Adaptive Deep Learning through Visual Domain Localization

Facoltà di Ingegneria Informatica
Corso di Laurea Magistrale in Artificial Intelligence and Robotics

Candidate
Gabriele Angeletti
ID number 1459796

Thesis Advisor:
Prof. Tatiana Tommasi
Co-Advisor:
Prof. Barbara Caputo

Academic Year 2016/2017

Abstract

The goal of Domain Adaptation is to minimize generalization error in those cases where the i.i.d. assumptions do not hold. In the context of computer vision, and image recognition in particular, there can be a variety of factors that influences the shift between training and test distributions: background, lighting conditions, resolution, translation, scale, ...

These factors can have a dramatic impact on test performance.

Deep learning technologies, despite the many breakthroughs produced in recent years in both academia and industry, suffer from the domain shift problem.

We designed a technique to extend Convolutional Neural Networks (CNNs) to minimize generalization error in the case of domain shift.

In particular, our method performs best when the causes of the shift are object transformations— such as translation and scale.

We show that our proposed technique outperforms previous state-of-the-art approaches across a number of benchmark datasets and baseline network architectures.

Contents

List of Tables

List of Figures

Table 1: Mathematical Notation

Symbol	Meaning
x	A scalar.
$\boldsymbol{x}$	A vector.
$\boldsymbol{X}$	A matrix.
X	A random variable.
$P(\cdot)$	A probability distribution.
net	The current state of a neural network.
$\hat{y}$	The output of a neural network.
θ	The set of parameters of a neural network.
$L(\cdot)$	A loss function.
$\Omega(\theta)$	A regularization function.
α	Learning rate parameter.
σ	An activation function.
$*$	The convolution operation.

Chapter 1

Introduction

After many decades of ups and downs, deep learning techniques recently produced unprecedented breakthroughs in complex domains like computer vision, speech recognition, and natual language processing.
Deep learning [17] is a subset of machine learning based on Artificial Neural Networks—statistical models loosely inspired by insights from neuroscience. Two were the main enablers for this revolution: the ever increasing availability of large labeled datasets and the terrific growth of computational power that took place in recent years, in particular the advent of GPUs to speed-up by orders of magnitude the kind of computations needed by deep learning.

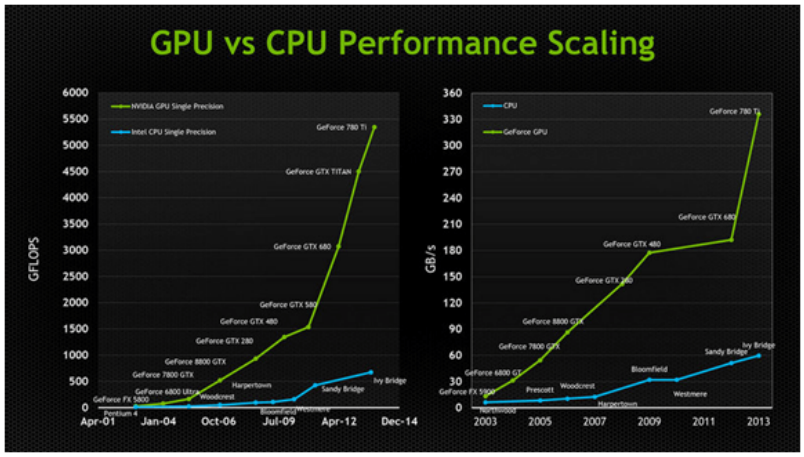

Figure 1.1: GPUs are optimized for parallel tasks, and thus are particularly suited to carry out computations like matrix multiplication, which is obiquitous in deep learning.

In computer vision in particular, state-of-the-art approaches for tasks such as image recognition, object detection and image segmentation all have

Convolutional Neural Networks (CNNs) [16] at their core, a specialization of the more general feedforward network architecture to deal with data with a grid-like topology, like images.

Like the majority of machine learning technologies, also deep learning relies on the i.i.d. assumptions, a set of assumptions drawn from statistical learning theory [31] by which the training and the test sets are generated from the same data generating process, i.e. they came from the same distribution.
Neural Networks are known for their great generalization capabilities. In fact, when trained on very large datasets, such as ImageNet [28], CNNs can be used to extract meaningful features from new datasets, on top of which a linear classifier (e.g. Linear SVM or Logistic Regression) can be successfully trained. This method goes under the name of Transfer Learning [30].

Transfer learning works well when the training and test sets came from very similar distributions. When this condition is not met, CNNs tends to perform poorly despite their outstanding generalization capabilities. Hence, techniques are needed to deal with such cases.
The problem of dealing with settings in which the i.i.d. assumptions do not hold is known as Domain Adaptation [13], in which we seek to minimize generalization error in those situations in which there is a distribution shift between the training set (called source domain in domain adaptation terminology) and the test set (target domain).
A satisfying solution to this problem would be very appealing, because it would mean being able to train offline huge models on very large datasets, and having them generalize well when deployed on a real-world scenario. Imagine for instance a mobile app in which users can take photos, which should be analyzed (e.g. classified) by the app in some way. These photos are likely to come from a different distribution from that of ImageNet. Having a model perform well on this target domain from the first use can represent an advantage from a user experience perspective.

This work is about designing techniques to deal with situations in which plain transfer learning does not work well. In particular, we focused on those cases where the domain shift is caused by object transformations, such as translation and scale. This is because we are mainly interested in robotics applications, in which a robot might take pictures from very different angles and viewpoints—a situation which is very different from that of the majority of image recognition datasets, in which the object always appears at the center of the image.

The rest of the thesis is organized as follows. After a review of background concepts on domain adaptation and deep learning in chapter 2, we review recent literature in chapter 3. In chapter 4 we describe our approach, the Domain Multiplicative Fusion (DMF) Network, describing the main components and the techniques it builds upon. In chapter 5 and chapter 6, experiments across a variety of robotics datasets and network architectures are reported. The thesis concludes with chapter 7 with a summary and a discussion on future research.

Chapter 2

Background

This chapter provides an overview of the problems addressed in this thesis work, as well as the main technologies upon which our approach is built. Section 2.1 is a formal description of the domain adaptation task, in which the main goals and challenges are highlighted. Section 2.2 is an introduction to the basics of Deep Learning, a branch of machine learning concerned with the study of Artificial Neural Networks (ANNs). This section draws heavily on [9].

2.1 Domain Adaptation

Introduction The majority of machine learning (ML) algorithms [1] is based on the minimization (w.r.t. the model's parameters) of some function of the model's errors, also called the **loss function**: $L(y, \hat{y}; \theta)$. Its value depends on the reference output y, the model's output $\hat{y}$, and the model's parameters θ.
At a first glance, one might think that ML is just plain optimization, but actually there is a profound difference between the two. In sheer mathematical optimization, we have full access to the function we want to minimize (or maximize), while in ML this is not true. In particular, in ML we have access to only a subset of the input-output relationship, and we minimize the error function over this sample in the hope that doing so will also minimize the overall loss function (which we don't have access to). The loss function over

[1] In the following, we will restrict our discussion of ML algorithms to the subset of supervised learning, represented by algorithms that learns a function $f(\cdot)$ mapping an input vector x to an output vector $y = f(x)$.

this finite sample is defined as:

$$L_{sample}(y, \hat{y}; \theta) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{y}_i; \theta)$$

In the ML framework what we care about is **generalization**, that is the ability of a model to obtain a good performance on samples it has never seen during training (such samples are called *test sets*), according to some performance measure P.

The i.i.d. assumption A typical assumption done in machine learning settings is that the *training set* (X_{train}, Y_{train})[2] (the sample used to minimize L) and the *test set* (X_{test}, Y_{test}) (the sample used to evaluate generalization) are **i.i.d.**, that is, they are independently sampled from the same underlying distribution $P(X, Y)$. The fact that samples are drawn independently is what allows us to write the loss function as a summation of a per-sample defined error function. The majority of learning algorithms operate under this assumption, which is usually a fairly good approximation. But there are cases in which this assumption does not hold, and this causes a significant drop between training and test performance.

Motivation Domain Adaptation [13] consists in the design of algorithms that work even when the i.i.d. assumption does not hold [3], i.e. when the distribution from which training samples are drawn is different from the one at testing time. Domain adaptation has a slightly different terminology w.r.t. classical ML. In particular, the training distribution is called the *source domain* $D_S \sim P_s(X, Y)$, while the testing distribution is called the *target domain* $D_T \sim P_t(X, Y)$. Typically, we assume that the source data is abundant and labeled, while the target data is only partially (or not at all) labeled. Clearly, this is the most interesting setting because solving this problem would allow us to leverage data sets for which we have lots of labeled data, and at the same time obtain good performance on the data sets we were interested in the first place, but which often have few labeled data. This setting goes under the name of *Unsupervised* Domain Adaptation. If instead we assume that the target is partially of fully labeled, we talking about *Semi-Supervised* or *Supervised* Domain Adaptation respectively. This thesis work in particular focuses exclusively on the (harder) unsupervised setting.

[2] Here we are focusing only to supervised settings, but the domain adaptation problem can be defined also for unsupervised scenarios.

[3] Note that the assumption that samples are drawn independently is still valid.

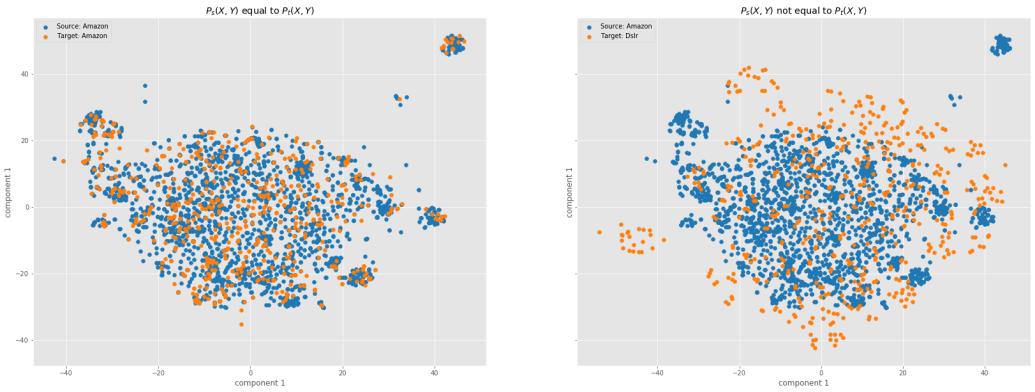

Figure 2.1: t-SNE [15] visualization of the features extracted from the Office dataset [**office**] using AlexNet [1]. When the source and the target distribution are the same (left), a classifier trained on the source data generalizes very well to the target data. When the i.i.d. assumption does not hold (right), the model will generalize poorly.

