Transferability of Features from Multiple Depths of a Deep Convolutional Neural Network

Jesper Edström, Emil Widell

Master's thesis
2018:E24

LUND UNIVERSITY

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

Acknowledgements

We would like to thank our supervisors Prof. Karl Åström and Dr. José Díaz López for their unwavering enthusiasm and continuous support throughout our thesis. We would also like to thank Qlik for lending us the resources to train the models that are essential for this research.

Abstract

Deep convolutional neural networks are great at learning structures in signals and sequential data. Their performance have surpassed most non-convolutional algorithms on classical problems within the field of image analysis. A reason behind their success is that even though these networks generally need a great amount of examples to learn from, they can be used to learn smaller tasks through different types of transfer learning techniques. When having small amounts of data, a common approach is to remove the output layer and use the remaining network as a feature extractor. In this work we attempt to quantify how network layers go from general to specific through extracting features from multiple depths. The transition was measured on some different object classification problems by training classifiers both directly on the feature vectors and on combinations of the vectors.

The reached conclusion was that the feature from the very last layer of a deep convolutional network are very specific to the source task and using it to learn other classification problems is often sub-optimal. The best depth to extract features from depends on how similar the problem you want to learn is to the source task.

Table of contents

1 Introduction **1**

1.1 Related Work . . . 2

1.2 Approach . . . 3

1.3 Domains . . . 3

1.3.1 Source Domain . . . 3

1.3.2 Target Domains . . . 4

2 Scientific Background **7**

2.1 Machine Learning . . . 7

2.1.1 Overfitting . . . 8

2.1.2 Supervised Learning . . . 8

2.1.3 Unsupervised Learning . . . 9

2.1.4 Dataset Partitioning . . . 9

2.1.5 Support Vector Machines . . . 10

2.2 Artificial Neural Networks . . . 13

2.2.1 Backpropagation . . . 15

2.2.2 Convolutional Layers . . . 16

2.2.3 Pooling Layers . . . 18

2.3 Inception Module . . . 18

2.4 Deep Learning . . . 19

2.5 Transfer Learning . . . 21

2.5.1 Source Domain Considerations . . . 22

2.5.2 Fine-tuning . . . 22

2.5.3 Feature Extraction . . . 23

2.6 Visual Analytics . . . 24

2.6.1 Linear Projections . . . 25

2.6.2 Linear Decomposition . . . 25

2.6.3 Non-linear Embeddings . . . 25

3 Experiments **29**
3.1 Choosing a Classifier . . . 29
3.2 Average vs. Max Pooling . . . 29
3.3 Preprocessing . . . 30

4 Method **33**
4.1 Dataset Manipulation . . . 33
4.2 Feature Extraction . . . 33
4.2.1 Average Pooling . . . 34
4.2.2 Combining Features . . . 34
4.3 Classification . . . 35
4.4 Evaluation . . . 35

5 Results **37**
5.1 Accuracy Tables . . . 38
5.2 Single Features . . . 41
5.3 Grouped Features . . . 42

6 Discussion **45**
6.1 Conclusion . . . 47
6.2 Future Work . . . 47

References **49**

Chapter 1

Introduction

Many of the recent achievements in the area of image analysis and computer vision can be attributed to the creation and continuous development of convolutional neural network (CNN) architectures. Features and filters that were previously created by experts can now be learned directly from images, shifting the focus from feature engineering to architectural model design. Experimental findings has shown time and time again [1, 13, 35] that the representational power of learned features extracted from deep CNNs are often general enough to be used on a wide variety of visual recognition tasks. Utilizing data from one domain to improve the results on another is called transfer learning and it is often the first alternative when trying to learn a new visual recognition tasks. The advantages of this kind of approach is that it enables analysis of a target task without having nearly as much data as on the source task. It is also faster and can achieve better accuracy than training a network from scratch, see [53].

It has been observed that deep convolutional networks trained on different large image datasets often learn similar features early in the network, [54]. The features from the bottom layers are more general and often resemble lines, color blobs and Gabor filters. Features from the later layers are more specific to the source domain and have learned more advanced representations, like textures and contours. We want to capture this transition from general to specific and investigate how these features are best used to solve problems different to that of the source task. When using feature extraction on deep neural networks, a common method has become to remove the output layer of the network and use the features from the second-to-last layer. While there are many examples where this method has proven useful, other ways of extracting features has not been explored as much and could potentially yield better results.

1.1 Related Work

In [53], the authors want to quantify the degree to which a particular layer is general or specific. They did so by using a transfer learning technique called fine-tuning to retrain a deep network towards a new target domain. In their approach, they used very large datasets to enable fine-tuning of their deep source network.

Our approach is different in that we are interested in exploring contexts where there are limited amounts of training data. Therefore we did not want to use fine-tuning of networks. Also, instead of transferring to sub-domains of the source domain, we wanted to transfer to different domains and see how the transferability varies between them.

In [13], the authors outperform previous state-of-the-art algorithms on some smaller classification tasks by training a new classifier on features extracted from a deep source network. We propose a similar method to theirs but instead of only examining the performance of the last layers, we will extract features from all of the layers in order to examine the transition.

The authors argue that features extracted from the convolutional layers are unlikely to contain a richer semantic representation than features extracted from the fully connected layers. While true in the general case, it does not necessarily mean that they are always more suitable for transfer. Especially if the new task is very different from the original one. Later layers could contain a lot of semantic representations that are too specific to be useful on the new task.

In [14], the authors used transfer learning on a deep CNN in order to solve a multi-instance multi-label (MIMS) problem. Features where extracted from all of the convolutional layers, concatenated and then fed into a shallow classifier. While their results on the datasets were impressive, they only trained models using all of the layers. This project aims to compare how different subsets of concatenated features would impact classification performance.

From these papers it can be noted that features early or late in the network are more or less desirable depending on the type of transfer learning they are used for. In the fine-tuning case, where the authors re-trained deep networks, it was favorable to only keep the first few layers. While in the feature extraction case, where the authors trained a shallow classifier, it was better to use features from later in the network to keep more of the semantic representations.

In our work we want to compare features extracted from different depths in order to verify the claim in [13], that late layers are more useful when training a shallow classifier. While deep layers should carry a more advanced semantic representation, we show that some features from late in the network can be too specific to the source task. In that case more general features extracted earlier in the network prove more useful, especially when the target domain is further from the source domain.

1.2 Approach

We investigate the performance of features extracted from the layers of a deep network and evaluate them on different image classification tasks. Rather than trying to maximize the classification performance, we want to compare the usefulness of features from different layers and see how well they generalize to other problems. We have two questions that we will try to answer:

1. How useful are features extracted from different layers of a source network when trying to learn a new target task?
2. Can features from different layers of the source network be combined to increase classification performance?

In our attempt to answer these questions we make the following contributions:

1. We quantify the classification performance of features from all depths of a deep source network across multiple target tasks.
2. We quantify how combining features extracted from adjacent layers impact classification performance.

We conclude that the common method of extracting features from the very last layer of deep networks is often sub-optimal and suggests some alternative approaches. We also show that combining some features from adjacent layers can boost classification performance but that concatenating all of the layers are generally worse. We hope to give the reader a better understanding of how deep network architectures learn, some better intuitions about the role of different network layers and how to best make use of them when learning a new task.

1.3 Domains

When using transfer learning on a deep CNN the goal is to make use of the semantic knowledge that the algorithm has learned on the original (source) domain, when learning a new (target) domain. We only used well known and researched domains in order to assure relevance and reproducibility of our results.

1.3.1 Source Domain

As the source task to transfer from we used the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [11]. It is constructed using the ImageNet dataset. ILSVRC is a

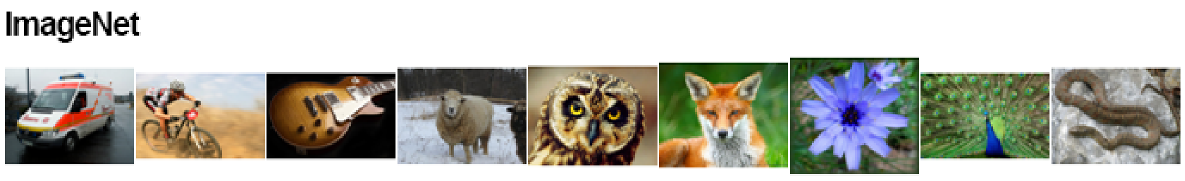

Fig. 1.1 Random images from the ImageNet dataset. Note that the images typically only contain one object each, are of different sizes and that there is great diversity between classes.

1000-way object classification task with 1.2 million natural photographs in its training set and 150,000 in the test set. The images are of varying sizes and aspect ratios but most images are bigger than 256x256 pixels. A random selection of images from the dataset can be viewed in figure 1.1.

ImageNet is often used when training new deep networks because it has been shown experimentally to produce features that are useful on many other problems within image analysis. As seen in [22], using features from networks that are pre-trained on ImageNet has produced a lot of impressive results on many types of problems. Using ImageNet in the pre-training step of deep CNNs has not only been a successful method for solving other image classification problems [13], but also to solve problems like action recognition in videos [45] and human pose estimation [8].

1.3.2 Target Domains

The selection of target domains were made trying to capture a large variety of object classification problems with different distances to the ImageNet source domain. How similar a problem - described by an image dataset - is to another is hard to quantify, but we try to explain our reasoning behind our choices. Images from the target domains can be viewed in figure 1.2.

MNIST

MNIST [29] is an image dataset of 70,000 handwritten digits. The images are in grayscale and of size 28x28. It is one of the classical datasets for algorithm testing within machine learning. Deep CNNs have already achieved human-level performance on MNIST [10] but it still makes sense to use it in this type of analysis since we are comparing differences in performance instead of trying to maximize performance.

CIFAR-10

As a second target domain we used the CIFAR-10 [26] dataset which contain 60,000 images of 10 different objects. It has also been one of the standard datasets within machine learning research. It was constructed by paying students to label images from the TinyImages dataset [49], which contains over 80 million images. The dataset is divided into four classes of man-made objects and six classes of animals. The classes are similar to the ones contained in the ILSVRC source task but the images are smaller, only 32x32 pixels.

CelebA

CelebA [31] is a multi-labeled face dataset where each image is labeled with 40 facial attributes. The faces are centered but are taken from a multitude of different angles. We used this dataset to construct two different binary classification problems with 60,000 images in each. The first problem is to classify whether or not a person is young and the second one to classify whether or not a person is smiling. We chose these problems because when examining the dataset we thought these attributes to be the most objective. Other attributes like *beautiful* or *pointy nose* seemed subjective and also inconsistently labeled within the dataset. We also reasoned that to classify whether or not a person is smiling the network need to consider a much smaller region of the image than for determining if a person is young.

From these two problems we can compare and reason about how well the locality is captured in the features. Trying to classify facial attributes is a problem that is distant from the ILSVRC source domain [1].

