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

Acknowledgements

Acknowledgements section.

Abstract

Abstract Saenko et al. 2010, yeah Yoo et al. 2016, moar cite Selvaraju et al. 2016

Contents

List of Figures

List of Tables

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

Chapter 2

Background

This chapter provides a brief overview of the problems addressed in this work, as well as the main technologies upon which our approach is designed. Section 2.1 is a description of the domain adaptation problem in the context of machine learning, in which the main challenges of the field are highlighted. Section 2.2 is an introduction to the basics of Deep Learning, a branch of machine learning concerned with the study and the design of Artificial Neural Network (ANN) models. This section draws heavily on Goodfellow, Bengio, and Courville 2016.

2.1 Domain Adaptation

Introduction At the core of the majority of machine learning (ML) techniques, there is the optimization of some error (also called loss) function $L(y, \hat{y}; \theta)$, that is a function of the reference output y, the output of the model $\hat{y}$, and the parameters of the model θ.
Although machine learning draws a lot from optimization methods, there is a seemingly small but 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 this will also minimize the overall error function (which we don't know).

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

In this framework, what we really care about is **generalization**, that is the ability of an algorithm to obtain a good performance on samples it has never

seen before (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})[1] (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 majority of algorithms operate under this assumption, which is usually a fairly good approximation. But the are cases in which this assumption does not hold, and this causes a significant drop in performance between training and testing.

Motivation Domain Adaptation Jiang 2008 consists in the design of algorithms that work even when the i.i.d. assumption does not hold, 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 with respect to classical ML. In particular, the training distribution is called the *source domain* D_S, while the testing distribution is called the *target domain* D_T. Typically, we assume that the source data is abundant and labeled, while the target data is only partially labeled, 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 work in particular focuses exclusively on the unsupervised setting, which is clearly harder.

Domain Adaptation In machine learning, the joint distribution of data and labels $P(X, Y)$ is, of course, 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 in figure 2.1.
The source data is drawn i.i.d. from the source distribution, while the target

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

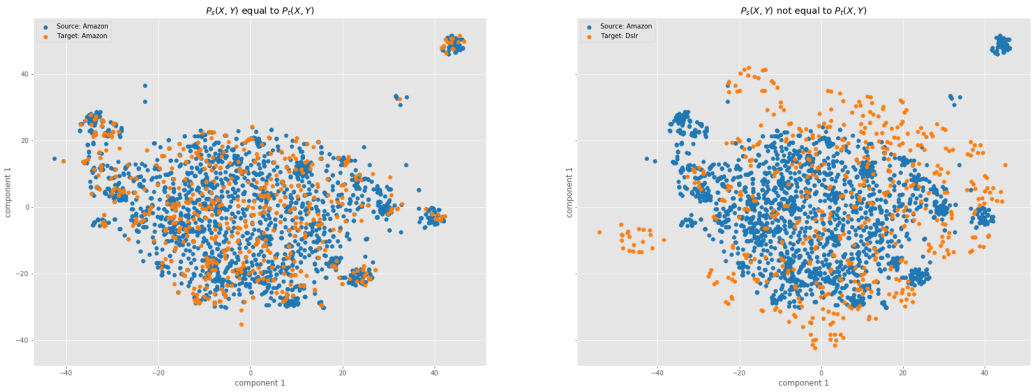

Figure 2.1: t-SNE Van Der Maaten L. 2008 visualization of the features extracted from the Office dataset using AlexNet. When the source and the target distribution are the same (left), a classifier trained on the source data generalizes very well to the target data. When the i.i.d. assumption does not hold (right), generalization performance will be poor.

data is drawn i.i.d. from the marginal over X of the target distribution:

$$S = (X_s, Y_s) \sim D_S = P_s(X, Y)$$
$$T = (X_t) \sim D_y = P_t(X)$$

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

$$\text{maximize} P(Y_t = \hat{Y}_t)$$

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 know.

