwan, Lok Tsun Sinn, Chu Han, Tien-Tsin Wong, and Chi-Wing Fu. Pyramid of Arclength Descriptor for Generating Collage of Shapes. *ACM Trans. Graphics (Proc. SIGGRAPH Asia)*, 35(6):1–12, 2016.

[17] Longin Jan Latecki, Rolf Lakamper, and T Eckhardt. Shape descriptors for non-rigid shapes with a single closed contour. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages 424–429. IEEE, 2000.

[18] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, December 1989.

[19] Tianqiang Liu, Aaron Hertzmann, Wilmot Li, and Thomas Funkhouser. Style Compatibility for 3D Furniture Models. *ACM Trans. Graphics (Proc. SIGGRAPH)*, 34(4): 1–9, 2015.

[20] Zhaoliang Lun, Evangelos Kalogerakis, and Alla Sheffer. Elements of Style: Learning Perceptual Shape Style Similarity. *ACM Transactions on Graphics (TOG)*, 34(4): 84:1–14, 2015.

[21] Peter O'Donovan, Jānis Lībeks, Aseem Agarwala, and Aaron Hertzmann. Exploratory font selection using crowdsourced attributes. *ACM Transactions on Graphics (TOG)*, 33(4):92, 2014.

[22] Robert Osada, Thomas Funkhouser, Bernard Chazelle, and David Dobkin. Shape distributions. *ACM Transactions on Graphics (TOG)*, 21(4):807–832, 2002.

[23] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep face recognition. In *Proceedings of the British Machine Vision Conference (BMVC)*, 2015.

[24] Boris Teodorovich Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[25] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[26] Babak Saleh, Mira Dontcheva, Aaron Hertzmann, and Zhicheng Liu. Learning Style Similarity for Searching Infographics. *Gi*, pages 3–5, 2015.

[27] Florian Schroff, Dmitry Kalenichenko, and James Philbin. FaceNet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 07-12-June, pages 815–823, 2015.

[28] Vidya Setlur and Jock D. Mackinlay. Automatic Generation of Semantic Icon Encodings for Visualizations. In *SIGCHI*, pages 541–550, 2014.

[29] Vidya Setlur, Conrad Albrecht-Buehler, Amy A. Gooch, Sam Rossoff, and Bruce Gooch. Semanticons: Visual metaphors as file icons. *Computer Graphics Forum (Proc. Eurographics)*, 24(3):647–656, 2005.

[30] Edgar Simo-Serra and Hiroshi Ishikawa. Fashion Style in 128 Floats: Joint Ranking and Classification using Weak Data for Feature Extraction. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[31] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[32] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[33] Laurens van der Maaten and Geoffrey E. Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[34] Wenpeng Yin, Hinrich Schütze, Bing Xiang, and Bowen Zhou. ABCNN: attention-based convolutional neural network for modeling sentence pairs. *CoRR*, abs/1512.05193, 2015.

[35] Dengsheng Zhang and Guojun Lu. Shape-based image retrieval using generic fourier descriptor. *Signal Processing: Image Communication*, 17(10):825–848, 2002.