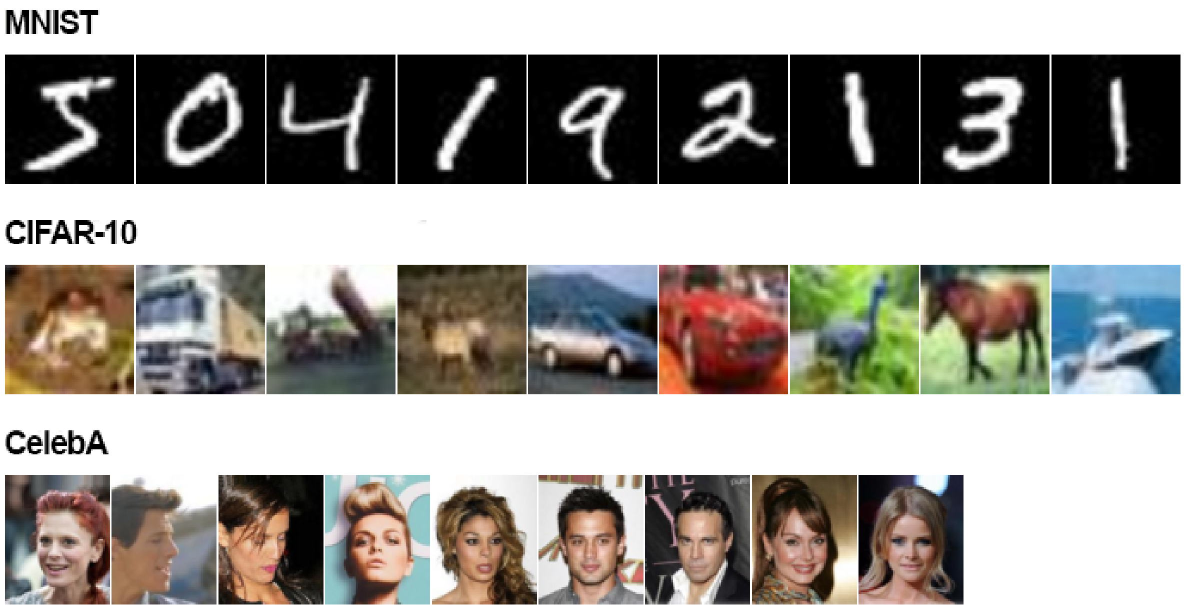

Fig. 1.2 Random images from each of the three different target datasets we used to compare the transferability of features. Each image contains one object that is more or less centered in the image. The problems described by the datasets are vastly different. In MNIST (*top*) the problem is to distinguish handwritten numbers from each other where each pixel is either black or white. In CIFAR-10 (*middle*) the problem is to distinguish between natural objects of 10 different classes that are mutually exclusive. In CelebA (*bottom*) each image is described by 40 different attributes. We chose two binary problems from these attributes. To distinguish whether or not a person is young and whether or not a person is smiling. The datasets are of different sizes and aspect ratios which is something we address in our method.

Chapter 2

Scientific Background

In this chapter we give a brief scientific overview of the different areas that we used in this work as well as other techniques that are useful for evaluating our method and results.

2.1 Machine Learning

Machine Learning (ML) is one of many fields within the broader subject of artificial intelligence (AI). A common misconception about AI research is that it is only carried out with the goal of creating sentient beings with a general intelligence, much like ourselves. While this is true for some areas within AI, it is not true for a majority of the field. Instead the focus is much the same as it has always been within mathematics, to solve different specific and technical problems. While the definition of AI is still argued about, AI algorithms are often characterized by their ability to solve problems by perceiving their environments and act to reach a goal. ML algorithms do this by learning from experience.

What differs ML from other fields within AI is that while other AI techniques often use carefully predefined search algorithms, heuristics and logical expressions, ML use a more black box strategy and focuses on training an agent to autonomously learn how to solve a problem. In practice the distinction is not always clear. Generally, ML algorithms have a defined approach but with fewer pre-defined parameters. Instead, they use optimization techniques during training to find values to their parameters. Exactly what an ML model can learn varies greatly between algorithms and architectures. A simple one layer neural network can model quite simple mathematical formulas, e.g a straight line. the slope and intercept of the line can be found through optimization algorithms like *gradient decent* [25] or *conjugate gradient* [42]. Other ML-algorithms like the Neural Architecture Search (NASNet) [55] learns not only the mathematical models, but also the actual architecture of the model.

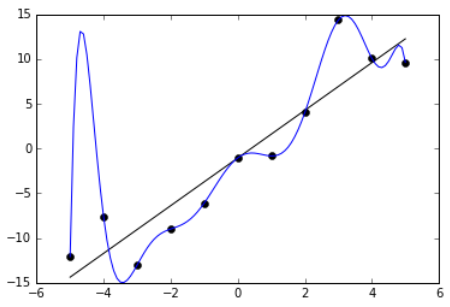

Fig. 2.1 The straight line models the true domain and the data points are sampled with noise. The blue line has overfitted the data. Image is taken from [15].

2.1.1 Overfitting

A problem with ML is that it is hard to train a model that solves the actual problem instead of the problem represented by the data. Overfitting is when *higher accuracy* is gained on the sampled data than on the actual domain. The collected data is almost always a small sample from the real domain, so if a trained model is to be used in the real domain, three things becomes important. One is that the dataset needs to represent the domain well. E.g. if the task is to detect any animal in an image, a dataset of images only on horses and dogs would not be descriptive enough. Second, sampled data can seldom be trusted to model the problem perfectly and often contains noise and bad samples. Examples of this in image analysis could be wrongly labeled images or color distortion. If one puts too much trust to the dataset, the model will be prone to overfitting, Figure 2.1 describes such a scenario. The third important thing is that a model should not be unnecessarily complex, since complexity does not generalize well. Figure 2.2 shows an example of this. Too complex models usually learns false patterns within the data which does not hold in the real domain. On the other hand, if too much effort is made to generalize, the model might instead end up learning nothing at all. That is called *underfitting* and there will always be a trade-off between the two. There are different techniques designed to mitigate overfitting, many of which are algorithm-specific, but one common approach is to partition the dataset. This technique will be further described in section 2.1.4.

2.1.2 Supervised Learning

Supervised learning is the process of training a ML model using labeled data. This is done by using the data as input-output pairs and train a model to map an input to its corresponding output. Classification and regression tasks are two examples of problems that can be solved using supervised learning. Supervised learning algorithms uses a loss function which can

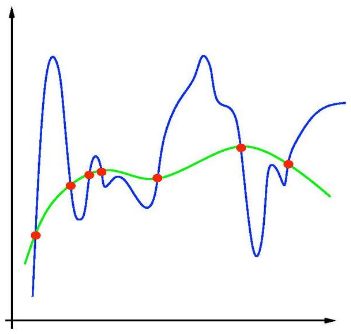

Fig. 2.2 A case where the data is sampled without noise. The blue line seems to be unnecessarily complex and a newly sampled data point will probably not be interpolated by the blue line. The green line is less complex and seems to model the data in a more reasonable way. There is a high risk that the blue line has overfitted the data. Image is taken from [37].

quantify the error using the ground truth data, this loss function is minimized during training time by iterative optimization functions.

One disadvantage of supervised ML models are that labeled data is necessary. It is important that the dataset acquired are qualitative and at the same time large enough. To uphold quality, the dataset must often be labeled manually and therefor the creation of labeled datasets are an extremely expensive process.

2.1.3 Unsupervised Learning

As opposed to supervised learning, unsupervised learning is the approach to use when the data acquired does not have any labels. Instead of doing regression or classification on such data it is possible to perform clustering techniques, noise reduction or anomaly detection. Unsupervised learning algorithms uses loss functions that - unlike supervised methods - are not dependent on any ground truth. Instead their loss functions tries to quantify some measure as *"how un-clustered is the data"*. This function is then iteratively minimized. K-means [6] and Autoencoders [3] are examples on techniques that can be used in a unsupervised setting.

2.1.4 Dataset Partitioning

To be able to train a model that captures the domain and adapts well to new data it is crucial to have a dataset that represents the domain well [7]. It is common practice to divide a dataset into two or three parts. A training set, a test set, and sometimes a validation set. The training set is used to train a model which is then tested on the test set. A common ratio is to divide the dataset into circa 80% training data and 20% test data. The more test data, the more certain one can be of the measured result, but with more test data comes less training data, in which case the model can be expected to perform worse. It is good practice to partition the dataset such that all parts have the same ratio of classes.

Many ML algorithms has two types of parameters: parameters which are learned, and hyper-parameters which are manually set before training. Experimenting with hyper-parameters is often necessary and with enough iterations the model might start to overfit to the test data. To avoid this, the test set should be used as few times as possible. With every iteration using the test set, the risk of overfitting to it is increased. This problem can be mitigated by dividing the training set further into a train and validation set. Hyper-parameters are iteratively tested on the validation set, and the final model are tested on the test set.

2.1.5 Support Vector Machines

A Support Vector Machine [46] (SVM) is a type of ML model. They are trained in a supervised fashion and are mostly used for binary classification. The details of SVMs can be varied widely and to the extent that the models can even be used to solve regression problems. One often distinguish between these subdivisions by referring to them as Support Vector Classification (SVC) and Support vector Regression (SVR) algorithms. Moreover, names as Linear SVMs, L1- or L2-SVMs are sometimes used to further describe the implementations. Regardless of the implementation, the idea behind an SVM is to find the optimal hyperplane to use as a decision boundary. The basic idea behind SVC and some of its possible variations are described below.

Mathematical Background

Given that we have a training set $(\mathbf{x}_i, y_i)$ for $i = 1...n$, where $\mathbf{x}_i \in \mathbb{R}^d$ are a d-dimensional data point and $y_i = -1$ or 1 are the labels. We formulate a hyperplane as

$$\mathbf{w} \cdot \mathbf{x} - b = 0,$$

where $\mathbf{w}$ is an orthogonal vector to the hyperplane. There are two slightly different approaches called *hard-margin* and *soft-margin* SVM. The former finds an exact solution and is used if the data is linearly separable, i.e, a straight line exists such that it can discriminate between the classes. The latter approach is used if the data is not linearly separable. First, the *hard-margin* approach is demonstrated. We want to find two parallel hyperplanes that separates the data. The two hyperplanes are

$$\begin{aligned} h_1 &= \mathbf{w} \cdot \mathbf{x}_a - b = 1, \\ h_2 &= \mathbf{w} \cdot \mathbf{x}_b - b = -1, \end{aligned}$$

where all data points above h_1 belong to class 1 and all data points below h_2 belongs to class -1. Instead of formulating the hyperplanes we could write this as the constraints

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x}_i - b &\geq 1, \text{ if } y_i = 1, \\ \mathbf{w} \cdot \mathbf{x}_i - b &\leq -1 \text{ , if } y_i, = -1, \end{aligned}$$

or even more efficiently,

$$y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1. \tag{2.1}$$

Since the data is linearly separable, $h_1$and h_2 exists such that there are no data points between them. This space $\triangle$ between the hyperplanes are called the margin, and we want to maximize this margin. We can calculate the margin geometrically according to

$$h_2 - h_1 = (\mathbf{w} \cdot \mathbf{x}_b - b) - (\mathbf{w} \cdot \mathbf{x}_a - b) = (1) - (-1) \Rightarrow \tag{2.2}$$
$$\triangle = \frac{\mathbf{w}}{||\mathbf{w}||}(\mathbf{x}_b - \mathbf{x}_a) = \frac{2}{||\mathbf{w}||}. \tag{2.3}$$

Figure 2.3 gives more intuition about what this calculation means. We can see that to maximize the margin 2.3, we need to minimize $||\mathbf{w}||$ under the given constraint 2.1. However there are cases when the data is not linearly separable, in which case we have to solve the *soft-margin* problem instead. In this case we want to minimize

$$\left[C \sum_i L(\mathbf{x}_i, y_i) \right] + \frac{1}{2} ||\mathbf{w}||^2, \tag{2.4}$$

where $L(\mathbf{x}_i, y_i)$ is some loss function. C determines the trade off between classifying all $\mathbf{x}_i$ correctly and increasing the margin size. As $C \to \infty$, we get increasingly close to solving the *hard margin* problem.