2.2 Deep Learning

Introduction Artificial Intelligence (AI) technologies have to deal with the fact that current machines are very different from humans. In fact, it can be said that they are the opposite. Many tasks that require effort and reasoning for a human to accomplish, such as chess, are pretty easy for machines, because such problems can be encoded into a set of formal rules that machines are particularly suited to deal with. Conversely, other kinds of tasks that a human doesn't even need to think about are tremendously difficult for machines. These are the tasks we find so easy to perform that we don't even manage to explain how we achieve them. Encorporating the

intuition and common sense that humans give for granted 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**. 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 $(X_i \in \mathbb{R}^{1\times n}, Y_i \in \{1, 2, \ldots, C\})$, where X_i is a row vector of numerical features, 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.[2]. 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 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) Cybenko 1989, 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:

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

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

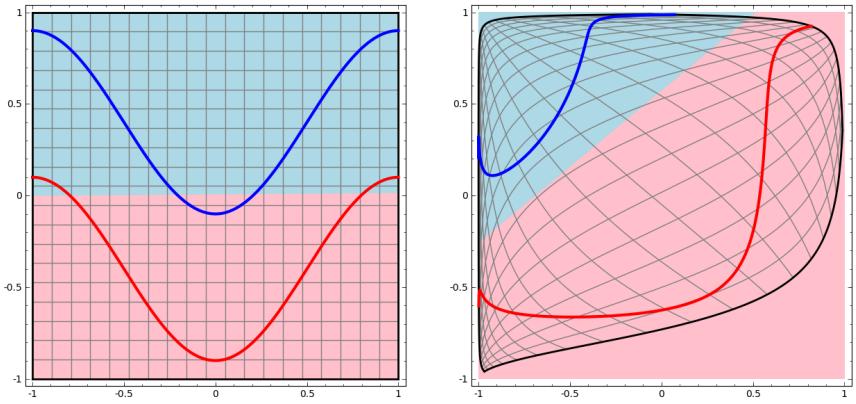

Figure 2.2: The original data (left) is not linearly separable, that is, there does not exists a hyperplane capable of separating the red curve from the blue curve. The transformation $h = \sigma(Wx + b)$ project the input x into a new space (right) where it's easy to find a separating hyperplane.[4].

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

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 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:

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

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

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

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.[5] 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:

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

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

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 version of gradient descent. Other techniques have been developed in recent years to overcome the major limitations of this scheme. The momentum Sutskever et al. 2013 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 Kingma and Ba 2014 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 David E. Rumelhart 1986 computes partial derivatives in a feed-forward neural network[6].
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.

[6]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

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 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 2.3:

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.

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

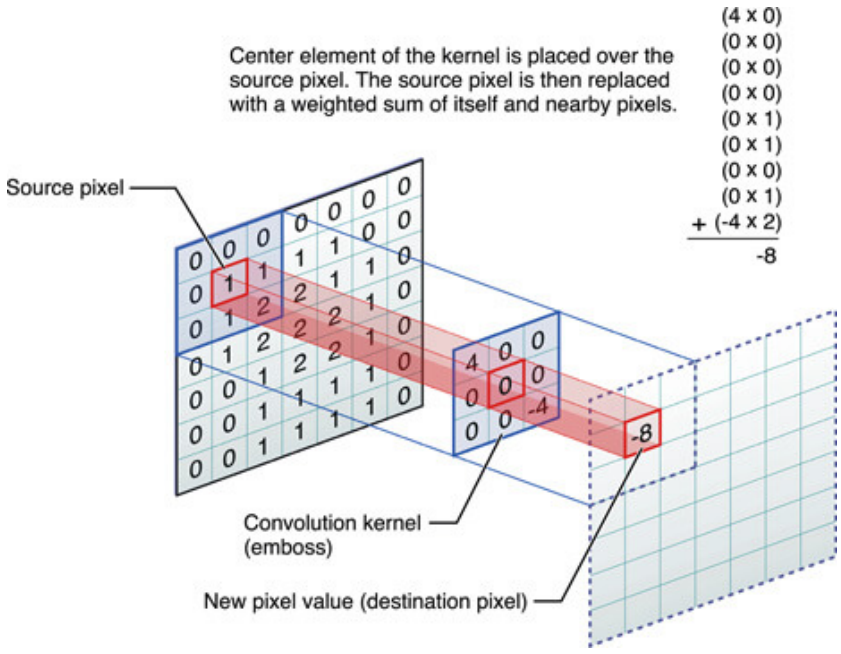

Figure 2.3: The Convolution operation[7].