Domain Adaptation In ML, the joint distribution between data and labels $P(X, Y)$ is unknown. In domain adaptation, we call the joint distribution of the source domain $P_s(X, Y)$, while $P_t(X, Y)$ is the joint distribution of the target domain. The i.i.d. assumption consists in a assuming that $P_s(X, Y) = P_t(X, Y) = P(X, Y)$. In Domain Adaptation instead, we have $P_s(X, Y) \neq P_t(X, Y)$. We can visually see what are the implications of this fact in figure 2.1.
The source data is drawn i.i.d. from the source distribution, while the target data is drawn i.i.d. from the target marginal distribution over X:

$$S = \{(x_i, y_i)\}_{i=1}^{N} \sim (D_S)^N$$
$$T = \{(x_i)\}_{i=1}^{M} \sim (D_T^X)^M$$

The goal is to build a classifier $f : X \to Y$ capable of making correct predictions about the target labels Y_t:

$$\theta^\star = \max_\theta P(Y_t = \hat{Y}_t | X_t; \theta)$$

Where $\hat{Y}_t = f(X_t)$ are the labels predicted by the classifier and Y_t are the true target labels, which we don't have access to.

Object Recognition In the context of object recognition, the domain shift can be very large and it can have a variety of different causes, which can also

interact between each other in nonlinear ways. Factors such as background, lighting conditions, resolution, translation, scale and rotation, all contribute to the shift in distribution between source and target.

Figure 2.2: Examples of visual domain shift. The shift between the left and center images is clearly due to the background. The shift between the center right images is due to resolution. In fact, the center image was taken with a high-resolution camera, while the right image was taken with a webcam.

2.2 Deep Learning

Introduction Current processors' hardware substrate is very different from the biological substrate of the human brain. In fact, they are opposite. For this reason, many tasks that require effort and reasoning for a human to accomplish, such as chess, are pretty straightforward for machines, for those problems can be encoded into a set of formal rules which can then be solved algorithmically. Conversely, other kinds of tasks that humans do without even thinking about them are tremendously difficult for machines to accomplish. We find so easy to perform this kind of tasks that we don't even manage to explain how we do them. Encorporating the intuition and common sense that humans give for granted into machines is one of the holy grails of AI.

History At first, it was believed that intelligent behavior could be achieved by hard-coding knowledge into the system by means of some formal language. Reasoning would then be achieved through the use of inference rules, such as Modus Ponens. These were the so-called **Expert Systems** [12]. It was soon evident that the number of formal rules needed for intelligent behavior exceeds by several orders of magnitude the number of rules one could possibly write by hand.
This led to a major shift from deductive systems to inductive ones, where

the machine itself *learns* to extract knowledge from data. By seeing a lot of input-output examples in the form $y = f(x)$ it would figure out the relationship f itself. This was the beginning of **Machine Learning** (ML). In classical ML the input representation is still hand-crafted though. For instance, a classical ML image classification algorithm does not take in input the raw pixels of the image, but some other representation created by an ad-hoc procedure. This was a drawback because the majority of efforts went into the *feature engineering* process itself, and in many cases this is more an art than a science.
The natural evolution of this is to not only let the program learn the input-output relation $y = f(x)$, but also the input representation as well. That is, the algorithm learns both a suitable representation of the input $\phi(x)$, and then the relationship between this representation and the output $y = f(\phi(x))$. This research field is called **Representation Learning**.
Deep Learning is an extension of representation learning. The machine learns multiple levels of representations in a hierarchical fashion, where more complex concepts are built on top of simpler ones. Deep Learning is nowadays behind many state-of-the-art techniques in many research fields, like computer vision and speech recognition, and many believes it to be one of the most promising ways to reach human level artificial intelligence someday.
In the following, we will provide a very concise introduction to the main concepts of deep learning. In particular, we will cover only a tiny subset of the whole field: *feed-forward (convolutional) neural networks* for *supervised learning* (classification). Thus, the setting is the typical one for classification problems: we have a *data set* of n samples, of which the i-th sample is $(X_i \in \mathbb{R}^K, Y_i \in \{1, 2, \ldots, C\})$, where X_i is a *tensor*, that is a multi-dimensional array, and Y_i is the class label for sample i. The task is to come up with a model capable of predicting the correct class label $Y^{'}$ for a previously unseen instance $X^{'}$.

2.2.1 Feed-Forward Neural Networks

The Feed-Forward Network can be thought of as a general framework in which different layers of processing units are stacked on top of each other. Each layer implements a function that takes in input the output of the previous layer.[4]. A single layer can be viewed either as implementing a vector valued function or as a set of parallel processing units, each of which takes in input the outputs of the units in the previous layer and performs some sort of computation. The name neural networks stem from the fact that

[4]Note that also the input and the output of the network are considered layers

these models are loosely inspired by how the human brain works, and in this regards, the parallel processing units in each layer perform a role analogous to that of neurons in the brain. The classical example of a feed-forward network architecture is the MultiLayer-Perceptron (MLP) [4], in which each layer implements an affine mapping of the input followed by an element-wise non-linear function:

$$h = \sigma(Wx + b) \tag{2.1}$$

Where $x \in \mathbb{R}^{m \times n}$ is the layer's input, $h \in \mathbb{R}^{m \times p}$ is the output, σ is the non-linearity (called the *activation function*) and $W \in \mathbb{R}^{n \times p}, b \in \mathbb{R}^{p}$ are the layer's parameters. The parameters W are called the *weights*; in MLPs there is a different weight for each pair of input-output units. The intercept of the affine mapping b is called the *bias*, because it represents the output of the layer when the input is zero. Regarding σ, there exists many non-linearities in the deep learning literature, some of the most popular are the *sigmoid or logistic* function, the *hyperbolic tangent (TanH)* and the *rectifier linear unit (ReLU)*:

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}, TanH(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, ReLU(x) = \max(0, x) \tag{2.2}$$

The layer $h = \sigma(Wx + b)$ described above can be thought of as a way of stretching and squashing space in order to project the input onto a new space in which it is linearly separable, that is, there exists a hyperplane separating the different classes. In the following depiction we can see such a layer in action:

When we stack a number of such layers one on top of the other, we have a deep network. In particular, the input-output mapping defined by a MultiLayer-Perceptron with n layers is recursively defined as:

$$\hat{y} = W_n(\sigma(W_{n-1}(\sigma(W_{n-2}(\ldots W_1 x + b_1 \ldots)) + b_{n-2})) + b_{n-1}) + b_n \tag{2.3}$$

With W_1, b_1 denoting the parameters of the first layer and W_n, b_n the parameters of the last layer. The dimension of the first layers is constrained to be $(m, \cdot)$, with m number of inputs, while the dimension of the last layer is constrained to be $(\cdot, c)$, with c number of classes. The dimensions of all the other layers are hyper-parameters of the model that the designer of the

[6] Images taken from: http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

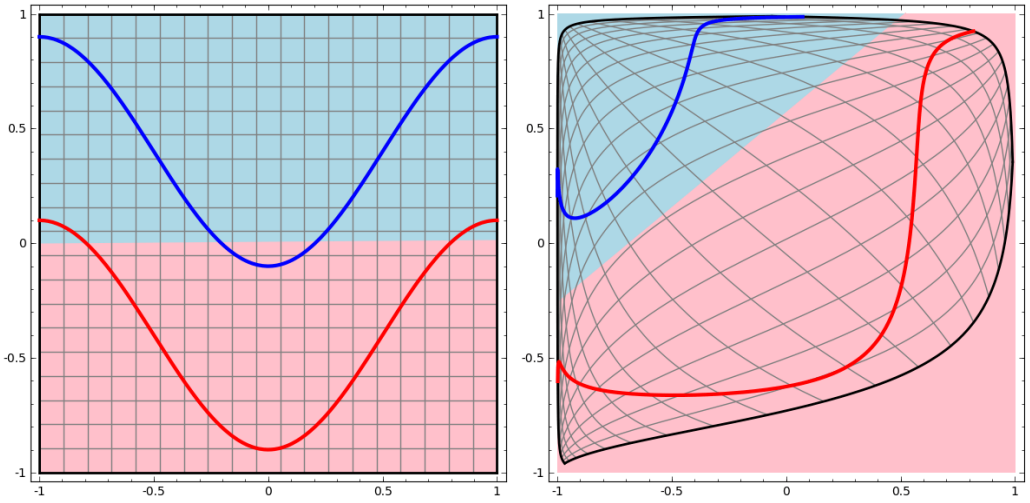

Figure 2.3: The original data (left) is not linearly separable, that is, there does not exists a hyperplane capable of separating the red curve from the blue curve. The transformation $h = \sigma(Wx + b)$ project the input x into a new space (right) where it's easy to find a separating hyperplane.[6].

architecture can arbitrarily chose. The number of layers n is also a hyper-parameter.
The output vector $\hat{y}$ has one entry for each class, and it can be thought of as the scores the network assigns to different classes. Since this vector can assume arbitrary values, it is typically followed by a function which converts the raw class scores into a probability distribution over classes, that is that normalizes the scores so that they sum to one. The most popular function used in the deep learning literature is the **softmax** function:

$$softmax(z) = \left[\frac{e^{z_1}}{\sum_{i=1}^{c} e^{z_i}}, \ldots, \frac{e^{z_c}}{\sum_{i=1}^{c} e^{z_i}}\right], z = [z_1, \ldots, z_c] \tag{2.4}$$

Thus, the complete input-output mapping of the MLP is:

$$\hat{y} = softmax(W_n(\sigma(W_{n-1}(\ldots W_1 x + b_1 \ldots)) + b_{n-1}) + b_n) \tag{2.5}$$

Now that we've defined our model, we need two more things: a definition of error and a procedure to learn from such errors.