L1- and L2-SVMs are often talked about and the difference between them is how to treat the loss function. For L1-SVM we want to minimize the function 2.4 while in the case of

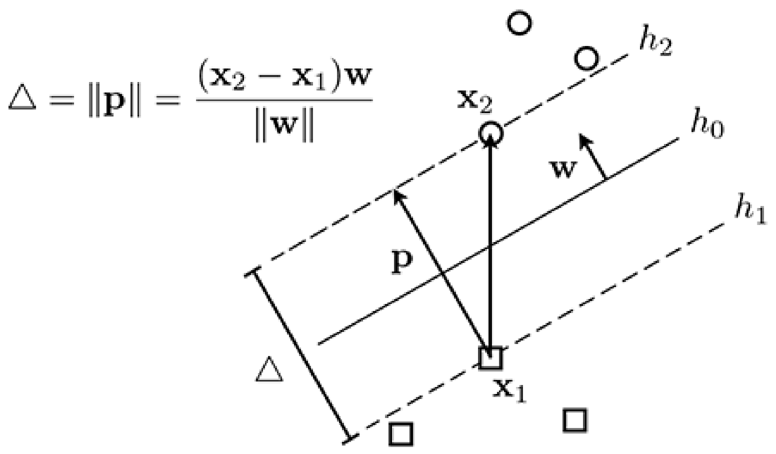
$\triangle = \|\mathbf{p}\| = \frac{(\mathbf{x}_2 - \mathbf{x}_1)\mathbf{w}}{\|\mathbf{w}\|}$
h_2
$\mathbf{x}_2$
h_0
$\mathbf{w}$
h_1
$\mathbf{p}$
$\triangle$
$\mathbf{x}_1$

Fig. 2.3 Image to give some intuition about the calculation of the distance between the hyperplanes h_1 and h_2. $\mathbf{x}_1$ and $\mathbf{x}_2$ are points on the respective hyperplane and $\mathbf{w}$ are a normal vector to h_0. The hyperplanes are parallel to each other. The data points laying on the hyperplanes h_1 and h_2 are called support vectors.

L2-SVM we want to minimize

$$\left[\frac{C}{2}\sum_i L(\mathbf{x}_i, y_i)^2\right] + \frac{1}{2}\|\mathbf{w}\|^2,$$

where the first term is differentiable and gives higher penalty to misclassified points compared to function 2.4. A common loss function for SVMs is the hinge loss function

$$\max(0, 1 - t \cdot e),$$

where $t = \{-1, 1\}$ is the output and e is the input. Or in our case,

$$\max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i - b)).$$

Summing this loss function over i quantifies the distance to the margin for all the misclassified points, notice also that only the misclassified points contributes to the expression we want to minimize. If we use the L1-SVM, for example, we want to minimize

$$\left[C\sum_i \max\left(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i - b)\right)\right] + \frac{1}{2}\|\mathbf{w}\|^2.$$

After calculating the $\mathbf{w}$ and b that solves this problem, our SVM model can classify a data point $\mathbf{x}_c$ according to the formula

$$y = sgn(\mathbf{w} \cdot \mathbf{x}_c - b).$$

Kernel Trick

When the data is not linearly separable, it is possible to overcome this by using a kernel trick. When using a kernel trick the SVM tries to map the data onto a space of higher dimensionality, where the data is linearly separable. The idea is to use a kernel function that satisfies

$$k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$$

to transform the data points. Polynomial, Sigmoid and Gaussian kernels are common kernels to use. For details, see [20].

SVM for multiclass problems

Though SVMs are trained to solve binary class problems, they can be used in a *one-vs-all* or *one-vs-one* scheme to solve multi-class problems.

In the case of one-vs-all, one SVM are trained for each class. Each models job is then to tell if the input corresponds to the class its trained on, or not. Then all models are used in a winner-takes-it-all strategy where the model with the highest output score decides the outcome.

The one-vs-one case is a little more complicated. In this case an SVM is trained for each pair of classes. Each models job is then to distinguish between only their pair of classes. There might be many models that is completely irrelevant to certain inputs. This problem is overcome by using the max-wins voting strategy where, given an input, each model votes on one of its two assigned classes, and the class with the most votes wins.

2.2 Artificial Neural Networks

In 1943, Warren S. McCulloch and Walter Pitts published an article [32] where they created a mathematical model with inspiration from their understanding of the biological neural network. This laid the foundation for what would become the field of Artificial Neural Networks (ANNs) and spurred researchers and philosophers to discuss whether intelligent machines could be within reach. In 1969, M. Marvin and S. Papert published a book [33]

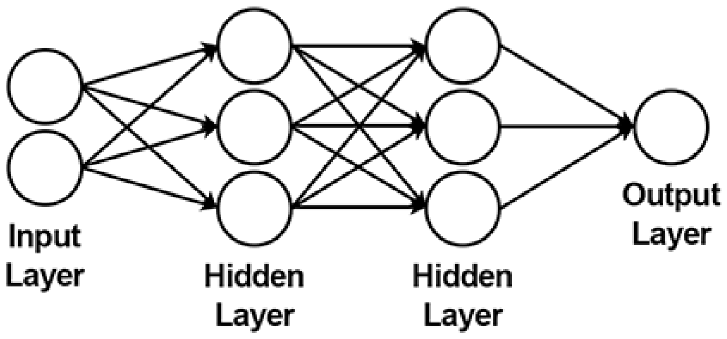

Fig. 2.4 Describes an example of a neural network of fully connected layers, in abstract terms. There are four layers of nodes. Each node are connected to each other node in the neighboring layers.

where they argued that ANNs could be a dead end considering perceptrons - a node within a network - inability to solve nonlinear problems. They showed that an ANN had to be multi-layered to solve a nonlinear task. All previous attempts to implement a multi-layer network had failed, and so the hype somewhat died out for many years. This problem was later solved with the discovery of *backpropagation*. Since then the field of ANNs have died out and been rediscovered many times through the years and have now grown to be a broad research field targeting countless problems.

Mathematically, an ANN is a network of nodes that can model a variety of functions [21]. A node has a number of inputs, and one output. All inputs are independently multiplied with weights, then summed together with a bias node and passed through an activation function. The output from the activation function is the output of the node. See Figure 2.5. A basic neural network consists of one input layer, one output layer and often one or more hidden layers in-between. Commonly every node is connected to every other node in-between the layers. For a visual description of a simple network, see figure 2.4. When all nodes between neighboring layers are connected, it is called a fully connected layer. There are many types of neural networks. When all data is flowing in one direction, i.e from input to output, it is called a feed forward network. Alternatives exists, where certain architectures allow information to loop into previous layers. Such networks are called Recurrent neural networks (RNNs).

There is not many objective right and wrongs when it comes to network design, the architecture of the network depends a lot on the type of problem it should solve and new ingenious architectures are continuously being invented.

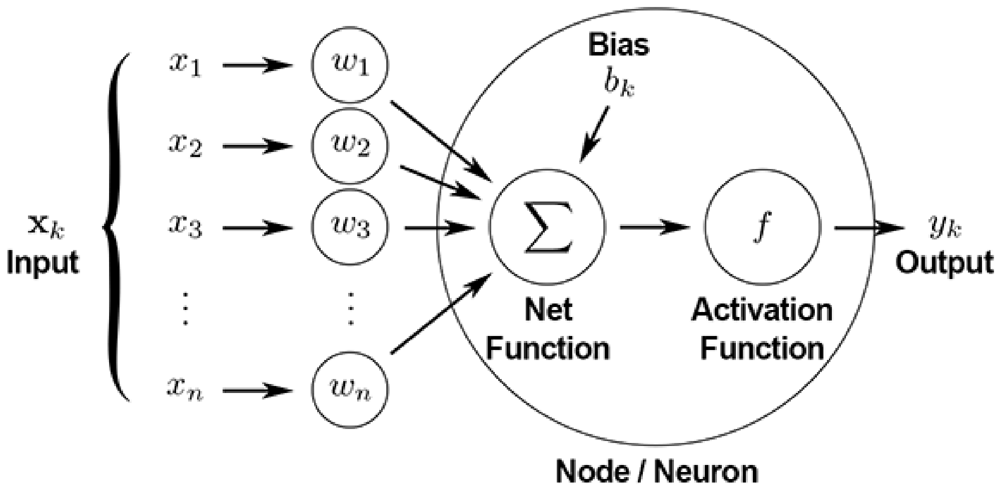

Fig. 2.5 Describes a node k within a neural network. The inputs $x_1 ... x_n$ are multiplied with the weights $w_1 ... w_n$, then summed together with a bias input. The sum is the input to an activation function that produces the output from the node. This can be expressed as $y_k = f(\mathbf{x}_k \cdot \mathbf{w}_k + b_k)$.

2.2.1 Backpropagation

An ANN has a defined loss function which it wants to minimize. The loss - or error - is computed as some quantization of the difference between the networks input and the desired output. Within the network, it is the weights that are subject to change during training. These weights are altered through the process of backpropagation [18], which propagates the error back through the network.

When tuning the weights, one needs to know how each individual weight affects the output error, and this is exactly what backpropagation computes. If we define a loss function E which we want to minimize. Then for each weight w in our network, we want to calculate $\frac{\delta E}{\delta w}$. When we know how each weight affects the error, we can act upon that information. One way to update w is to use the update rule

$$w := w - \alpha \frac{\delta E}{\delta w} \tag{2.5}$$

where α is the learning rate which decides how fast the network should learn.

To understand this process further, we need to think about what a node is, and what it means for the differentiation ($\frac{\delta E}{\delta w}$) that nodes are connected. The output of a node i is a function of its inputs $y_i = f_i(\mathbf{w}_i \cdot \mathbf{x}_i)$, where $\mathbf{w}_i$ is a vector of weights in node i and $\mathbf{x}_i$ is a vector of inputs to node i. Note that we for simplicity use a somewhat sloppy notation and disregard the bias node. Consider a node j in the same way $y_j = f_j(\mathbf{w}_j \cdot \mathbf{x}_j)$, and where the output y_j is one of the values in $\mathbf{x}_i$. Meaning, the output of j is one of the inputs to i. Further, lets say that the weight w_{ja} are a single weight among all weights in $\mathbf{w}_j$. Since these nodes are connected, y_i is in part a function of y_j, and thereto in part a function of $\mathbf{w}_j$. This is important because if we want to calculate, say $\frac{\delta E}{\delta w_{ja}}$ the chain rule comes into play,

$$\frac{\delta E}{\delta w_{ja}} = \frac{\delta E}{\delta y_j}\frac{\delta y_j}{\delta w_{ja}} = \frac{\delta E}{\delta y_i}\frac{\delta y_i}{\delta y_j}\frac{\delta y_j}{\delta w_{ja}}.$$

Thus, large networks with hundreds or thousands of weighs generates long expressions that quickly gets computationally heavy. It follows that if too many of the terms in a large expression are too small, the product will rapidly go to zero. If we use the update rule in 2.5 our network will practically stop learning. This is the problem of *vanishing gradients* [36]. On the other hand, if too many terms are too large, we get the *exploding gradients* problem [36], which will make the network unstable and might output values as *NaN*. Also, before training a network, we need to initialize the weights with backpropagation in mind. In small networks, we can afford to write out these differential equations by hand to verify that it will compute. In large networks this gets impractical, but this knowledge can still be used to define activation functions, loss functions or even whole architectures in a way that is easy to work with in a differential setting.