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 hand-crafted, 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

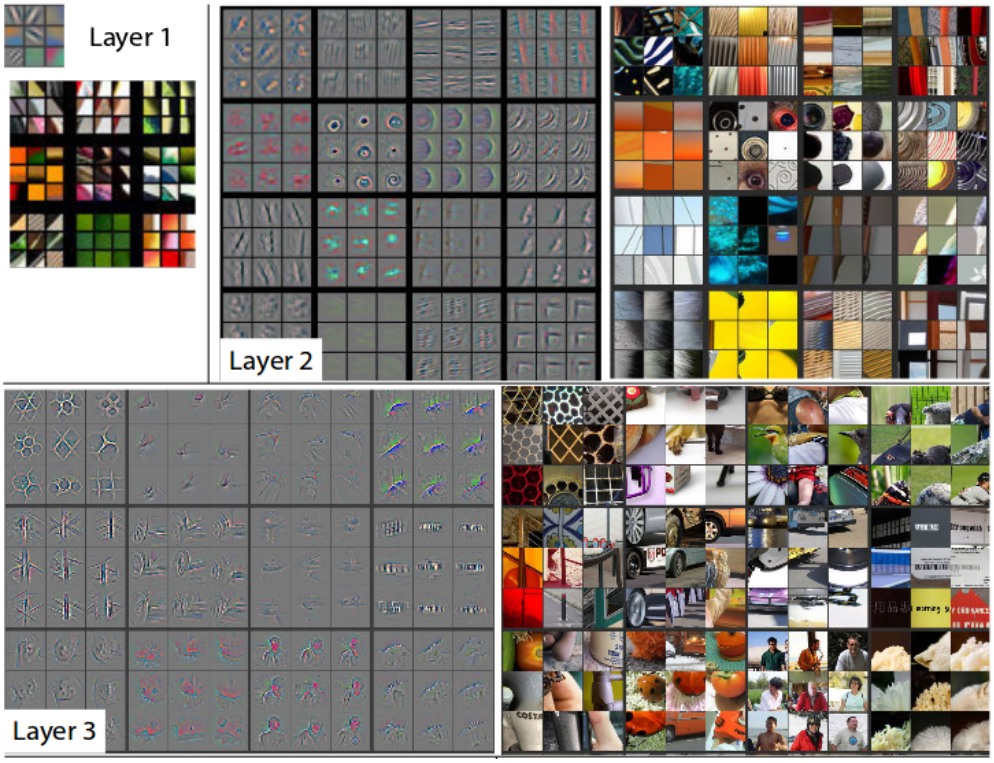

Figure 2.4: Visualizing Conv net filters, taken from Zeiler and Fergus 2013

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:

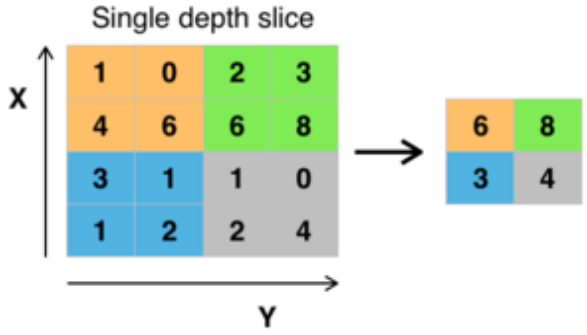

Figure 2.5: Max Pooling operation.

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 Csurka 2017 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 Shimodaira 2000. To solve the covariate shift problem, Shimodaira 2000 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) B. Fernando and Tuytelaars 2013, 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 T. Tommasi 2013. 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) Y. Ganin 2016, 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.

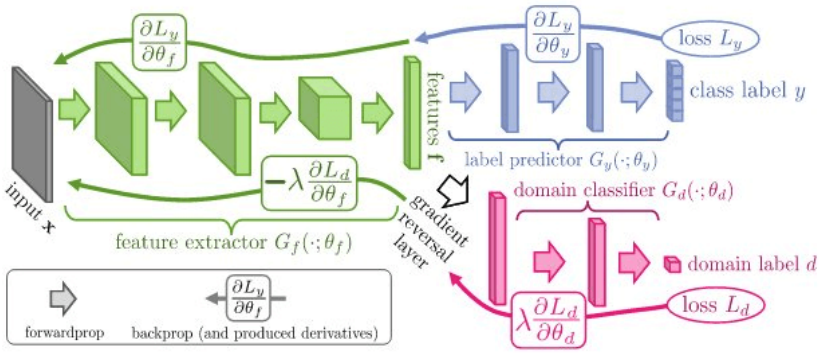

Figure 3.1: DANN Architecture. Image taken from Y. Ganin 2016.