The Loss Function The first thing to note is that the output of the model is a vector with c entries, one for each class. So we need to define the reference

value y also as a vector with c entries. This is done by using the so-called *one-hot-encoding*, that is, class i is represented by a vector in which the i-th position is 1 and the other $c-1$ positions are 0.[7] Having both the model output $\hat{y}$ and the reference value y, we can define the loss function. The most popular choice for classification tasks in deep learning is the **cross-entropy** loss function, defined as:

$$L(y, \hat{y}; \theta) = -\sum_{i=1}^{c} y_i \log(\hat{y_i}) + (1 - y_i) \log(1 - \hat{y_i}) \tag{2.6}$$

It is easy to see that for all the entries for which $y_i = 0$, only the second term contributes to the loss, and for the entry for which $y_i = 1$ (the true label), only the first term contributes to the loss. The previous equation is the cross-entropy loss for a single input sample. In deep learning (and more generally in machine learning), the loss is usually averaged over a number of samples, called a *mini-batch*. In this case, the loss becomes:

$$L(y, \hat{y}; \theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{c} [y_{i,j} \log(\hat{y_{i,j}}) + (1 - y_{i,j}) \log(1 - \hat{y_{i,j}})] \tag{2.7}$$

Now we know how to compute the model's errors with respect to reference values. The last thing to cover is the design of a learning procedure, that is how to modify the model's parameters in order to reduce future errors.

The Learning Algorithm In Deep Learning, the most popular technique for minimizing the loss function is **gradient descent**. The basic idea is that the gradient of the loss function with respect to the model parameters gives us the direction of maximum increase of the loss. Hence, by updating the parameters in the opposite direction, we are effectively minimizing the error. Formally, the learning rule is the following:

$$\theta^{k+1} = \theta^k - \alpha \nabla L(y, \hat{y}; \theta) \tag{2.8}$$

Where α is a hyper-parameter of the algorithm that controls the step size in the direction of steepest descent. It is called the **learning rate** in the literature. This parameter is of fundamental importance in the optimization procedure, as its value often makes the difference between convergence and divergence of the minimization process. The previous was the most basic

[7] For instance, in a ten class classification problem, class 6 would be represented as $y_6 = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]$.

version of gradient descent. Other techniques have been developed in recent years to overcome the major limitations of this scheme. The momentum [26] method provides a sort of short-term memory of recent updates to the optimizer and it helps smoothing the trajectory of minimization and improving speed of convergence. More recent methods like Adam [14] are more sophisticated, as they employs per-parameter adaptive learning rates.
Now we know how to modify the network's parameters in order to minimize the loss function, but how do we compute the gradients? It turns out there is a very computationally efficient algorithm which allows us to compute the partial derivatives of the loss with respect to the model's parameters. This algorithm is called **backpropagation**.

The BackPropagation Algorithm The BackPropagation algorithm [20] computes partial derivatives in a feed-forward neural network[8].
The reference value y is defined with respect to the output layer of the network, so it is easy to compute the gradient of this layer. Hidden layers instead do not have target values, so their gradient cannot be computed directly. But we know that hidden layers influence the output of the last layer, because they provide the representation on top of which the output layer is built. BackPropagation uses this fact to recursively compute the gradients starting from the output layer and going backwards to the first hidden layer. Hence the name BackPropagation. In particular, BackPropagation cleverly uses the chain rule of calculus to compute the partial derivatives in previous layers. The chain rule of calculus basically states that if we have a nested function $z = f(y) = f(g(x))$, the derivative of z w.r.t. x is equal to:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

This is similar to the structure of equation 2.5. The full learning procedure of a MultiLayer-Perceptron is showed in Algorithm 1.

2.2.2 Convolutional Neural Networks

As we saw in the previous section, in the standard feed-forward net each layer is composed by an affine mapping followed by some non-linearity. But this is not the only way in which layers can be designed. There exists several kinds of feed-forward architectures, and in this section we will see what is arguably the most popular and successful example of specialization: the Convolutional

[8] Although this technique is mainly used in the context of neural networks optimization, the algorithm is general and can be used to differentiate arbitrary functions

Algorithm 1 MultiLayer-Perceptron learning algorithm.
The **forward** function computes the output of the network given input x: $\hat{y} = f(x;\theta)$. The **backward** function computes the gradients of the loss function with respect to the model parameters. It starts with the gradient of the output layer, and then it propagates it backwards. The **update** function updates the parameters of the network according to the gradient descent update rule. The three functions must always be called in the exact order: forward $\rightarrow$ *backward* $\rightarrow$ *update*

Require: net, the network
Require: (x, y), (input, target output)
function FORWARD(net, x, y)
  $h[0] = x$ ▷ h is a tensor holding the outputs of all layers
  for $k = 1, \ldots, d$ **do** ▷ d is the depth (number of layers)
    $a[k] = net.W[k]h[k-1] + net.b[k]$
    $h[k] = \sigma(a[k])$
    $net.a[k] = a[k]$
  end for
  $\hat{y} = h[l]$
  $J(\hat{y}, y; \theta) = L(\hat{y}, y; \theta) + \lambda\Omega(\theta)$ ▷ $\lambda\Omega(\theta)$ is the regularization term
  return $(\hat{y}, J(\hat{y}, y; \theta))$
end function
function BACKWARD(net, y)
  $g = \nabla_{\hat{y}} J(\hat{y}, y; \theta)$ ▷ gradient of the output layer
  for $k = l, \ldots, 1$ **do** ▷ from output to input
    $g = \nabla_{a[k]} J = g \odot f'(net.a[k])$ ▷ gradient before the non-linearity
    $net.gradB[k] = \nabla_{net.b[k]} J = g + \lambda\nabla_{net.b[k]}\Omega(\theta)$
    $net.gradW[k] = \nabla_{net.W[k]} J = gh^{(k-1)T} + \lambda\nabla_{net.W[k]}\Omega(\theta)$
    $g = \nabla_{h^{(k-1)}} J = net.W[k]^T g$ ▷ propagate the gradients below
  end for
end function
function UPDATE(net)
  for $k = l, \ldots, 1$ **do**
    $net.b[k] = net.b[k] - \alpha \cdot net.gradB[k]$
    $net.W[k] = net.W[k] - \alpha \cdot net.gradW[k]$
  end for
end function

Neural Network (CNN).
CNNs are a specialization of the feed-forward network, particularly suited to work with data in which spatial information is important, like time series in one dimension and images in two dimensions. If we have to train a standard feed-forward net to classify images, the first thing we would do is to transform the 2D image into a 1D vector by concatenating together all the pixel values. By doing this we are clearly losing the spatial information relating the pixel to one another. Conv nets instead are capable of fully exploit the spatial correlations between pixels.
More formally, a Convolutional Neural Network is a feed-forward network in which at least one layer uses the convolution operation.

The Convolution Operation

The convolution operation employed in deep learning (also called *cross correlation*) is a linear operation. In the case of two dimensional data, in particular images, the output at pixel (i, j) is a weighted linear combination of a neighborhood around (i, j). The set of weights is called the *kernel*. It is easy to understand this operation by looking at picture 2.4:

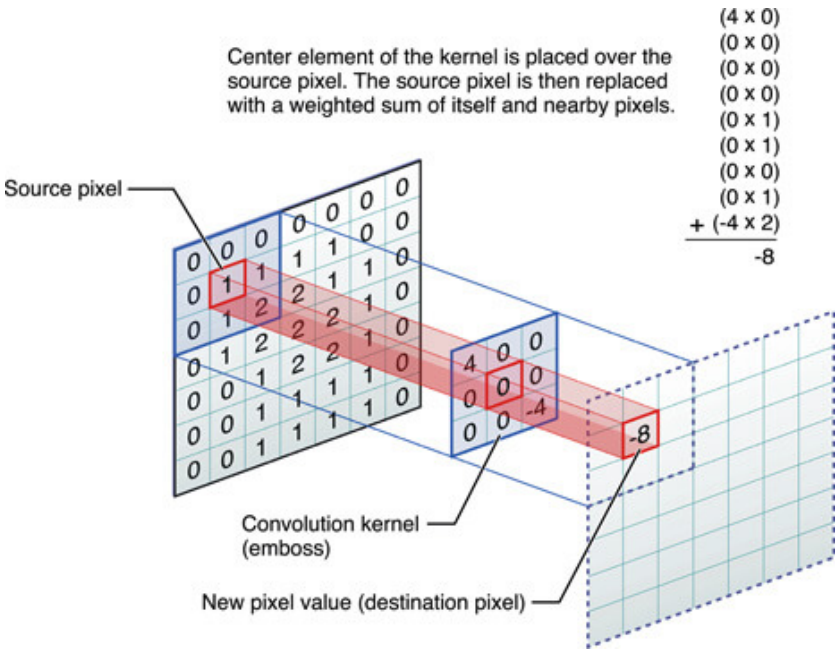

Figure 2.4: The Convolution operation[9].

[9]Image taken from: https://developer.apple.com/library/content/documentation/Performance/Conceptual/vImage/ConvolutionOperations/

More formally, the convolution operation for pixel (i, j) is defined by the following:

$$F(i,j) = (I * K)(i,j) = \sum_w \sum_h I(i+w, j+h)K(w,h) \tag{2.9}$$

Where I is the input image, K is the convolution kernel, and F is the output, also called *feature map*. The full feature map is defined by applying the previous operation for $i = 1, \ldots, w$ and for $j = 1, \ldots, h$, that is, the kernel is a window that slides over the entire image, computing the weighted combination at each pixel location. The kernel can be thought of as a specialized filter, which produces high responses to certain features of the input. For instance, a filter can produce a high response when it sees a vertical edge, or a blotch, and a low response in any other case.