2.2.2 Convolutional Layers

Traditional ANNs built on fully connected layers are ineffective when dealing with images as inputs. The reason for this is the *curse of dimensionality*. If an image is used as an input to a network, there are no other information to use but the intensities on the pixels. A small RGB image might have $32 \times 32 \times 3 = 3072$ pixels. A fully connected layer would connect each pixel value to each of its nodes. A moderate layer of 100 nodes would result in the network having $307,200$ weights only in the first layer. With larger images or more nodes this becomes completely infeasible to train.

Convolutional layers use convolutions to map certain areas of an image - or a previous layer of nodes - to a single node in the next layer. This is done with convolution filters, called kernels. Figure 2.6 describes the process. No nodes share the exact same area from the previous layer, and this reduces the dimensionality substantially. Convolutional layers does not only solve the dimensionality problem, it also brings some good intuition about how and what the network might learn. Convolution in general can detect certain patterns in data. It is good for detecting edges and shapes, or blobs of colors. Consider a kernel that detects edges, the scalar that the kernel outputs is a measure of "how much" edge was found in the area.

One question still stands. What kernels should be used in what occasions? What a kernel detects depends on the values within the kernel . These values are treated as weights, meaning the kernels are found in the learning process. This is powerful since the network is free to design its own kernels it finds useful. Figure 2.7 shows a visualization of the kernels learned by the first layer of the famous network Alexnet [27], which was the first deep convolutional neural network. The kernel size also affects what the kernel detects, but unlike the kernel weights, they are often set manually. Figure 2.6 uses a kernel of size 3×3, but any size is possible. Smaller kernels detects smaller details within the image. Bigger kernels in turn detects bigger details and are more computationally expensive.

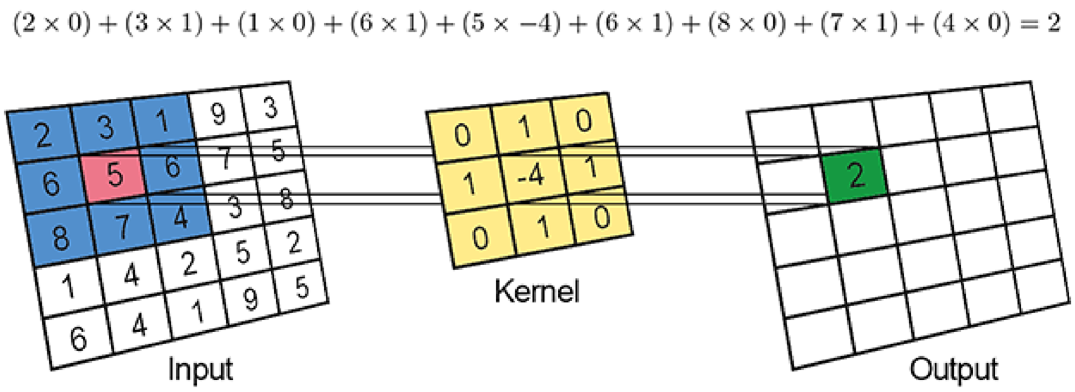
$(2 \times 0) + (3 \times 1) + (1 \times 0) + (6 \times 1) + (5 \times -4) + (6 \times 1) + (8 \times 0) + (7 \times 1) + (4 \times 0) = 2$
2 3 1 9 3
6 5 6 7 5
8 7 4 3 8
1 4 2 5 2
6 4 1 9 5
Input
0 1 0
1 -4 1
0 1 0
Kernel
2
Output

Fig. 2.6 Visual description of a convolution. Each input value is multiplied with the corresponding value in the kernel. Then all products are summed to produce the output value. Note that the output value is only connected to the green area from the input, whereas in a fully connected layer, it would be connected to all all input values. The kernel will slide along the input matrix to fill the output matrix with values. Although from the image one call tell there is a border in the output matrix that will not be filled. One way to solve this is to add a border of zeros on the input matrix. How large steps the kernel takes sideways is decided by the *stride*. Note that 3×3 convolution are being made on a 5×5 input. This is 2D convolution. Lets say the input is an RGB image with $5 \times 5 \times 3$ pixels. A kernel of size $3 \times 3 \times 3$ would then be used, but it would still produce a single value for each convolution step, i.e the output would still be of size 5×5. Normally, many kernels are used in parallel, and the number of kernels decides the third dimension of the output. If there are 32 kernels, the output would be of size $5 \times 5 \times 32$.

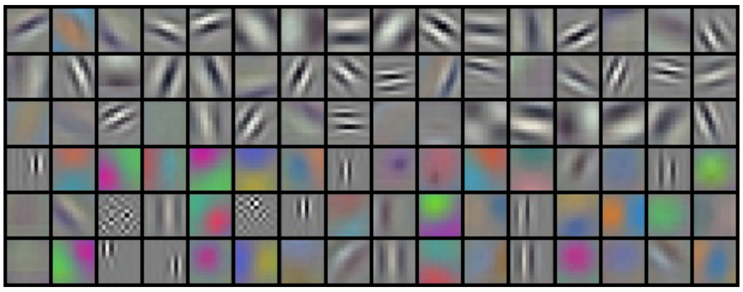
Fig. 2.7 The 96 kernels of size $11 \times 11 \times 3$ learned by the first layer in the deep convolutional neural network Alexnet. Image taken from [27].

2.2.3 Pooling Layers

If a networks output is to be of a smaller size than its input then somewhere within it the size needs to be reduced, which is also beneficial for computational reasons. This can be done with fully connected layers, but such layers introduces a lot of weights, could easily overfit and adds unnecessary complexity. Pooling layers allow us to reduce the dimensionality in a cheaper way, and like convolutional layers, they provide intuition about what is carried through from the previous layers. Pooling layers works like convolution kernels, but instead of learning its weights, they use some predefined function of the input area. Two common techniques are max and average pooling. A max pooling layer takes the largest value within the input area while average pooling takes the average value. Usually with pooling kernels, a stride is used that is large enough to only cover each pixel once. Figure 2.8 shows us how a max pooling layer reduces a feature map to a smaller feature map. An alternative is *global* pooling, which takes a feature map and produces only one output value. For example, a global max pooling layer takes the largest value in the whole feature map, in the case of figure 2.8 it would be 9. If a pooling layer is used after a convolutional layer they can be though of as extracting some of the information found by the convolution kernels. One way to say it is that max pooling kernels outputs a measure of the most prevalent feature found in the input area, and average pooling kernels outputs some measure of the overall quality of what was found in the previous layers.

2.3 Inception Module

In the case of convolutional neural networks, fully connected layers are mostly put close to the output layer. Earlier in the network, convolutional and pooling layers are used to learn features from the input data. Convolutional layers and pooling layers can use kernels of various sizes and it is hard to know what kernels to use in order to get the best results. The

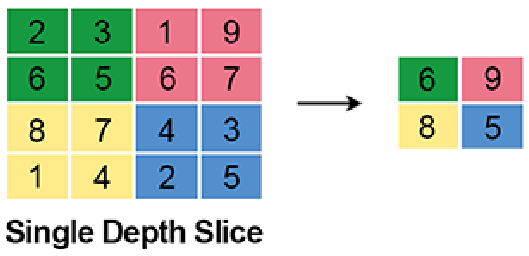

Fig. 2.8 Description of the result when using a max pool kernel that reduces the complexity from 16 to 4. The kernel have used a stride of 2 to cover each node only once.

inception module is a set of layers that intends to solve this decision problem by using various kernels at the same time, letting the network decide what it finds useful. These modules does not only build layers depth-wise but also breadth-wise using different types of layers in parallel.

Figure 2.9 shows two examples of inception modules that were introduced in [47], the authors used the module to the right to construct the first version of the inception networks. The left image (figure 2.9) shows the basic concept of using multiple layers in parallel to let the network decide what is useful. This is a problem since doing 3×3 and 5×5 convolutions directly on an input contributes with much complexity. Instead, as done by the model to the right (figure 2.9), convolution kernels of size 1×1 can be used to reduce the dimensionality substantially before using the larger convolutions. In contrast to pooling layers which does also reduce the complexity, 1×1 kernels also provide the ability to learn.

InceptionV3 - the architecture we chose to use in our method - was introduced in [48]. It uses inception modules but has made a number of improvements. For instance, it is recognized that a 5×5 kernel can be reduced to two 3×3 kernel layers coupled in series to reduce complexity. Also to reduce complexity, the idea is introduced to use asymmetric kernel sizes of for example 3×1 and 1×3 coupled in series or in parallel instead of one 3×3 kernel. Variations of these inception modules are stacked depth-wise to form a large CNN. Figure 2.10 shows a schematic diagram of the InceptionV3 architecture.

2.4 Deep Learning

Deep learning is a field within ML that focuses on deep neural networks (DNN). DNNs are characterized by multiple hidden layers and often use more complex architectures. Deep Learning has recently become very attractive within signal and sequence processing where the usage of convolutional layers have been used to great success [27]. More and more

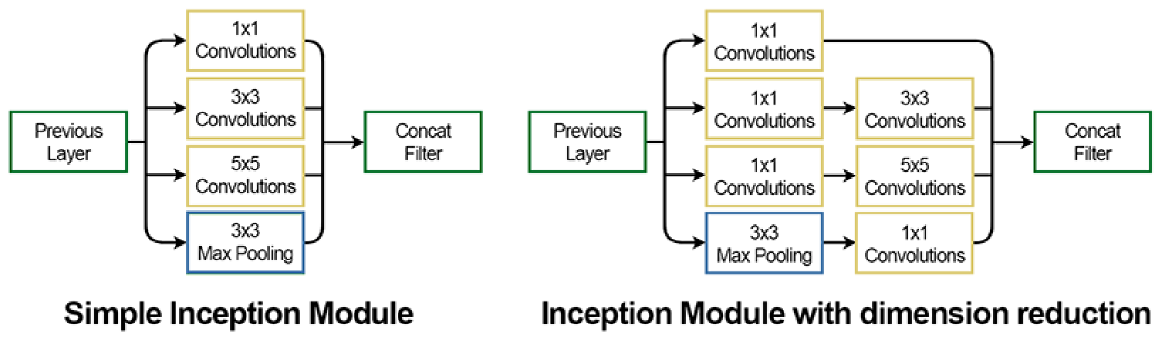

Fig. 2.9 The simple inception module (*left*) shows the basic concept of the inception module to use different kernel types and sizes in parallel. The module using dimension reduction (*right*) have added 1×1 convolutional layers to reduce the number of operations. Modules like these are often referred to as *network in network* layers.