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

4.1 Grad-CAM

Gradient-weighted Class Activation Mapping Selvaraju et al. 2016. Technique for producing visual explanations about why a Convolutional Neural Network opted for a certain class. 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 work, we'll use this technique to produce visuals for an object classification task. This technique 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. We extracted the Grad-CAM visualization from the last convolutional layer because, as also the authors or the technique note, the last convolutional layer 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 features one can expect to be extracted. The latter is a property of Convolutional Feature Maps, which are designed to retain as much spatial information as possible about the data. The high-level architecture of the method can be seen in the following figure.

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

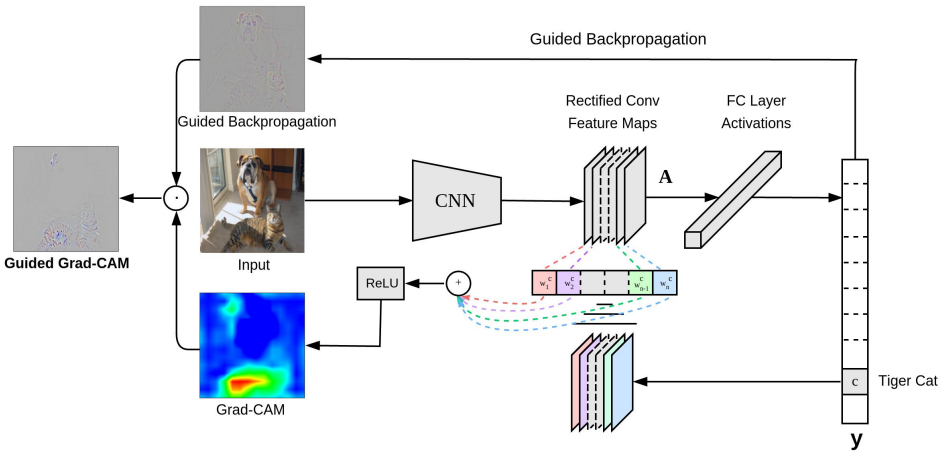

Figure 4.1: Grad-CAM Architecture, taken from Selvaraju et al. 2016

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 untill the last convolutional layer, where the Grad-CAM map is created. Assume 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. 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 generate, 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 Springenberg et al. 2014, to produce the final result Guided-GradCAM.

4.1.1 Grad-CAM for domain localization

In the previous section, we described the full Guided-GradCAM procedure, but in our approach we integrated only a subset of it. In particular, we used only Grad-CAM maps, and we retained the full *per feature-map activations*, instead of summing them. 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 the information into a single map by summing over depth, we retain the information about how each feature map contributes to the decision of the class. This is crucial, as it enables us to enhance and inhibit features with a greater precision. This procedure is applied to a convolutional network with a binary classifier on top of it, trained to discriminate source samples from target ones. Our rationale is that by applying Grad-CAM to this binary network we are able to determine regions in the image that are *domain-specific* or *domain-generic*: that is information that is contained only in the source (or target) domain and information that is shared between the two domains. Having this information, we should be able to make source and target more similar at the representational level. In order to do this, we will make use of another technique described in the next section.

4.2 Spatial Pyramid Pooling

Motivation Convolutional Networks can be thought of as composed by two parts, a feature extractor and a classifier. The former is a series of

convolutional layers, while the latter is simply a Multi-Layer Perceptron. Each convolutional layer has three sequential stages: the convolution, the non-linearity, and the 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. After the last pooling layer, the resulting 3D tensor of dimension $C \times H \times W$ is flattened into a 1D tensor with $C * H * W$, 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 He et al. 2014 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 4.2 is a depiction of how this layer works.