Learning the kernels

The exists many techniques based on convolution and kernel to compute edges or other features in the input image (Ex: Sobel operator). The main difference with those approaches is that in Conv nets, the filters are not handcrafted, but instead the network **learns** the values of the filters. In particular, in each convolutional layer, several kernels are computed in parallel, with each one specializing in the detection of a different feature. As a consequence of the hierarchical nature of deep learning, filters in low level layers specialize in the detection of low-level features such as edges and blobs, while high-level filters specialize in the detection of more high-level features, such as faces and animals.

In the next section we will see the other building block of CNNs, the *pooling* operation.

Pooling in Convolutional Networks

In CNNs, a convolutional layer is usually followed by a pooling layer, that is a way of performing dimensionality reduction and improving the network overall robustness to input transformation. In particular, a pooling layer replaces the output of the network at pixel location (i, j) with a summary statistics of nearby pixels. Many statistics are employed in the deep learning literature (refs). The most popular is arguably Max Pooling (ref), in which the summary statistic is simply that maximum value. The following depition shows how Max Pooling works:

`ConvolutionOperations.html`

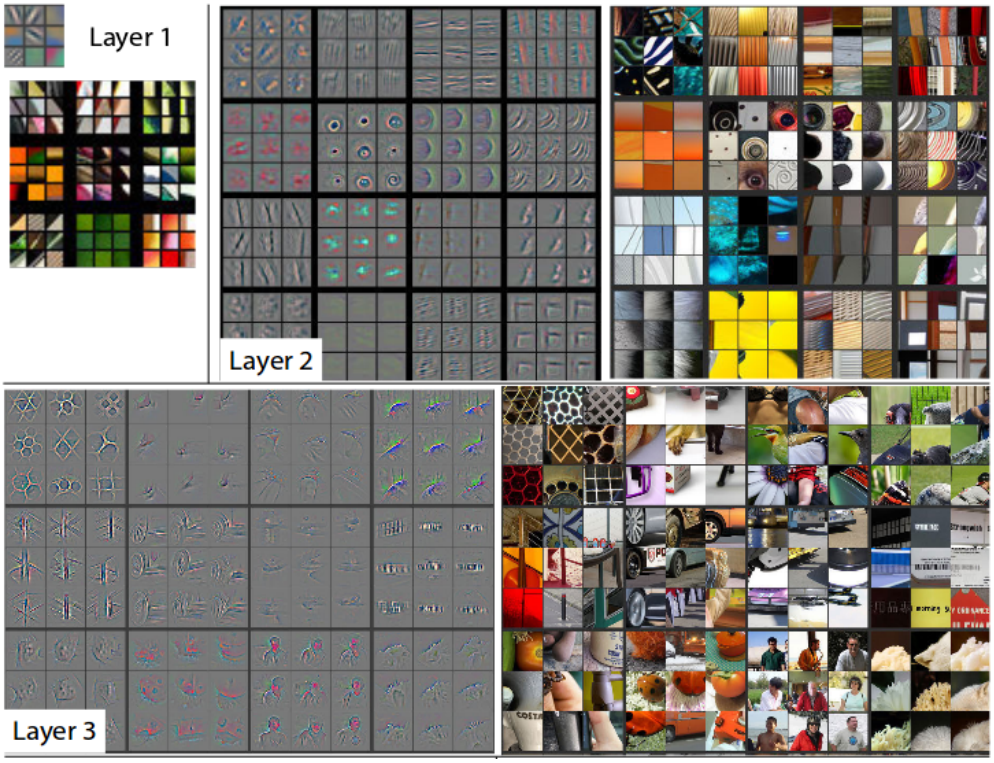

Figure 2.5: Visualizing Conv net filters, taken from [32]

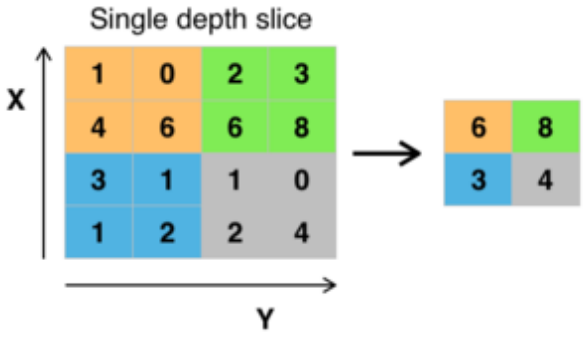

Figure 2.6: Max Pooling operation.

It has been shown (ref) that the spatial pooling operation improves invariance to small translations in the input, i.e. if the input image is translated some pixels to the left or right, the output is still the same. This is a nice property to have when dealing with images, because we aren't usually interested in knowing precisely where a feature, we are only concerned with

whether the feature is actually present or not.

Advantages of Convolutional Networks

Convolutional Neural Networks have some unique features that make them particularly efficient, from both a computational and a computer vision perspective. The three main advantages of CNNs with respect to standard feed-forward networks are:

- **Sparse Interactions**: the feed-forward net is composed of affine layers $Wx+b$ where there is a different parameter for each pair of input-output units. Thus, each layer of the net has a number of parameters that is $O(m \times n)$, where m is the number of input units and n the number of output units. CNNs instead has a number of parameters that is $O(k \times w \times h)$, where k is the number of filters and w, h are respectively width and height of each filter. Since kernels are typically much smaller that the input, CNNs can have less parameters by orders of magnitude.

- **Parameter Sharing**: As we said in the previous point, feed-forward nets have a different parameter for each pair of input-output units. In CNNs instead, each kernel is re-used over the entire input. This can be viewed as a parameter sharing technique, in which we have m kernels with the constraint that they have to have the same parameters. Of course, in practice, we have only one kernel that is used as a sliding window over the input, but thinking of it as a use of parameter sharing can give more insights into how CNNs work.

- **Translation equivariance**: The convolution operation is naturally equivariant to translation, i.e. if the input is translated by a small amount of pixels to the left or right, the output is translated by the same amount. This is clearly a desirable property when analizying images. This operation is not naturally invariant to other input transformations such as scaling and rotation, but CNNs as a whole can achieve such properties with appropriate mechanisms.

Chapter 3

Related Work

3.1 Domain Adaptation approaches

Domain Adaptation is a fundamental problem in machine learning, and the need for techniques that works when $P_s(X, Y) \neq P_t(X, Y)$ is preminent in the majority of practical applications. In computer vision in particular, there is a large number of factors which can cause the domain shift: background, lighting conditions, resolution and scale, position and orientation of the object of interest and so on. Various approaches to address the domain adaptation problem have been proposed in the literature. In the following overview of the domain adaptation literature, we will use the same distinction of [3] into shallow methods and the more recent deep methods, which employs deep learning models, in particular the deep convolutional networks described in Section 2.2.

3.1.1 Shallow Methods

In this context, shallow methods refers to those methods which are based on feature vector representations X extracted from the images with non-deep-learning methods.

Instance Weighting The earliest solutions to the domain adaptation problem go under the name of *Instance Weighting* approaches. The basic idea is that of assigning a different weight to each sample in the computation of the loss. The definition of the weight varies based on the assumptions one makes about the source and target distributions. One possible approach is assuming that the conditional distributions of the labels given the same observation are the same, but clearly the marginal distributions of the observations are

different. Formally:

$$P_s(Y|X = x) = P_t(Y|X = x) \text{ with } P_s(X) \neq P_t(X)$$

This assumption is called *covariate shift* and it is explored in depth in [23]. To solve the covariate shift problem, [23] weights each training instance with $\frac{P_t(X)}{P_s(X)}$, that is, the weight is the ratio between the likelihood of being a target and a source sample.

Feature Space Alignment Another class of techniques that tries to align source features with the target ones. A very simple method in this class is Subspace Alignment (SA) [6], where the alignment is made between the subspaces obtained by PCA reduction:

$$P_s = PCA(X_s, d) \qquad P_t = PCA(X_t, d)$$

$$X_s^{'} = X_s P_s P_s^T P_t \qquad X_t^{'} = X_t P_t$$

And then $X_s^{'}$ and $X_t^{'}$ are used in place of X_s and X_t.

Feature Transformation through metric learning These are semi-supervised domain adaptation methods, as they require at least a limited amount of target labels available. One method worth mentioning is DA-NBNN [29]. The main idea of DA-NBNN is to replace at each iteration the most ambiguous source example of each class by the target example for which the classifier (Naive Bayes Nearest Neighbor (NBNN)) is the most confident for the given class. It does this by iteratively learning a metric of the relatedness between the source and target domains.

3.1.2 Deep methods

The expression deep methods refers to all those approaches which are based either on fixed features extracted from a deep learning model or on the design of a novel deep network architecture which encorporates elements to solve the domain adaptation problem. One method worth mentioning in this respect is the Domain Adversarial Neural Networks (DANN) [8], which obtained state-of-the-art results on several benchmark datasets. It is called adversarial training because a part of the network is optimized in such a way to confuse another part of the network.

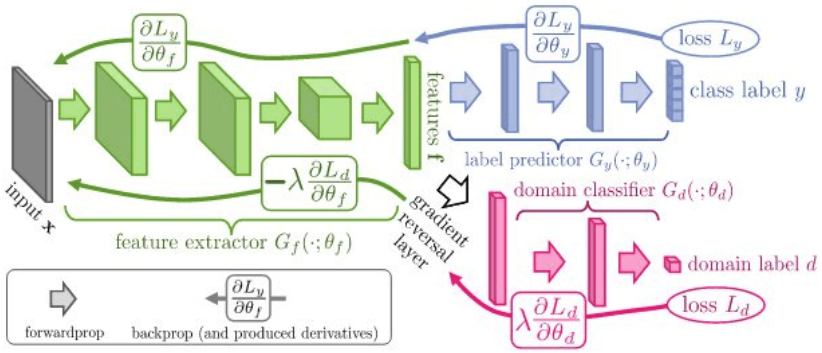

Figure 3.1: DANN Architecture. Image taken from [8].

DANN

Adversarial Training. The networks is composed of three parts:

- **feature extractor**: this part of the network is trained to simultaneously optimize two objetives: learn features that are discriminant for the source classification task, and at the same time learn domain invariant features, that is, features that would increase the classification error of a binary domain classifier.
- **image classifier**: this is the classifier that carries on the main image classification task.
- **domain classifier**: this is the binary classifier that carries on the domain classification task between source and target. It is the adversary of the feature extractor.