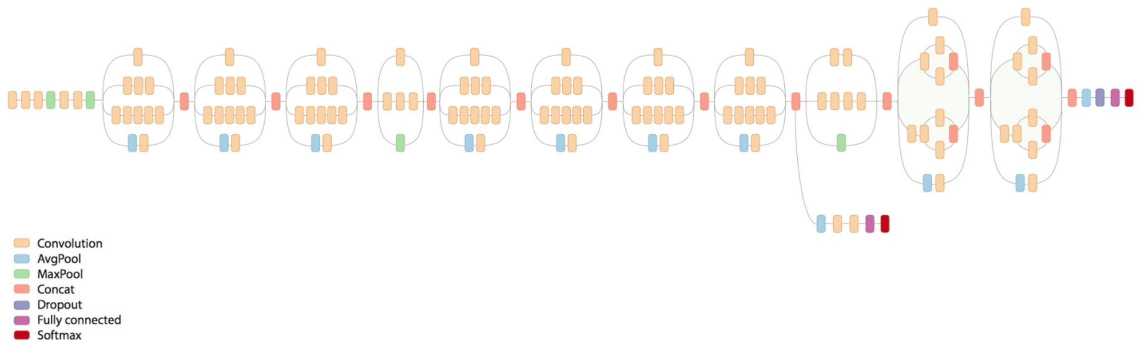

Fig. 2.10 Schematic diagram of the InceptionV3 architecture. Note that the network is branching out on the 8th inception module. This layer was simultaneously trained directly on the ground truth to ensure that the intermediate features were relevant. Image is taken from a blog post by Jon Shlens [44], one of the creators of InceptionV3.

ingenious designs like the inception module [48] has further improved the capabilities and have inspired current state-of-the-art networks [9] within image analysis.

The drawbacks of using deep learning instead of other learning algorithms such as SVM or Gradient Boosting are that it requires extremely large datasets and lots of computational power. The field is very ad-hoc and lacks a strong theoretical foundation, which makes the behavior of the models hard to understand. Much effort is still put into understanding why the current techniques works and if the intuitions generally used are correct. Despite this, DNNs has a big advantage over other methods in that they scale well. As the performance of other ML-methods eventually plateaus with larger datasets, performance has the potential to continuously increase with DNNs. Smarter and more cost-efficient architectures can make better use of data, and deeper and more complex architectures can utilize larger datasets.

2.5 Transfer Learning

Transfer learning is a field within ML that focuses on transferring knowledge between tasks. Knowledge transfer on ML models has long been used to make old models useful in new domains and to minimize or avoid expensive data labeling of new datasets. Source and target tasks as well as source and target domains are core concepts in transfer learning.

A task exists within a domain, e.g a task could be to classify horses within a domain that is a set of images. If the source and target task stay the same but the domain changes, it is called domain adaption, which is not the same as transfer learning. Although the terminology are sometimes mixed.

In transfer learning, a model is trained on a source task, and utilized on a target task. The more similar the source and target tasks are to each other, the more useful information can be expected within the model. The advantages of using transfer learning is primarily that it holds resource cost down. A smaller dataset is required to achieve good accuracy and one can many times expect a fraction of the training time to achieve the same accuracy. Sometimes even when the resources to train from scratch are present, higher accuracy can be achieved using transfer learning. This was for example shown in [53] using fine-tuning which will be described further down. Figure 2.11 describes three ways transfer learning might help improve performance if done right. The training process might start out with better accuracy, gain accuracy faster and end up with higher accuracy. Succeeding in transferring relevant information from the source to the target task is called positive transfer. Negative transfer is also possible and is the process of transferring information that has a direct negative effect on our outcome. There is often a trade-off between focusing on amplifying positive transfer or preventing negative transfer.

In this report we focus merely on transfer learning within the context of deep learning. Two of the most prevalent techniques in this area are feature extraction and fine-tuning. In this work we investigated feature extraction but will describe both of them as it is important to understand in what sense they stand in opposition to each other and why we chose to only delve deeper into one of them.

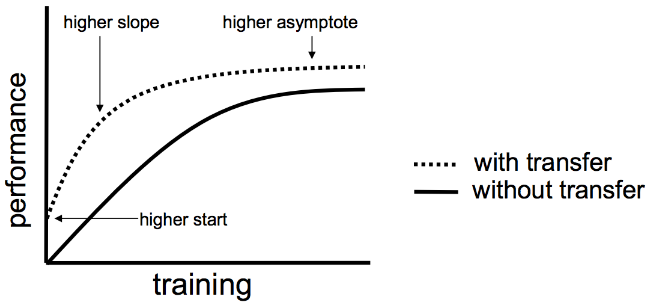

Fig. 2.11 Describing three ways transfer learning might improve performance. Know that there is no guarantee to benefit from any of them. Image is taken from [50].

2.5.1 Source Domain Considerations

As the field of transfer learning grows, it gets increasingly important to know what constitutes a good source domain. Despite that, there are few studies about this in the literature. Many have turned to ImageNet as the best transfer learning source domain due the large amount of classes and instances within each class. Arguments have been put forward claiming that the similarity between classes in ImageNet, for example different breeds of dogs, allows for CNNs to get fine-grained distinguish capabilities. Other arguments suggests that the amount of classes in ImageNet forces networks to learn generalizable features on different abstractions levels. In [22], the authors used different subsets of ImageNet to explore how the transferability changed. Their results was somewhat counter-intuitive as it suggested that none of the mentioned arguments had any significant effect on transferability. It is possible that the importance of the quantity of source datasets is overestimated. Much work needs to be done to assess this matter but [22] claims that the intuition-based arguments presented in various papers should be taken with a grain of salt. What we do know is that there is something about ImageNet that makes it a good source network in a transfer learning contexts.

2.5.2 Fine-tuning

If the target dataset is big and computational resources sufficient, one should consider using fine-tuning instead of feature extraction. Fine-tuning is a popular transfer learning approach and can be conducted in numerous ways and for different purposes. The idea is to use a pre-trained source network, change the output layer if necessary, and continue the training process on a new (target) task.

When fine-tuning, one has to keep in mind that early layers learn more general features and late layers learns features that are more specific to the source task, as shown by [53]. Some different approaches are:

- Only replace the output layer and continue to train the whole network. When starting to train a new network, one needs to initialize the weights. This approach can be used as a weight initialization technique that can be beneficial even in cases where one can afford to train a network from scratch. Know that in a case where the source and target tasks are the same but the domains are different, the output layer need not be replaced, however, this is often called domain adaption. In [17], this fine-tuning technique was compared to training a network from scratch. The authors used the CaffeNet [23], architecture in a face verification setting where they increase the accuracy significantly when initializing with pre-trained weights instead of random initialization.
- Replace the n latest layers and continue to train the new network, i.e, only fine-tuning the m earliest layers, where $m+n$ is the total number of layers in the pre-trained network. This could be done if the source and target tasks are not similar and one suspects that the features in the n latest layers are too specific to the source task and could result in worse performance due to negative transfer.
- In combination with any of the two earlier points, a common approach is to *'freeze'* some number of the earliest layers. The weights in these layers will not be tuned in the training process. This spare some computational resources during training and can be a good idea if the features learned in the frozen layers are general enough to contribute with positive transfer. This can be viewed as a combination of fine-tuning and feature extraction.

If the target dataset is too small, fine-tuning runs the risk of overfitting. In that scenario feature extraction should be used instead.

2.5.3 Feature Extraction

One approach to transfer knowledge from a source task to another is to view DNNs as feature extractors. The output from each layer can be interpreted as a set of features the network found useful during training on the source dataset. If the source and target domains are similar enough, there is a good chance that some of these features are applicable in both domains. If the target dataset are passed through the network, the output from different layers can be used as features for some other classifier. What layers to extract features from depends on the source and target task similarity. As previously mentioned, late layers in networks generally produces more source task specific features and earlier layers produces more general features. The outputs from the internal layers in a network is often highly dimensional. Table 4.1 shows the dimensionality of the features from different layers in the source network used in this report. There are different approaches if one wishes to reduce the dimensionality from feature maps to feature vectors. Average or max pooling, as well as fully connected layers are common and convenient methods for this purpose.

During the feature extraction based transfer learning, a deep neural network never has to be trained. Given that the features are any good, this is a huge advantage since training a DNN is such a costly process. Instead, one can train a simpler classifier on the features and still get impressive results. In [28] the authors used features extracted from the Overfeat network [41] to train a classifier to detect cancerous tissue. They found that their method produced accuracy on the same level as previous work using hand crafted features. Hand crafting features is a costly process and this shows how feature extraction as a transfer learning method can save resources and still produce competent models. One disadvantage of feature extraction is that the performance will never be better than the features allows for, so the choice of pre-trained source network is important. Still the feature extraction approach should be considered as a first alternative when resources are sparse in any form.

2.6 Visual Analytics

The goal of Visual Analytics is to create technology that combines the strength of human and electronic data processing [24] and enables analysts to view the underlying processes in order to come up with new hypotheses. In a society which produces data at an ever increasing rate, tools are needed to turn raw data into valuable insights. Modern visual-analytics approaches [19] make use of charts, histograms and scatter plots to explore data. While useful in a low-dimensional context where it is only relevant to look at a couple of data variables at a time they cannot be used to plot high-dimensional data.

Visualizations of image features could be a great way to gain insights about the semantic contents of image datasets. In order to visualize and understand high-dimensional vectors they have to be reduced to three or two dimensions. To accomplish this different dimensionality reduction techniques can be used.

We will briefly describe some different concepts through some examples and in relation to deep convolutional feature extraction. Visualizations of the different techniques can be found in figure 2.13.

2.6.1 Linear Projections

The easiest way of reducing the dimensionality of data is to do a random linear projection, [5]. It takes data in a space $\mathbb{R}^d$ into a lower dimensional space $\mathbb{R}^k$ using a random $d \times k$ dimensional matrix where the sum of each row is one. The result is a d-dimensional representation of ones data which preserve the distance between points well but does not take the structure of the data into account. It is a very efficient way of doing dimensionality reduction but when working on extracted features it does not result in very interesting plots.

2.6.2 Linear Decomposition

Principal Component Analysis (PCA) [43] takes data in a space $\mathbb{R}^d$ and transforms them into a new space where all dimensions are linearly uncorrelated. This can be viewed as rotating the axes spanning $\mathbb{R}^d$ to eliminate linear correlation within the data and it is useful since it reduces redundancy within the data. Eigenvalue decomposition [38] is often used in order to find eigenvectors which constitutes the new axes in PCA. The new axes are called principal components and are ranked in such a way that the first component has preserved the largest variance within the data, and the last has preserved the least. This is visualized by figure 2.12.

When the principal components are ranked by variance, one could drop the least important components to reduce the dimensionality. Observations often contain noise that is captured mainly in these last components. Dropping them can potentially reduce the noise within the measures, which could be an effective pre-processing step. One could also drop all but the first two components to be able to visualize the data directly. The two-dimensional plot is an approximation that often capture a lot of the structure of the data.

2.6.3 Non-linear Embeddings

When doing dimensionality reduction it is often a problem that a lot of the inherent high-dimensional structure of the data is lost. To deal with this one has turn to non-linear embedding techniques that tries to find more complex transformations. In Manifold Learning [16] (*not ML*) the assumption is made that many complex high-dimensionality datasets are governed by a comparatively few number of parameters. The goal of these types of algorithms is to reduce the dimensions of the data without losing information about how the data is structured.