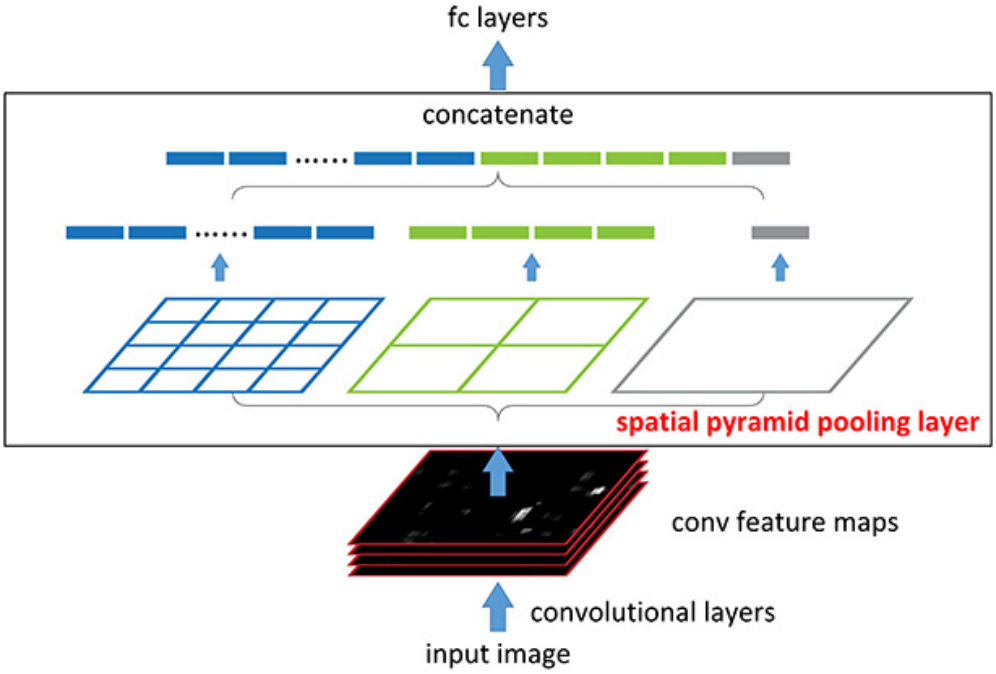

Figure 4.2: Spatial Pyramid Pooling layer. Image taken from: He et al. 2014

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 Simonyan and Zisserman 2014 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 4.2 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

[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.

sure that the dimension will be the same regardless of the input size.

4.3 Domain-Multiplicative Fusion

4.3.1 Motivation

The architecture of our network, which we called **Domain-Multiplicative Fusion (DMF)** is inspired from this work Park et al. 2016. In particular, the authors of Park et al. 2016 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 "Optical Flow Estimation" 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 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).

4.3.2 Architecture Design

Although we draw the idea of combining knowledge through multiplication from Park et al. 2016, 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

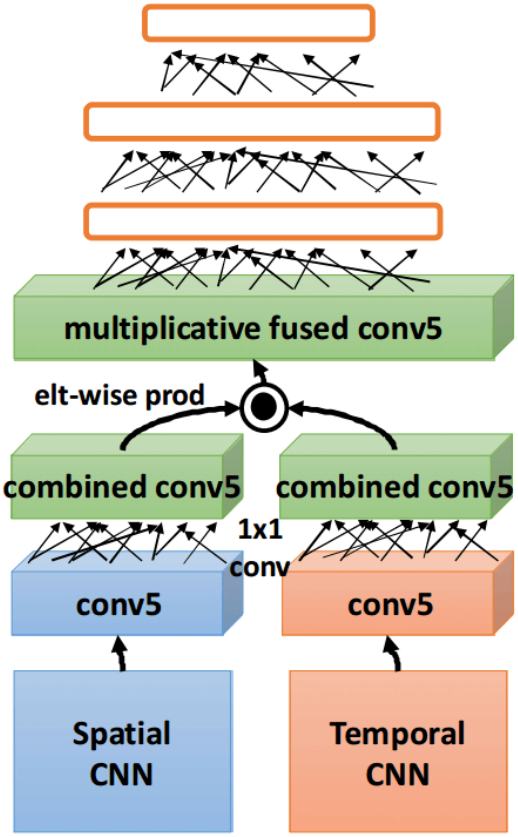

Figure 4.3: Multiplicative-Fusion architecture. Image taken from Park et al. 2016