Chapter 4

Approach

In this chapter, the Domain Multiplicative-Fusion (DMF) Network is presented. This technique builds upon three methods which are described in the following:

1. We employ a modification of Grad-CAM [22] to generate what we call *domainness maps*, which tell us on a per-feature-map basis which regions in the image are discriminative between source and target domains.

2. Spatial Pyramid Pooling [11] is used to perform pooling at multiple levels in a parallel fashion. This can be useful when localization of the object in the image is an issue. It also generates fixed-length representations regardless of the input size.

3. We combine information in the feature maps with information in the domainness maps multiplicatively, as in [19]. The maps inhibit domain-specific regions (regions that are discriminative for the two domains) and enhance domain-generic regions, thereby reducing the domain shift between source and target.

4.1 Grad-CAM

Motivation At testing time, when a convolutional network looks at an image, there are certain regions that causes higher activations than others, thus contributing the most to the final classification choice. Clearly, we hope such regions to be exactly those containing the object of the class output by the network.

Grad-CAM (Gradient-weighted Class Activation Mapping) [22] is a method for visualizing the spatial locations in an image that contributed the most

to the CNN's classification choice. The technique is applicable to a wide variety of tasks, ranging from image captioning to visual question answering to reinforcement learning, and also to a wide variety of CNN architectures. In the context of this thesis work, we use this technique to produce visuals for an object recognition task. Grad-CAM was designed as a visualization tool to better understand how CNNs reason and to improve their interpretability, which is a fundamental property to have for AI systems that wants to be deployed in the real world.

How it works Although visualizations can be extracted at any layer of the hierarchy, we opted for the last convolutional layer because, as the authors note, it is the best compromise between high-level semantics and spatial localization. The former is a general property of deep learning models: the further the layer from the input, the more high-level and meaningful the features extracted are. The latter is a property of convolutional feature maps, which are designed to retain as much spatial information as possible about the input. Fully-connected layers down the hierarchy do not have this property.
The high-level architecture of the method can be seen in the following figure:

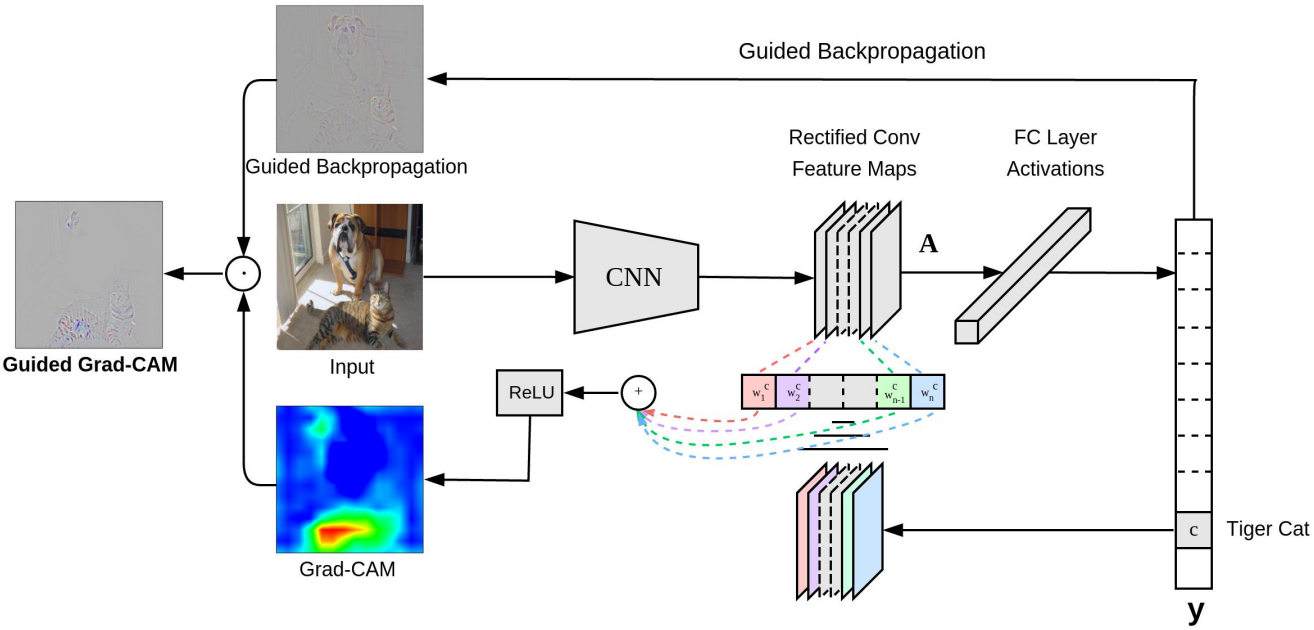

Figure 4.1: Grad-CAM Architecture, taken from [22]

Basically, the input image is forward propagated through the network in

order to compute the softmax scores of the last layer, which we call $y^c = f(x)$, where $f()$ is the input-output network mapping. If we were training the network, at this point we would have computed the gradients of the last layer neurons with respect to some loss function, like cross-entropy. Instead, in Grad-CAM, we manually create these gradients. In particular, we set them to one for the neuron corresponding to the class of interest, while all the other neurons are set to zero.

$$\frac{\partial y^c}{\partial \theta_{output}} = [0, 0, \ldots, 1, 0, 0, \ldots]$$

Then, we backward propagate these gradients through the network until the last convolutional layer, where the Grad-CAM map is created. Assuming that the last convolutional layer feature maps are in $\mathbb{R}^{K \times H \times W}$, where H and W are respectively the height and the width of each feature map, and K is the number of feature maps (depth). The map that will be generated is thus in $\mathbb{R}^{H \times W}$. The generation procedure has multiple steps:

1. Compute the weight of each feature maps. We want a number measuring how much each filter contributed to the final score. It is therefore natural to compute this number as the derivative of the output gradients with respect to the weights of the filter. These derivatives are then global average pooled in order to get a single number:

$$w_k^c = \frac{1}{Z} \sum_{i=1}^{h} \sum_{j=1}^{w} \frac{\partial y^c}{\partial F_{i,j}^k} \tag{4.1}$$

   w_k^c is the weight of feature map k with respect to target class c, while F is the feature map itself. We can see that the derivative is positive if an increase of the value of the pixel $F_{i,j}^k$ yields an increase of the value of y^c.

2. The Grad-CAM map is obtained by performing a rectified weighted linear combination of the forward feature maps:

$$M^c = ReLU\left(\sum_{k=1}^{K} w_k^c F^k\right) \in \mathbb{R}^{H \times W} \tag{4.2}$$

The $ReLU()$ function simply set to zero all the negative values. The Grad-CAM technique uses this function to retain only the values that have a *positive* influence on the class decision. Once the Grad-CAM map is generated, it is then upsampled to the pixel space using bi-linear interpolation, then a heatmap is generated from the values and the heatmap is point-wise multiplied with the output of the Guided-BackPropagation procedure [25], to produce the final result called Guided-GradCAM.

4.1.1 Grad-CAM for domain localization

To perform domain localization with Grad-CAM, we replaced the 1000-units softmax of ImageNet CNNs with a single sigmoid unit, and we fine-tuned the network to perform a binary classification between source and target samples. Our rationale is that by applying Grad-CAM to this binary network we are able to determine which regions in the image are *domain-specific* (information contained only in one domain and not in the other) or *domain-generic* (information shared between domains). Having this clue on a *per-feature-map* basis, we are able to make source and target more similar at the representational level, by removing domain-specific locations and enhancing domain-generic ones.

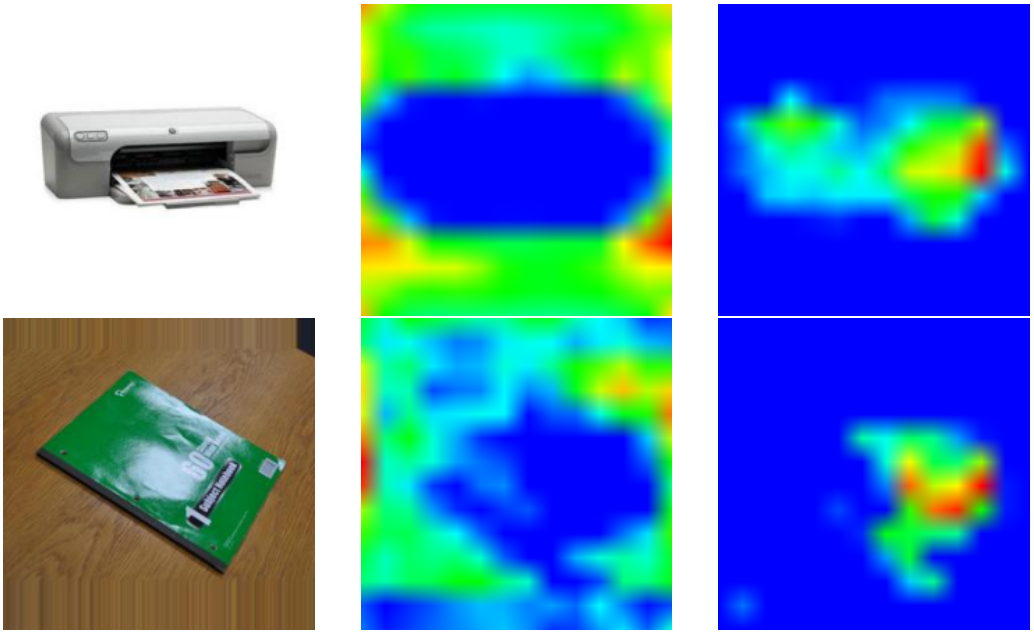

Figure 4.2: Examples of domainness maps. Both rows have: (left) original image, (center) domain-specific map, (right) domain-generic map. We can see that the domain-specific map captures information contained in one domain but not in the other, in this case the background. Domain-generic maps instead captures information shared between domains, in this case the objects themselves.