There are many types of manifold learning algorithms that often produce very different mappings. Some examples are: Isometric Feature Mapping (ISOMAP) [2], Locally Linear

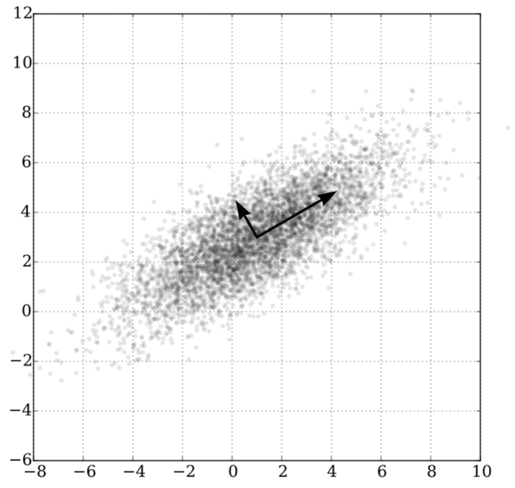

Fig. 2.12 PCA without dimensionality reduction done on 5,000 data points [34]. The black arrows are the new axis which eliminates the linear correlation.

Embedding (LLE) [40] and t-Distributed Stochastic Neighbor Embedding (t-SNE) [52]. When testing different embeddings on extracted features the t-SNE algorithm was found to produce the most interesting mapping.

t-SNE tries to reduce the dimensionality of data in a way that preserves the similarity between data points. It is typically used to create a low-dimensionality embedding of high-dimensional data. The approach is to model a distribution over pairs of points in a way that similar points have a high probability of being picked whilst dissimilar points have low probabilities. This is done one time in the true dimension space and another time in the target dimension space, with two slightly different probability distributions. Then the Kullbeck-Lieger divergence is calculated to measure the divergence between the distributions. t-SNE is an iterative method and uses gradient descent to minimize the Kullbeck-Lieger divergence. Euclidean distance is the standard measure of similarity between points, but this measure can be varied if appropriate. t-SNE is very computationally heavy since it considers every pair of points and it also falls victim to the curse of dimensionality since it often uses euclidean distance to measure similarity [12]. The algorithm is not considered reliable to use for clustering since it does not necessarily preserve distances between points. Another disadvantage of t-SNE is that it is *data dependent*, it can not be used to transform new data points. Instead it only outputs the transformed points and has to be recalculated on new data. There are numerous implementations of t-SNE, in figure 2.13 we used the version as described in [51], that uses the faster Barnes-Hut algorithm.

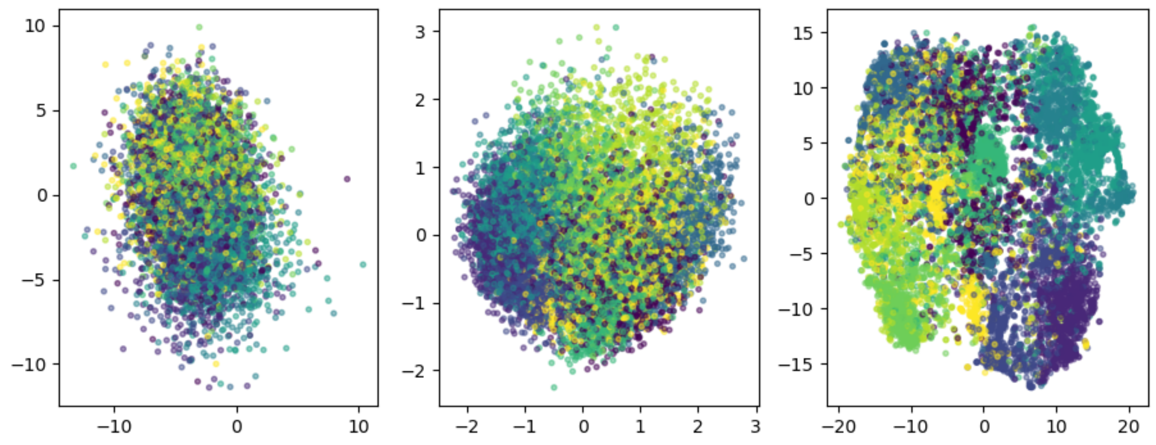

Fig. 2.13 Features extracted from 10,000 images and colored after their content. The extracted vectors where of 2048 dimensions. The projection plot (*left*) shows the vectors after have gone through a random linear projection. The decomposition plot (*middle*) shows the features after being reduced to two dimensions using linear PCA. The embedding plot (*right*) was created by doing PCA down to 50 dimensions, then using a t-SNE embedding down to two dimensions. Although the t-SNE plot seem to have captured a lot more of the high-dimensional structures it has not been verified that this type of visualization of extracted features is a good way of qualitatively assessing the usefulness of said features. Best viewed in color.

Chapter 3

Experiments

Experiments we conducted along the way before reaching the chosen method. While not integral to understanding the results it could be interesting for a reader what decisions was made along the way.

3.1 Choosing a Classifier

A few classification techniques was experimented with before we decided to use SVMs to produce the results. First, a single softmax layer was tested, but with slightly worse results than SVMs. Then we added a fully connected layer with a ReLU activation function before the softmax layer, but this technique quickly started to overfit the training set. Both approaches was tested with different learning rates and regularization as well as optimization techniques without satisfying results. We settled for SVM fairly quickly as it gave reliable and robust results. It is likely that higher accuracies could be achieved with some further experiments.

3.2 Average vs. Max Pooling

We compared global average pooling with global max pooling as techniques for producing feature vectors from the feature maps outputted by the convolutional layers. The authors in [14] argues that max pooling adds position robustness since it passes through the most representative instance from its input, and therefor helps them in a setting where multiple classes can be found within one single image. Though they did not use *global* max pooling but max pooling in combination with a softmax classifier, the paper inspired us to try global max pooling. But it did not give as good results in SVM classification accuracy on the experiments we ran on the CIFAR-10 dataset. The accuracy on a few samples was $\sim 10-20\%$ worse, and we decided to continue our experiments using average pooling.

3.3 Preprocessing

In [1], the authors use dimensionality reduction to find the effective dimensionality of feature vectors extracted from the fully connected layers of a deep network. While 500 seemed to be the optimal length, vectors that were longer than that did not have a great negative impact on performance. SVMs did not seem to have much problem dealing with the longer vectors. In [4], the authors argue that while learning performance can be improved by removing the low variance components, it is generally better to use sparse and overcomplete representations than dense and undercomplete ones when working with discriminant learning algorithms. With this in mind we chose to not use dimensionality reduction on our feature vectors.

We tried different preprocessing techniques including vector standardization, normalization and decomposition. Since all of our feature vectors were extracted using average pooling the resulting vectors were often well formed with the majority of values ranging between 0 and 1. While some techniques like PCA looked promising we opted for not performing any scaling or decompositions since we did not see much improvement in classification accuracy on the experiments we ran on the CIFAR-10 task. It is possible that some of these techniques would have yielded better classification performance but since our focus is on comparing accuracies it would not have had much impact on our results.

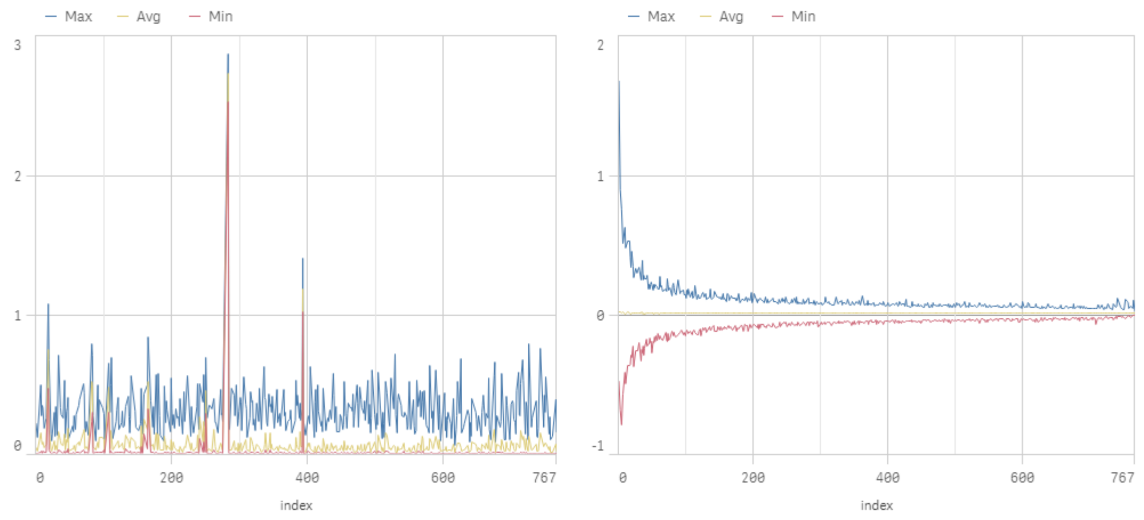

Fig. 3.1 Using PCA as a preprocessor. The left plot shows the min, average and max value for the 10,000 feature vectors from the CIFAR-10 test set. The right plot shows the same vectors but after being transformed through a PCA that was trained on the 50,000 feature vectors in the CIFAR-10 training set. The feature vectors where extracted from the 8th layer of our source network and are of length 768. Note that the transformed vectors are of the same length as the base ones, no dimensionality reduction was performed.

Chapter 4

Method

Our method of analysis can be divided into three chronological parts: Dataset Manipulation, Feature Extraction, Classification and Evaluation.

4.1 Dataset Manipulation

CelebA provides 40 different face attributes and a visual assessment was made to choose two of them. Two binary datasets was created from the attributes *Smiling* and *Young* with $60,000$ images in each and with the same amount of negative as positive examples. Each dataset was further divided into a training set of $50,000$ images and a test set of $10,000$, with the same ratio of labels in the train and test set. The MNIST dataset was used with the pre-defined configuration of $60,000$ train samples and $10,000$ test samples. Also on CIFAR-10 we used the pre-defined configuration which has $50,000$ train samples and $10,000$ test samples.

To be able to use the input layer of our source network, all images had to be resized to 299×299, using nearest neighbor interpolation when necessary. All pixels where then also scaled between -1 and 1. Each image was scaled individually.

4.2 Feature Extraction

The features were extracted using a pre-trained InceptionV3 architecture acquired from the Keras framework version 2.1.5 (downloaded on 08-02-2018). The feature maps were extracted from the outputs of all 11 inception modules, see figure 2.10 for a visual description of the network.

4.2.1 Average Pooling

A common approach to reduce the dimensionality of feature maps is to use fully connected layers and structure the output layer as the desired feature vector. Since these layers require training they are not always convenient in a feature extraction setting. In [30] the authors proposes the use of global average pooling to vectorize feature maps. Though they present many arguments for why it could be a preferable method during training of the network, we use it to avoid the inconvenience of training a fully connected layer and to keep our method relevant also in unsupervised settings. As mentioned in the scientific background, table 4.1 shows the shape of the features extracted from the network, and the shape of the features after performing global average pooling.

Table 4.1 The shapes of the features extracted from the convolutional layers of our source network. The target shape is the shape if the input to the classifier and was achieved by using global average pooling on the extracted features.

Layer	Shape	Target shape
1	35 x 35 x 256	256
2	35 x 35 x 288	288
3	35 x 35 x 288	288
4	17 x 17 x 768	768
5	17 x 17 x 768	768
6	17 x 17 x 768	768
7	17 x 17 x 768	768
8	17 x 17 x 768	768
9	8 x 8 x 1280	1280
10	8 x 8 x 2048	2048
11	8 x 8 x 2048	2048

4.2.2 Combining Features

In our research we also wanted to combine features from different layers to see how the classification performance was affected. In order to do this without getting too many combinations of vectors we chose to only construct groups of vectors adjacent to each other. While other feature combinations are certainly possible we thought this subset to be the best compromise between interesting and feasibility.