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,

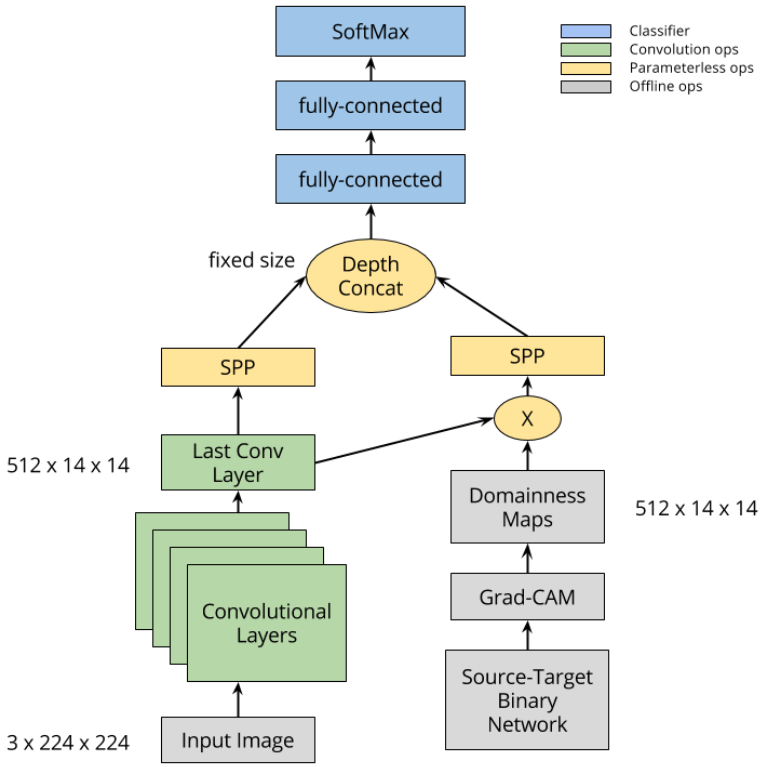

Figure 4.4: Domain-Multiplicative Fusion architecture.

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.

Chapter 5

Experiments

5.1 Experiment settings

Following recent developments in the field of *Transfer Learning* Lisa Torrey 2009, and due to the fact that the ICW dataset is pretty small, we didn't train a deep convolutional network from scratch. Instead, we started from a network fine-tuned on the popular dataset *ImageNet* *The ImageNet Dataset* 2017, which contains millions of images from 1000 object categories. A network trained on such a large dataset is capable of extracting meaningful features even from datasets it has never seen. The last layer of the classifier (the softmax layer) of course should be replaced with one which has the number of categories of the new dataset. Two popular network architectures were used in the experiments: AlexNet Alex Krizhevsky 2012, which represented a break point in the computer vision community by winning the ILRSVC 2012 image recognition challenge by a large margin, and the VGG16 Simonyan and Zisserman 2014 network, a very deep network with performance much higher than AlexNet. Each object classification network was trained using the following hyper-parameters: The first convolutional layers were freezed (layer 1 to f) Layers from f+1 to n-1 were soft freezed, that is they were let train, but with a rather small learning rate (1e-5) The last layer had a higher learning rate (8e-4) The optimizer used in the training procedure was Stochastic Gradient Descent with Momentum Sutskever et al. 2013. The momentum parameter was set to 0.9, with nesterov momentum Nesterov 1983 enabled. Batch sizes varied from network to network: we used a batch size of 256 for AlexNet and 16 for VGG16 (due to the high memory consumption of this network). Training was performed in parallel on multiple NVIDIA Titan X GPUs using the Torch Ronan Collobert 2011 deep learning library.

5.2 Office 31

5.2.1 Dataset Statistics

5.2.2 Training methodology

5.2.3 Testing methodology

5.2.4 Results

5.3 ICW

The iCub World dataset *The iCub World Dataset* 2013 contains objects from 15 categories, acquired through a robot camera. A human moves the object holding it in the hand and the robot tracks it by exploiting either motion cues or depth cues. Of the four different datasets, we will use the iCubWorld Tranformations dataset, in which 'each object is acquired while undergoing isolated visual transformations, in order to study invariance to real-world nuisances'. In particular, the visual transformations are the following:

- **2D Rotation**: The human rotated the object in front of the robot, parallel to the camera plane, keeping it at the same distance and position.
- **Scaling**: The human moved the hand holding the object back and forth, thus changing the object's scale with respect to the cameras.
- **Background Change**: The human moved in a semi-circle around the iCub, keeping approximately the same distance and pose of the object in the hand with respect to the cameras. The background changes dramatically, while the object appearance remains the same.