Our domainness maps are actually generated using only a subset of the full Guided-GradCAM procedure. In particular, we used only Grad-CAM maps (without Guided-BackPropagation), and we retained the full *per feature-map activations*, instead of summing them across depth. Namely, in our

method, equation 4.2 becomes:

$$M^c = ReLU\left(w_k^c F^k\right) \in \mathbb{R}^{K \times H \times W} \tag{4.3}$$

That is, instead of summarizing into a single map, we retain information about how each filter contributed to the decision of the class. This is crucial, as it enables us to enhance and inhibit filters with a greater precision.

4.2 Spatial Pyramid Pooling

Motivation Convolutional Networks can be thought of as composed by two parts: a feature extractor followed by a classifier. The former is a hierarchy of convolutional layers—the latter is a Multi-Layer Perceptron on top of the former's output. Each convolutional layer has three sequential stages: the convolution operation, the non-linear activation, and the non-linear down-sampling operation (pooling). The pooling operation is needed to reduce the dimensionality of the data and to improve robustness with respect to input distortion. Usually, max pooling is used, that is a sliding window of a certain size in which for each window, the pixel with the max value is taken as output. Other summary statistics were employed in the literature, such as average pooling or sum pooling, but max pooling was found to work best in the context of object recognition [21]. After the last pooling layer, the resulting 3D tensor of dimension $C \times H \times W$ is flattened into a 1D tensor of dimension $C * H * W$, and a series of fully-connected layers is placed on top of this layer. The are at least two problems with this architecture:

- Convolutional layers can deal with images of different sizes, but they will also produce outputs of different sizes. Fully-connected layers instead, only accept inputs of fixed size. Thus, the previous architecture cannot exploit this strength of convolutional layers.
- The designer should carefully choose the dimension of the pooling layers because, as we have seen in our experiments, this can have a major impact on the overall performance of the model. In particular we have seen that the last pooling operation is of particular importance when the object is translated or scaled by a large amount of pixels.

SPP-Net Spatial Pyramid Pooling (SPP) is a method that was used extensively by the computer vision community before the deep learning revolution. This paper [11] introduced the technique in the context of CNNs. Basically, a Spatial Pyramid Pooling operation is composed by multiple max pooling

operations performed at different window sizes and then concatenated together in the depth dimension. This seemingly simple modification of the standard pooling layer has profound implications, such that:

- With a SPP layer as the last layer before the MLP, the network can take in input images of arbitrary sizes, scales and aspect ratios.
- The network is much more robust to translation and scaling of the input, because a pooling done at multiple levels is much able to capture such variations.
- The authors shown a small but consistent classification accuracy improvement over a large range of architectures and datasets.

Figure 4.3 is a depiction of how this layer works.

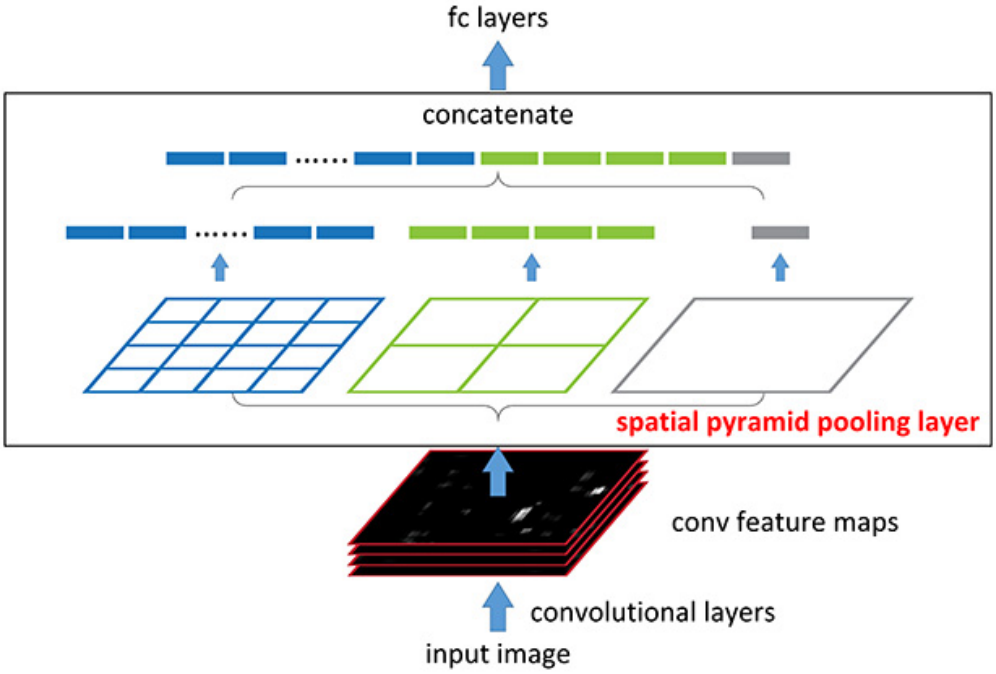

Figure 4.3: Spatial Pyramid Pooling layer. Image taken from: [11]

The pooling operation employed at each level is slightly different from the one we have described before. In particular, the one used here is called *Adaptive Pooling*: instead of choosing the window size and the stride, we

directly choose the output dimensions we want (say $n \times n$). From this, the window and the stride are automatically computed as $window = \lceil \frac{a}{n} \rceil$ and $stride = \lfloor \frac{a}{n} \rfloor$, where a is the dimension of the last conv layer feature maps. As a practical example, the VGG16 network [24] last conv layer feature maps have dimension $512 \times 14 \times 14$[1]. If we want an adaptive max pooling of dimension (4×4), the window and the stride are computed as $window = \lceil \frac{14}{4} \rceil = 4$ and $stride = \lfloor \frac{14}{4} \rfloor = 3$. Figure 4.3 has a SPP layer with three levels: $(4 \times 4)(2 \times 2)(1 \times 1)$. We can easily compute the (fixed) output dimension of this layer. For instance, with the VGG16 network, the output size is: $(512 * 4 * 4) + (512 * 2 * 2) + (512 * 1 * 1) = 10752$, so we can place a fully-connected layer on top of this SPP and we can be sure that the dimension will be the same regardless of the input size.

As a final note, during our experiments we observed that the size of the window in the pooling operation is an important factor to take care of when dealing with a domain shift problem. Usually, one takes whatever dimension was chosen for the baseline network (AlexNet or VGG) and keeps it fixed. However, we adopted the original SPP strategy with the aim of testing different window sizes and we effectively verified that results can vary significantly based on this parameters. As a consequence, this parameter should be carefully optimized.

4.3 Domain Multiplicative-Fusion

Motivation The architecture of our network, which we called Domain Multiplicative-Fusion (DMF) is inspired from this work [19]. In particular, the authors of [19] explored two different ways to combine information from multiple sources into the same CNN architecture, in the context of action recognition. The first technique, which they called *Feature Amplification*, consists in an element-wise (hadamard) product between two sources of information: the first was the CNN feature maps and the second was the optical flow [7] information. The rationale was that through the multiplication, CNN feature maps information would detect important features for the action recognition task, and optical flow information would amplify (cancel out) important regions in the feature maps with respect to the task of detecting motion. Then the author proposed another method to combine different sources, which they called *Multiplicative Fusion*. This method combines the

[1] these are the output dimensions when the input image is 224×224. For bigger (smaller) images, a would be greater (less) than 14. However, using adaptive pooling, the output dimension will be $512 \times 4 \times 4$ regardless of the input size.

feature maps coming from two different CNNs into a merged representation which is then propagated through the fully-connected layers. In particular, they employed a layer that performs a linear combination of feature maps (the coefficients of the combination that are learnable parameters).

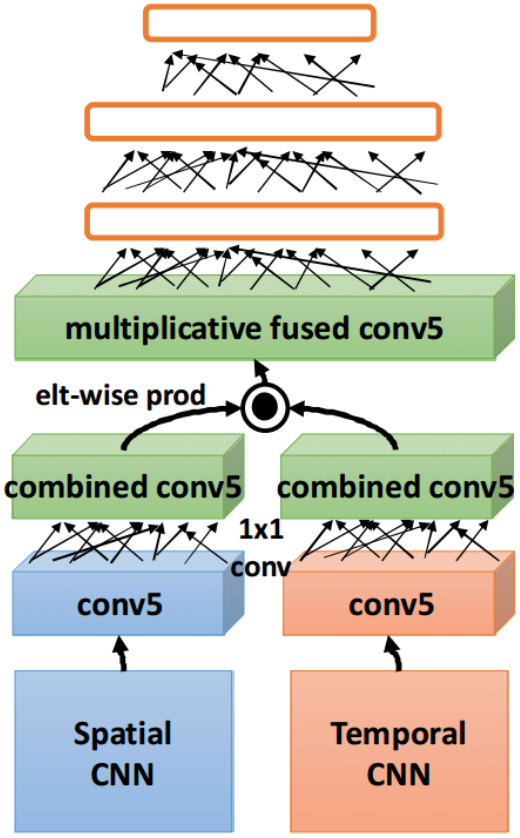

Figure 4.4: Multiplicative-Fusion architecture. Image taken from [19]

4.3.1 Architecture Design

Although we draw the idea of combining knowledge through multiplication from [19], our architecture is quite different from theirs, as is the learning task. First of all, let's recap our domain adaptation task and setting. We have two datasets which were drawn from two different distributions: the *source* dataset (X_s, Y_s) was drawn from $P_s(X, Y)$, and the *target* dataset (X_t) was drawn from $P_t(X)$, with $P_s \neq P_t$. Our learning task is to design a model that learns how to infer Y_t given (X_s, X_t, Y_s). Our method can be summarized in the following way:

1. We train a CNN with a binary classifier on top of it to discriminate between source and target samples. The source samples are assigned a label of 0, and the target samples a label of 1. The CNN ends with a single sigmoid unit, which outputs the probability of the sample being a target sample:

$$P(Y = 1|X) = \hat{y}$$

   This model is trained using the binary-cross-entropy loss function and stochastic gradient descent.