Table 4.2 Visualizing the different groups of features that we created by concatenating feature vectors extracted from adjacent layers. There is a total of 66 combinations of vectors.

Group	1	2	3	…	11
	1	1+2	1+2+3	…	1+2+3+4+5+6+7+8+9+10+11
	2	2+3	2+3+4	…	
Layer	⋮	⋮	⋮	…	
	9	9+10	9+10+11		
	10	10+11			
	11				

4.3 Classification

L2-SVMs using the hinge loss function and a $C = 1$ was used as the classification technique. Since SVMs are binary classification methods they could only be used as is on the *Smiling* and *Young* datasets. In the multi-class case of MNIST and CIFAR-10, many L2-SVMs was trained together in a one-vs-all scheme.

4.4 Evaluation

To evaluate the classification models we used the Jaccard Index similarity score [39] for both the binary classification tasks as well as for the multiclass tasks. Jaccard Index is defined as the size of the intersection divided by the size of the union between two sets. The formula can be written as

$$J(A,B) = \frac{|A \cap B|}{|A| + |B| - |A \cup B|}. \tag{4.1}$$

We use this score on the predicted classes and the ground truth of the test set. A greater score means the prediction where closer to the ground truth and therefore better.

Chapter 5

Results

In this chapter we show the results of our experiments. First the achieved accuracies across the datasets are presented in tables. Then there are graphs that describe the nature of our findings and aid in the upcoming discussion.

5.1 Accuracy Tables

Table 5.1 Top 10 best features ranked after their main Jaccard score across the four target domains. Note that long features does not necessarily mean better SVM classification performance and that the best scores are very similar. Shorter features of size one and two does not make it onto the list but some are not far off.

Layers	Size	Avg. Score
5+6+7+8	4	0.8911
4+5+6+7+8+9	6	0.8911
2+3+4+5+6+7+8+9	8	0.8899
1+2+3+4+5+6+7+8	8	0.8896
5+6+7+8+9	5	0.8883
6+7+8+9	4	0.8878
4+5+6+7+8	5	0.8873
6+7+8	3	0.8870
2+3+4+5+6+7+8	7	0.8870
3+4+5+6+7+8+9	7	0.8870

Table 5.2 Top 10 worst features ranked after their mean Jaccard score across the four target domains. Notable is that layer 11 - which is the top layer of the network and one of the features commonly used for transfer learning - makes it onto the list.

Layers	Size	Avg. Score
1	1	0.7357
2	1	0.7739
1+2	2	0.7754
1+2+3	3	0.7789
3	1	0.7856
2+3	2	0.8042
1+2+3+4+5	5	0.8187
1+2+3+4	4	0.8207
11	1	0.8248
3+4	2	0.8263

Table 5.3 Table containing the evaluation of SVMs trained on feature vectors of varying length and in different adjacent groupings. Note that the best features are often from the middle of the network. Also the best feature for each dataset are often from different layers. The difference could give some insight in how the target domains differ from the source domain.

Index	Layers	Length	MNIST	CIFAR10	CelebA	
			Multi	Multi	Young	Smiling
0	1	256	0.8722	0.5756	0.7588	0.7360
1	2	288	0.9264	0.6358	0.7639	0.7694
2	3	288	0.9449	0.6509	0.7679	0.7786
3	4	768	0.9744	0.7040	0.7952	0.8435
4	5	768	0.9852	0.7687	0.8139	0.8562
5	6	768	0.9872	0.7817	**0.8215**	0.8471
6	7	768	**0.9911**	0.8088	0.8147	0.8570
7	8	768	0.9852	0.8266	0.8027	0.8422
8	9	1280	0.9890	**0.8349**	0.8157	**0.8611**
9	10	2048	0.9857	0.8179	0.8067	0.8426
10	11	2048	0.9770	0.7844	0.7872	0.7506
11	1+2	544	0.9162	0.6216	0.7856	0.7780
12	2+3	576	0.9541	0.6671	0.7884	0.8070
13	3+4	1056	0.9751	0.6808	0.8019	0.8472
14	4+5	1536	0.9864	0.7604	0.8064	0.8716
15	5+6	1536	0.9903	0.8048	0.8283	**0.8778**
16	6+7	1536	0.9921	0.8260	**0.8291**	0.8741
17	7+8	1536	**0.9922**	**0.8435**	0.8237	0.8715
18	8+9	2048	0.9893	0.8366	0.8208	0.8679
19	9+10	3328	0.9901	0.8269	0.8157	0.8630
20	10+11	4096	0.9843	0.7958	0.7960	0.8380
21	1+2+3	832	0.9553	0.6349	0.7277	0.7978
22	2+3+4	1344	0.9742	0.7258	0.8039	0.8080
23	3+4+5	1824	0.9840	0.7833	0.8171	0.8752
24	4+5+6	2304	0.9901	0.8047	**0.8350**	**0.8873**
25	5+6+7	2304	0.9920	0.8314	0.8316	0.8845
26	6+7+8	2304	**0.9929**	**0.8489**	0.8260	0.8802
27	7+8+9	2816	0.9919	0.8451	0.8249	0.8810
28	8+9+10	4096	0.9901	0.8270	0.8175	0.8667
29	9+10+11	5376	0.9881	0.8068	0.7990	0.8420

Index	Layers	Length	MNIST	CIFAR10	CelebA	
			Multi	Multi	Young	Smiling
30	1+2+3+4	1600	0.9693	0.6892	0.7957	0.8284
31	2+3+4+5	2112	0.9839	0.7234	0.8218	0.8710
32	3+4+5+6	2592	0.9895	0.7927	0.8122	0.8707
33	4+5+6+7	3072	0.9922	0.8305	0.7488	0.8878
34	5+6+7+8	3072	**0.9932**	**0.8524**	**0.8297**	**0.8892**
35	6+7+8+9	3584	0.9919	0.8464	0.8286	0.8842
36	7+8+9+10	4864	0.9917	0.8350	0.8211	0.8749
37	8+9+10+11	6144	0.9882	0.8117	0.7994	0.8570
38	1+2+3+4+5	2368	0.9851	0.7529	0.6706	0.8661
39	2+3+4+5+6	2880	0.9884	0.7838	0.8251	0.8203
40	3+4+5+6+7	3360	0.9921	0.8293	0.8279	0.8812
41	4+5+6+7+8	3840	**0.9934**	0.8422	0.8256	**0.8879**
42	5+6+7+8+9	4352	0.9922	**0.8465**	**0.8290**	0.8854
43	6+7+8+9+10	5632	0.9921	0.8379	0.8225	0.8791
44	7+8+9+10+11	6912	0.9904	0.8202	0.7901	0.8627
45	1+2+3+4+5+6	3136	0.9904	0.7971	0.8293	0.8842
46	2+3+4+5+6+7	3648	0.9921	0.8248	0.8284	0.8893
47	3+4+5+6+7+8	4128	**0.9934**	0.8454	0.8251	0.8781
48	4+5+6+7+8+9	5120	0.9933	**0.8484**	**0.8319**	**0.8906**
49	5+6+7+8+9+10	6400	0.9927	0.8390	0.8258	0.8806
50	6+7+8+9+10+11	7680	0.9907	0.8251	0.8041	0.8660
51	1+2+3+4+5+6+7	3904	0.9923	0.8140	**0.8225**	0.7669
52	2+3+4+5+6+7+8	4416	0.9932	0.8453	0.8189	**0.8902**
53	3+4+5+6+7+8+9	5408	**0.9933**	**0.8481**	0.8159	0.8859
54	4+5+6+7+8+9+10	7168	0.9932	0.8421	0.8150	0.8831
55	5+6+7+8+9+10+11	8448	0.9909	0.8272	0.7769	0.8668
56	1+2+3+4+5+6+7+8	4672	0.9926	0.8459	0.8307	**0.8891**
57	2+3+4+5+6+7+8+9	5696	**0.9934**	**0.8489**	**0.8318**	0.8857
58	3+4+5+6+7+8+9+10	7456	0.9927	0.8394	0.7945	0.8506
59	4+5+6+7+8+9+10+11	9216	0.9908	0.8289	0.8057	0.8633
60	1+2+3+4+5+6+7+8+9	5952	0.9924	0.8270	**0.8277**	**0.8842**
61	2+3+4+5+6+7+8+9+10	7744	**0.9933**	**0.8426**	0.8227	0.8711
62	3+4+5+6+7+8+9+10+11	9504	0.9911	0.8308	0.8062	0.8741
63	1+2+3+4+5+6+7+8+9+10	8000	**0.9925**	**0.8443**	**0.8100**	**0.8777**
64	2+3+4+5+6+7+8+9+10+11	9792	0.9908	0.8307	0.8048	0.8733
65	1+2+3+4+5+6+7+8+9+10+11	10048	**0.9908**	**0.8318**	**0.6751**	**0.8789**