In figure 5.1, we can see a random sample of images from the dataset.

Dataset Statistics For our domain adaptation purposes, we divided the original dataset into two sub-datasets, that we called *Left 1* and *Left 2*. These names stem from how we divided the original dataset. In the background change transformation, the human moved in a semi-circle around the robot. We put frames at the beginning of the semi-circle in *Left 1*, and frames at the end of it in *Left 2*, while we have discarded frames at the center of the circle. Thus we have two DA settings: one in which the source is Left 1 and the target is Left 2, and vice-versa. Left 1 has 4500 images, while Left 2 has

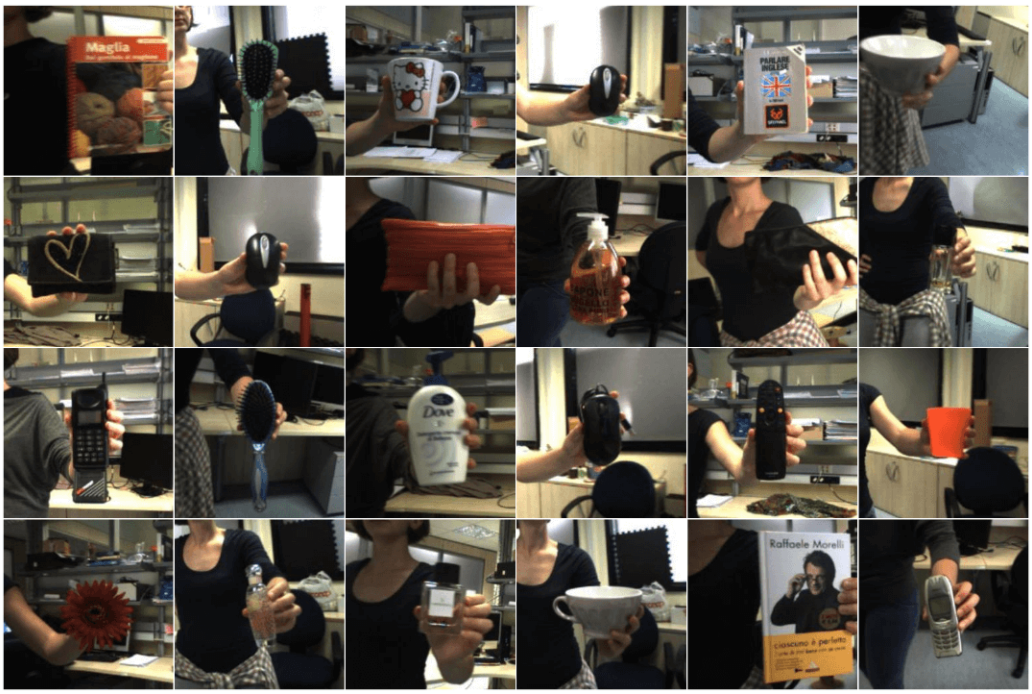

Figure 5.1: Random sample from the iCubWorld dataset.

3000 images. The classes are perfectly balanced, in that each one has the same number of samples.

Training methodology

Testing methodology

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

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

B. Fernando A. Habrard, M. Sebban and T. Tuytelaars (2013). "Unsupervised visual domain adaptation using subspace alignment". In: *IEEE International Conference on Computer Vision (ICCV)*.

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

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

David E. Rumelhart Geoffrey E. Hinton, Ronald J. Williams (1986). "Learning representations by back-propagating errors". In: *Nature* 323.

David J. Fleet, Yair Weiss. "Optical Flow Estimation". In:

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

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

Jiang, Jing (2008). "A Literature Survey on Domain Adaptation of Statistical Classifiers". In:

Kingma, Diederik P. and Jimmy Ba (2014). "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980. URL: http://arxiv.org/abs/1412.6980.

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

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

Park, Eunbyung et al. (2016). "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. DOI: 10.1109/WACV.2016.7477589.