2. Our modification of the Grad-CAM technique is used to generate activation maps from the previous CNN. In particular, for each image, we generate a map of the regions that would make the classifier change its decision, from source to target or vice-versa. The activations are of the last conv layer, which has a more compact and meaningful representation, as the author of Grad-CAM also pointed out.

3. The maps are integrated into a final CNN that performs object classification. The final architecture can be seen in figure (FIG). From the input layer to the last conv layer, the architecture is the same as a standard CNN. Then, the output feature maps are replicated,

We also combine two different networks in our model, but the two are trained separately, with the binary network trained first and then used to generate Grad-CAM maps, and the object classification network trained after using the maps.

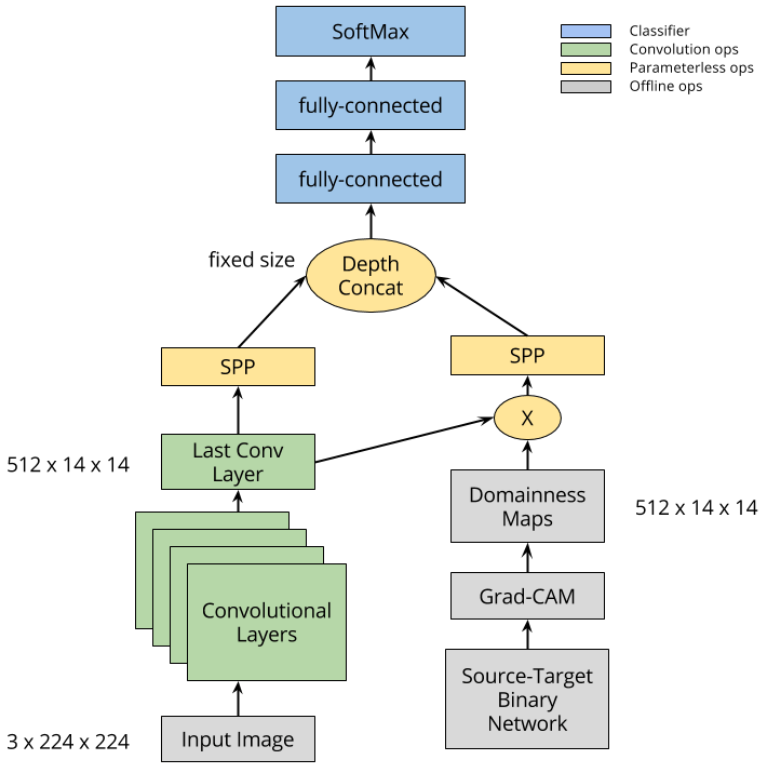

Figure 4.5: Domain-Multiplicative Fusion architecture.

Chapter 5

Experiments

5.1 Implementation

We use Torch7 library [2] for implementation. The DMF building blocks (apart from standard layers) are the element-wise multiplication and the depth contact operation, which are already available in Torch7. Regarding Grad-CAM, we start from the code provided by the authors [10] and we modify it as discussed in Section 4.1.1. We implement the Spatial Pyramid Pooling layer as an extension of Torch7, and we build the computational graph to compose the building blocks into the overall end-to-end architecture.

For each experiment setup, we train and test three different network configurations. All three variants starts from a baseline network pretrained on the *ImageNet* dataset [28]. This procedure of starting with a network pretrained on a dataset and fine-tuning it on a new-but-similar dataset is called *Transfer Learning* [30] in the literature. In the first configuration (which we call Softmax), we replace only the last layer of the network with a new layer with dimension equal to the number of classes in the dataset. We train only this last layer, keeping all the other parameters frozen during training. In the second network (Spp), we remove all the fully-connected layers, as well as the last Max Pooling layer. We replace the latter with a Spatial Pyramid Pooling layer, and we add new fully-connected layers on top of it. The third network is DMF, whom architecture is discussed in Section 4.3. Note that in Spp and DMF the substitution of the fully-connected layers is necessary, as in a feed-forward network the removal of layer i constrains us to remove also layers $i+1$ to n.
With this configuration of networks, we are able to verify the effectiveness of each component of the Domain-Multiplicative Fusion Network. In one

experiment setting, we also compare our technique against state-of-the-art approaches.

5.2 Experiment settings

5.2.1 Baseline Networks

In our experiments we use two baseline networks. The first is *Alexnet* [1], winner of the ILSVRC 2012 competition and the responsible for the recent deep learning revolution in computer vision. The paper introduced many interesting innovations, most notably the use of Graphics Processing Units (GPUs) to train neural networks with millions of parameters at large scale, a method to reduce overfitting known as Dropout, and the use of the Rectified Linear Unit (ReLU) activation function. The second network is VGG16 [24], winner of the ILSVRC 2014 competition. VGG16 is much deeper than Alexnet, in fact Alexnet has 5 convolutional layers and 3 fully-connected layers, while VGG16 has 13 convolutional layers and 3 fully-connected layers. The use of two baseline networks improves the reliability of our findings, as this decreases the probability that our results were due to the use of a specific architecture.

5.2.2 Training procedure

Each network is trained using the same set of hyperparameters, established by using a validation set to search over a grid of possible values. The parameters of the first f layers are frozen, where $f = 22$ for AlexNet (Softmax), $f = 38$ for VGG16 (Softmax), $f = 5$ for AlexNet (Spp and DMF), $f = 12$ for VGG16 (Spp and DMF). The parameters of layers $f + 1$ to g are not frozen, but they are set with a small learning rate $\alpha_{f+1,g} = 10^{-5}$ (Spp and DMF) and $\alpha_{f+1,g} = 0$ (Softmax). g is the number of layers that are retained from the ImageNet baseline. The parameters of layers $g + 1$ to n have a higher learning rate $\alpha_{g+1,n} = 10^{-3}$ (Softmax) and $\alpha_{g+1,n} = 5 \times 10^{-4}$ (Spp and DMF). Also, for all the layers that are trained from random initialization, a weight decay of $w = 5 \times 10^{-4}$ is used as a regularizer. In Spp and DMF, fully-connected layers have dimension $2048 \rightarrow 2048 \rightarrow 15$ for AlexNet and $4096 \rightarrow 2048 \rightarrow 15$ for VGG16, and they uses dropout with $p = 0.6$.

The optimizer used in the training procedure is Stochastic Gradient Descent with Momentum [26]. The momentum parameter is set to 0.9, with nesterov momentum [18] enabled. We use a batch size of 256 for AlexNet and 16 for VGG16 (due to the high memory footprint of this network). Training

was performed in parallel on multiple NVIDIA Titan X GPUs.

5.2.3 Datasets and Metrics

We evaluate our DMF on two variants of the iCub World dataset [27], which we call respectively iCW-translation and iCW-scale. Both datasets have two domains. As the names suggest, in the former case the shift between the two domains is caused by large object translations, while in the latter it is caused by scale variations. It is worth noting that, as discussed in Section 2.2.2, CNNs are known to handle some degree of scale and translation invariances. However, using a network structure specifically designed to handle such invariances have been shown to be more successful in many tasks [11]. In both datasets, one domain is composed by 4500 images and the other one by 3000 images. These are rather small datasets for data-hungry deep learning models, hence the choice of starting from ImageNet baselines. For iCW-translation, we also compare against prior state-of-the-art approaches.

To measure performance, we train each network over one domain and use the other domain as the test set, measuring classification accuracy. In order to improve statistical robustness, we run all the experiments 5 times, and we report mean and standard deviation of the results.

5.2.4 The iCub World dataset

The iCW Transformations dataset [27] contains images with objects from 15 categories, acquired through a robot camera. Each object is acquired while undergoing isolated visual transformations, in order to study invariance to real-world nuisances. A human operator moves the object holding it in the hand and the robot tracks it by exploiting either motion cues or depth cues. Different datasets are available depending on the object transformation carried out by the human. We will use two different variants of the iCW Transformations dataset:

- **Translation**: The human moves in a semi-circle around the iCub robot, keeping approximately the same distance and pose of the object in the hand with respect to the cameras. We use only images taken at the extreme of the semi-circle, thus the object appears either at the left of the image or at the right. Another cause of domain shift is that the background changes dramatically, while the object appearance remains the same.

- **Scaling**: The human moves the hand holding the object back and forth, thus changing the object's scale with respect to the cameras. In this case, the background remains almost the same, while the object appearance undergoes large changes due to the large scaling factor.

For each of the two variations the classes are perfectly balanced, namely each one has the same number of samples. Each dataset has two domains, hence it defines two domain adaptation tasks: the first is train on domain 1 and test on domain 2 and the second is the opposite. In figure 5.1, we can see a random sample of images from the dataset.

Figure 5.1: Random sample from the iCubWorld dataset.

Domain Shift The first thing to do is to verify if the iCW dataset actually represents a domain adaptation task. That is, we should measure the domain shift in the dataset. In the domain adaptation literature, this is usually done in the following way: we train a classifier on 80% of the source domain, and we test it both on the remaining 20% and on the target domain. The absolute difference of performance gives a rough quantitative measure of the domain shift. Intuitively, the more the performance difference, the more the two datasets were generated from different distributions. The following

process is employed to measure this domain shift. First, image features are extracted at the second fully-connected layer of the VGG16. This gives a feature vector of 4096 entries for each image. We then run a Linear Support Vector Machine classifier [5] on top of these features. Results for both iCW translation and iCW scale are shown in Table 5.1.

	S (80%) → S (20%)	S (80%) → T	Difference
Left 1 → Left 2	99.67	59.23	40.44
Left 2 → Left 1	99.83	62.47	37.36
Scale 1 → Scale 2	100.00	26.54	73.46
Scale 2 → Scale 1	99.67	40.98	58.69

Table 5.1: iCW domain shift measure. S stands for Source. T stands for Target. $X \rightarrow Y$ means that the SVM is trained on X and tested on Y.

In both datasets, there is a clear domain shift between the two domains. Also, by looking at Table 5.1, we can also see for instance that the task Scale 1 → Scale 2 is much more difficult than the opposite one. This might be because Scale 2 images contains information that helps the classifier also on Scale 1, but the opposite is not true.

Having seen that there is indeed a domain shift between the domains, in the next paragraph we compare our method both with standard transfer learning techniques and with the state-of-the-art Domain-Adversarial Networks [8].

Results

In the following tables we report the results of our experiments. We can see that our method DMF outperforms all the other techniques by a significant margin.

	Left 1 → Left 2	Left 2 → Left 1	Average
Softmax	50.41 ± 0.98	54.01 ± 0.59	52.21
Spp	63.55 ± 1.49	61.15 ± 0.69	62.35
DANN	63.20	42.93	53.07
DMF	$\mathbf{67.35 \pm 0.97}$	$\mathbf{61.75 \pm 0.54}$	**64.55**

Table 5.2: iCW Translation Alexnet Results.

	Left 1 → Left2	Left 2 → Left 1	Average
Softmax	62.15 ± 0.94	64.74 ± 0.92	63.44
Spp	75.14 ± 1.09	75.91 ± 1.26	75.53
DANN	76.43	54.60	65.52
DMF	$\mathbf{78.17 \pm 0.66}$	$\mathbf{77.30 \pm 0.80}$	**77.73**

Table 5.3: iCW Translation VGG16 Results.

	Left 1 → Left2	Left 2 → Left 1	Average
Softmax	18.20 ± 0.71	27.45 ± 1.23	22.83
Spp	25.77 ± 0.88	30.52 ± 0.60	28.15
DMF	$\mathbf{29.68 \pm 1.52}$	$\mathbf{31.43 \pm 0.49}$	**30.56**

Table 5.4: iCW Scale Alexnet Results.

	Left 1 → Left 2	Left 2 → Left 1	Average
Softmax	30.85 ± 0.67	49.88 ± 0.73	40.36
Spp	35.69 ± 1.08	49.85 ± 1.38	42.77
DMF	$\mathbf{42.16 \pm 1.09}$	$\mathbf{50.69 \pm 0.96}$	**46.42**

Table 5.5: iCW Scale VGG16 Results.

Chapter 6

Comparison

In the ICW dataset, our method was able to achieve state-of-the-art performance with respect to the Domain-Adversarial Neural Network (DANN). This was due in part to the fact that we discovered that greater performance can be achieved by carefully tuning the dimension of the last pooling layer, and in part to the improvement of the DMF that was capable of providing a representation to the higher layers in which the two domains where made more similar. We also argue that tuning the dimension of the latest pooling layer can be an effective way of incorporating *prior knowledge* about the dataset at hand. For instance, if we know that the object will cover the majority of the image and there will not be much noise, we can go with a smaller window size to identify more details of the object. Conversely, if the object is but a small part of the image, and there is a lot of noise, like background etc., a bigger window size might be appropriate, in order to perform disturbance rejection.
Another thing we noted during our experiments, that is related to the previous point is that DMF needs the dimension of the last pooling layer to be carefully tuned. In particular, we noted that if this is not the case, than the DMF extension to the original architecture is not always able to provide an improvement in terms of performance. Also, source and target domains need to be *different but related*. We tested our method with domains for which the domain shift was very large, and our method performed poorly. However, this is also true for the other state-of-the-art methods for domain adaptation.
Another point of comparison with the state-of-the-art regards the size of the model: that is, the number of parameters. The DMF architecture has a much smaller number of parameters w.r.t. DANN. If we take for example the AlexNet architecture, which DANN authors extended in their paper, the number of parameters is huge, in the order of 70 million parameters only in the fully-connected part of the architecture. The DMF architecture that

we employed was about 13 million parameters in the fully-connected part, that is about a fifth. This leads to a much lesser memory consumption and greater efficiency from a computational view-point. These are important considerations given the ever growing size of deep learning models and from an industry perspective.

Chapter 7

Conclusions

We designed a method to tackle the increasingly important problem of domain adaptation, that is the problem of taking a machine learning classifier trained on a dataset and making it work on a different (but related) dataset, for which no labels are available. We described the foundations upon which our method is built, and state-of-the-art approaches present in the literature. We provided experimental evidence across a range of datasets and network architectures that our method is an improvement over standard architectures, and in some cases also achieved state-of-the-art performance, while using very few parameters with respect to other methods. We argue that our work points out the importance of localizing *domain-generic* and *domain-specific* regions at a representational level, and that the use of this information can make the source representation more similar to the target representation.

7.1 Lessons learned

We performed many experiments, and tried lots of approaches, and we learned many things along the way. One thing is that if we carefully tune the dimension of the latest pooling layer, we can achieve much greater performance (in fact, in some settings, we achieve a 10% improvement in test accuracy). Another thing is that with deep learning model the size of the dataset is really a fundamental issue: methods that work well when the dataset is big enough often do not work at all when the dataset is small. One should carefully tune the model capacity with respect to the dataset at hand, because with deep networks, overfitting is often around the corner. We also learned that domain adaptation is a hard problem that is yet to be solved: despite the huge successes that deep learning methods have achieved in computer vision over the last few years, a model trained on one dataset and tested on a different

one continues to yield very low performance. Some also argue (ref rethinking generalization) that deep networks do not in fact learn semantically meaningful features, but instead they simply memorize entire datasets. With the advent of deep learning, we traded interpretability for performance, and we still know very little about how these models really work. Much more work needs to be done in deepen our understanding of deep neural networks.

7.2 Future work

We think that a promising future direction of this work would be to make the *domainness maps* created by a generative model: that is, by replacing the use of the Grad-CAM procedure with a generative network that is embedded in the network. In this way, instead of having two learning processes that are carried out in a sequential manner, one would have a single end-to-end architecture jointly trained in one step. Besides reducing the computational burden of our method, we also argue that this would increase the quality of the produced maps.

Bibliography

[1] Geoffrey E. Hinton Alex Krizhevsky Ilya Sutskever. "ImageNet Classification with Deep Convolutional Neural Networks". In: (2012).

[2] Ronan Collobert, Koray Kavukcuoglu, and Clement Farabet. "Torch7: A Matlab-like Environment for Machine Learning". In: *NIPS* (2011).

[3] Gabriela Csurka. "Domain Adaptation for Visual Applications: A Comprehensive Survey". In: *CoRR* abs/1702.05374 (2017). URL: http://arxiv.org/abs/1702.05374.

[4] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems* 2 (1989).

[5] Rong-En Fan et al. "LIBLINEAR: A library for large linear classification". In: *Journal of machine learning research* 9.Aug (2008), pp. 1871–1874.

[6] B. Fernando et al. "Unsupervised visual domain adaptation using subspace alignment". In: *IEEE International Conference on Computer Vision (ICCV)* (2013).

[7] David J. Fleet and Yair Weiss. "Optical Flow Estimation". In: ().

[8] Y. Ganin et al. "Domain-Adversarial Training of Neural Networks". In: *Journal of Machine Learning Research* 17 (2016).

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

[10] *Grad-CAM Torch7 Implementation*. 2016. URL: https://github.com/ramprs/grad-cam.

[11] Kaiming He et al. "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition". In: *CoRR* abs/1406.4729 (2014). URL: http://arxiv.org/abs/1406.4729.

[12] P. Jackson. *Introduction to expert systems*. Addison-Wesley Pub. Co.,Reading, MA, 1986.

[13] Jing Jiang. “A Literature Survey on Domain Adaptation of Statistical Classifiers”. In: (2008).

[14] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2014). URL: http://arxiv.org/abs/1412.6980.

[15] Van Der Maaten L. and Hinton GE. “Visualizing High-Dimensional Data Using t-SNE”. In: *Journal of Machine Learning Research* 9 (2008). URL: http://prlab.tudelft.nl/content/visualizing-high-dimensional-data-using-t-sne.

[16] Y. LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (1989), pp. 541–551.

[17] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521.7553 (May 2015), pp. 436–444. ISSN: 0028-0836. DOI: 10.1038/nature14539.

[18] Y. Nesterov. “A method of solving a convex programming problem with convergence rate $O(1/k^2)$”. In: *Soviet Mathematics Doklady* 27 (1983).

[19] Eunbyung Park et al. “Combining multiple sources of knowledge in deep CNNs for action recognition”. In: *2016 IEEE Winter Conference on Applications of Computer Vision, WACV 2016*. Institute of Electrical and Electronics Engineers Inc., May 2016. DOI: 10.1109/WACV.2016.7477589.

[20] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (1986).

[21] Dominik Scherer, Andreas Müller, and Sven Behnke. “Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition”. In: *Proceedings of the 20th International Conference on Artificial Neural Networks: Part III*. ICANN’10. Thessaloniki, Greece: Springer-Verlag, 2010, pp. 92–101. ISBN: 3-642-15824-2, 978-3-642-15824-7. URL: http://dl.acm.org/citation.cfm?id=1886436.1886447.

[22] Ramprasaath R. Selvaraju et al. “Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization”. In: *CoRR* abs/1610.02391 (2016). URL: http://arxiv.org/abs/1610.02391.

[23] Hidetoshi Shimodaira. “Improving predictive inference under covariate shift by weighting the log-likelihood function”. In: *Journal of Statistical Planning and Inference* 90 (2000).

[24] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556.

[25] Jost Tobias Springenberg et al. "Striving for Simplicity: The All Convolutional Net". In: *CoRR* abs/1412.6806 (2014). URL: http://arxiv.org/abs/1412.6806.

[26] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 2013, pp. 1139–1147. URL: http://proceedings.mlr.press/v28/sutskever13.html.

[27] *The iCub World Dataset*. 2013. URL: https://robotology.github.io/iCubWorld/#icubworld-transformations-modal.

[28] *The ImageNet Dataset*. 2017. URL: http://www.image-net.org/.

[29] T. Tommasi and B. Caputo. "Frustratingly Easy NBNN Domain Adaptation". In: *IEEE International Conference on Computer Vision (ICCV)* (2013).

[30] Lisa Torrey and Jude Shavlik. "Transfer Learning". In: *Handbook of Research on Machine Learning Applications* (2009).

[31] Vladimir N. Vapnik. *Statistical Learning Theory*. Wiley-Interscience, 1998.

[32] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *CoRR* abs/1311.2901 (2013). URL: http://arxiv.org/abs/1311.2901.