5.2 Single Features

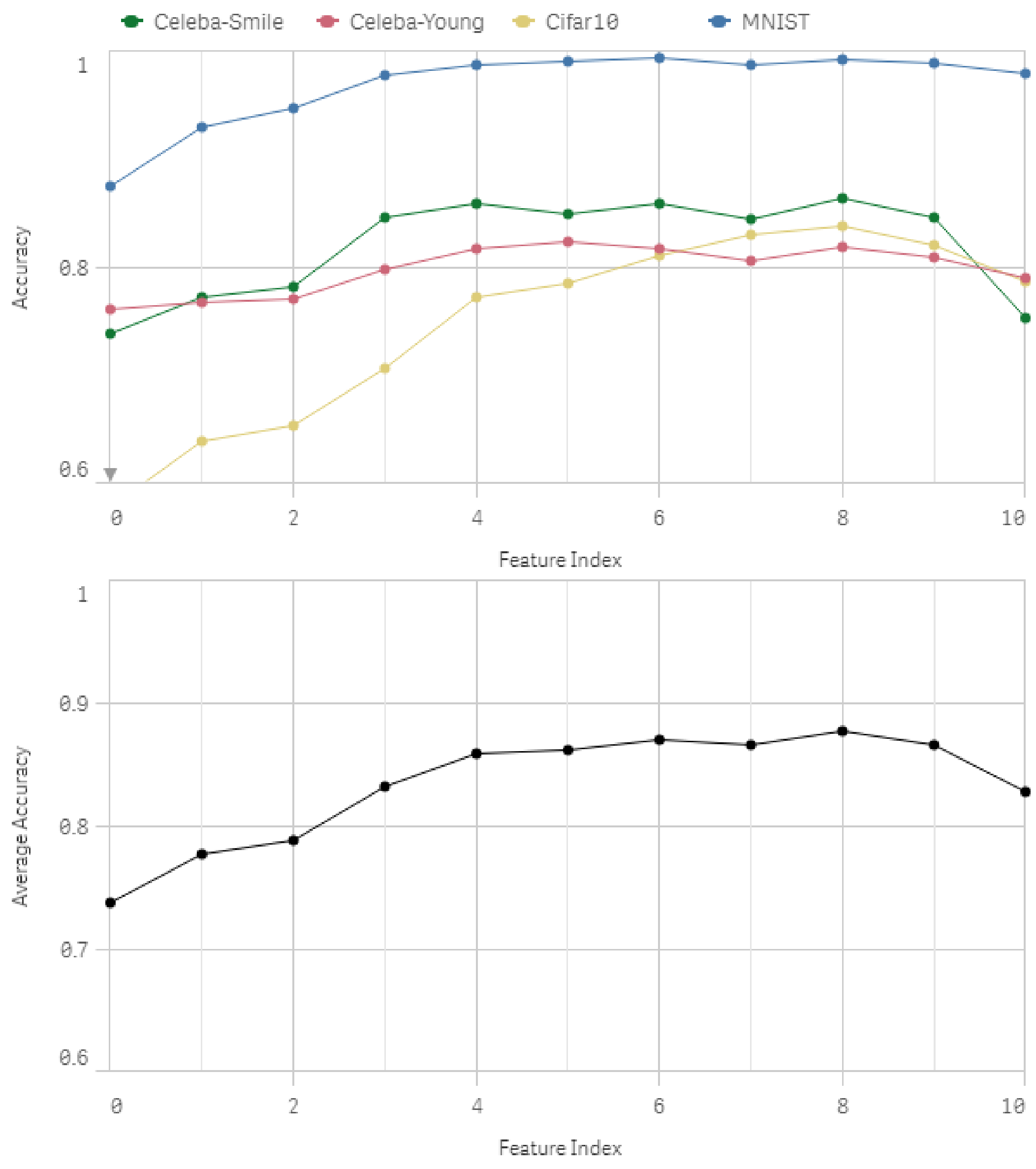

Fig. 5.1 Plots of the Jaccard score accuracy achieved when training SVMs on the base features without grouping adjacent features. The x-axis is the index of the feature used as defined in Table 5.3. Note that the feature vector with index 8 is always better than the vector of index 9 and 10 on all of our classification tasks. This is interesting since it is common to use the feature from the top layer, but these results suggest that using features a couple of layers down is always better. The reader should also note the steep improvement of performance on CIFAR-10 in the upper plot.

5.3 Grouped Features

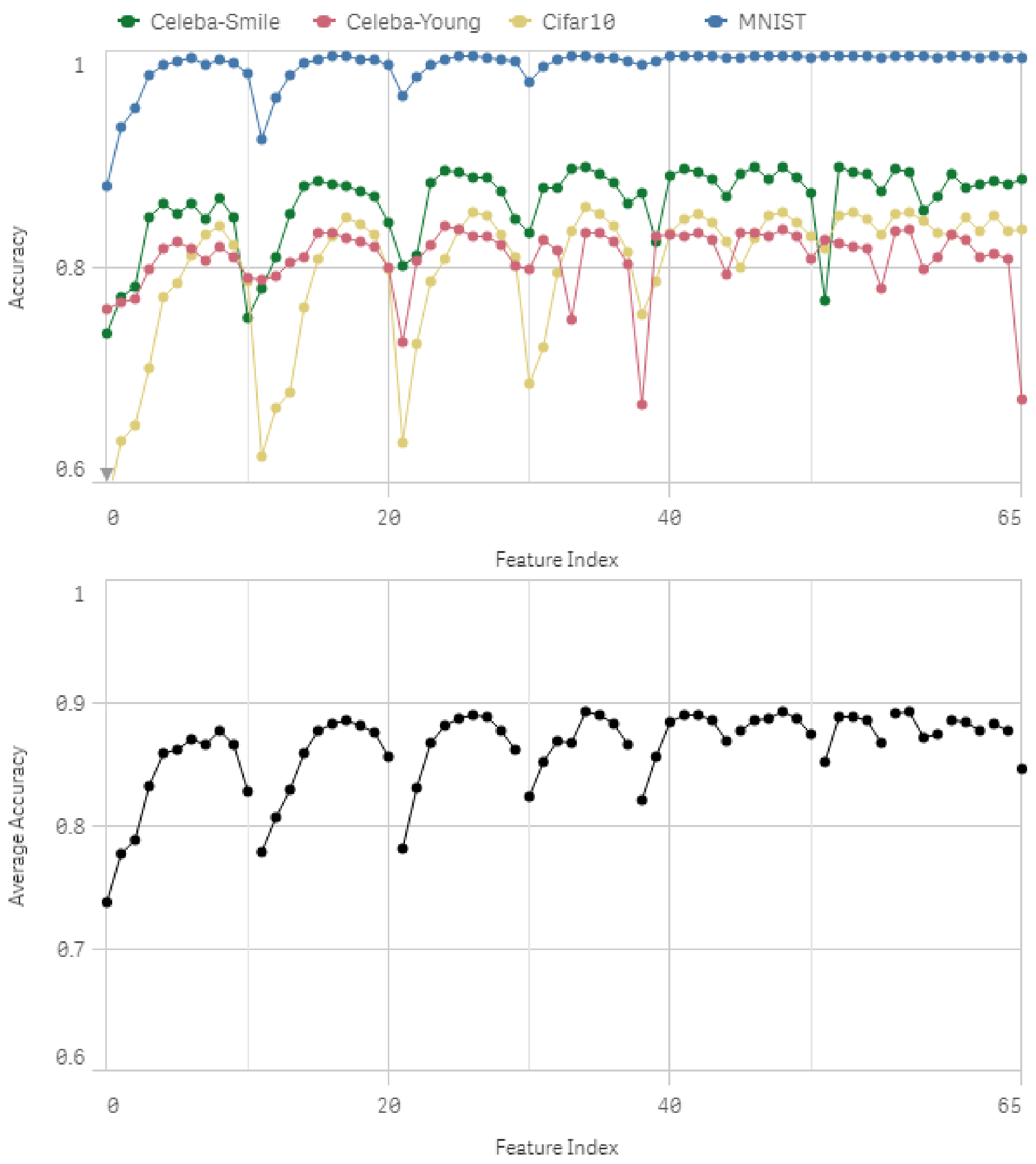

Fig. 5.2 Plots of the Jaccard score accuracy achieved when training SVM models on the base features and also grouping adjacent features. The x-axis is the index of the feature used as defined in Table 5.3. Note that any feature including the first or last layer is worse than features from the middle of the network. Also grouping two, three or four adjacent features together seems to improve accuracy. After that the results seem to plateau or even decline.

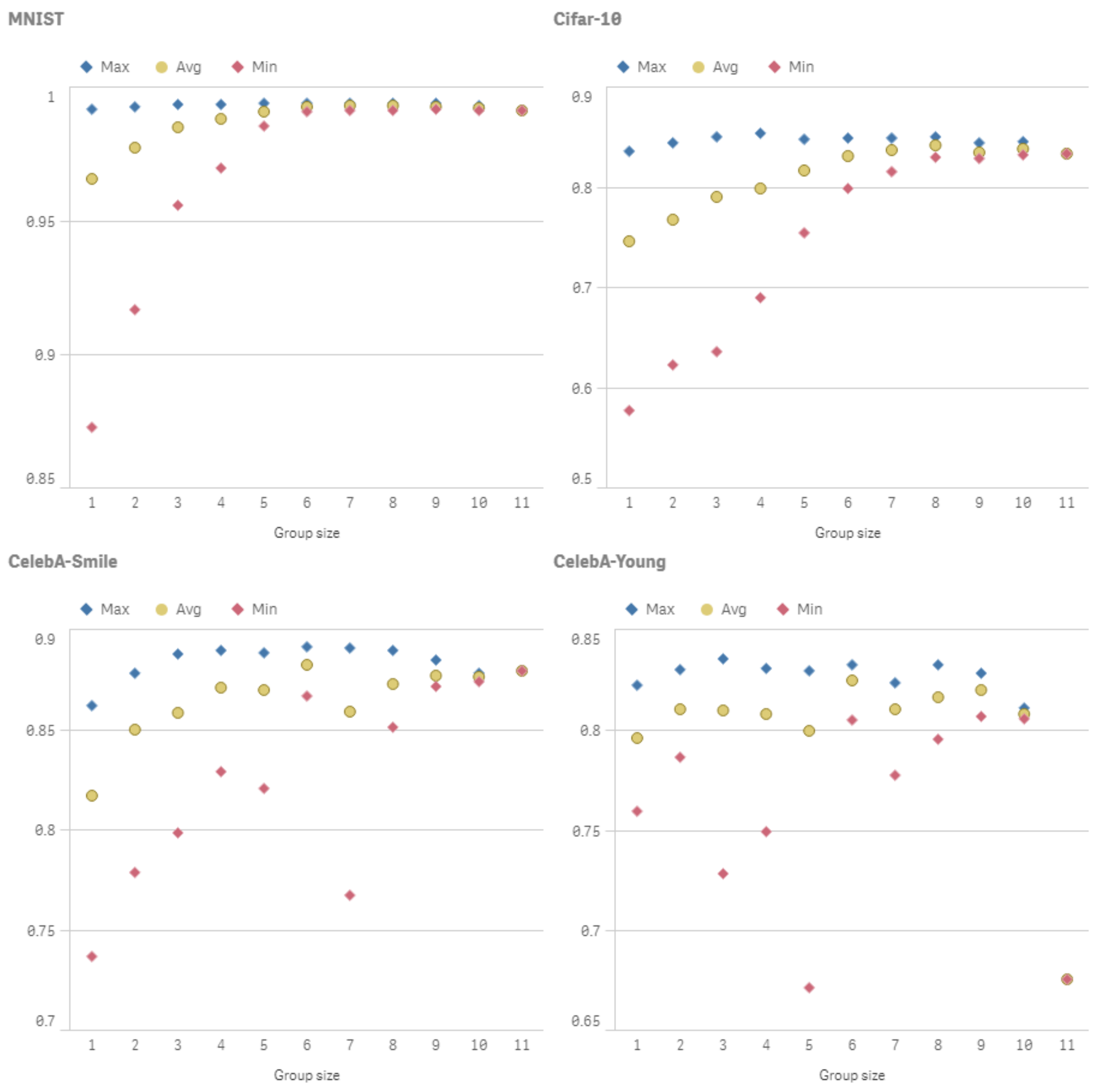

Fig. 5.3 Plots of the average as well as min/max variance in accuracy achieved for groups of features. The x-axis is the number of feature vectors being concatenated as seen in Table 5.3. Note that the max accuracy increases when starting to group features but plateaus or even declines as the groups get bigger. Also noteworthy is that some group sizes on the CelebA features performed significantly worse and did not follow the overall trend.

Chapter 6

Discussion

One observation that immediately struck us were that among the single features in figure 5.1 the features from the last (11th) layer of the network was reliably worse than earlier features for all of our target tasks. This is also shown in table 5.2 where it qualifies as one of the worst features among the ones we tried. This is interesting because a popular method of transfer learning is to train a new classifier on the last layer of a source network. On our tasks the features from the 9th layer had the highest average score and using layers later than that had consistently worse classification accuracy. Although layer 9 had the highest average accuracy, the depth of the best performing layer varies a lot between the different problems. The variation is big both between the different datasets but also between different tasks on the same domain, as in the CelebA case. Determining whether a person was Young or not worked best using the 6th layer while determining whether a person was Smiling or not got better results using the 9th layer.

Another interesting trend in figure 5.1 is that features already at the 5th layer have a good enough semantic representation to beat many of the later features in classification accuracy. The exception is CIFAR-10 that have a significant improvement all the way to the later layers. One theory is that this is because the CIFAR-10 task is similar to the ILSVRC source task in that their classes describes the same types of animals and objects. When transferring to solve the CIFAR-10 task the later semantic representations are useful even though they are starting to get more specific towards ILSVRC. Symmetrically the reason for the CelebA tasks to plateau earlier could be explained by the tasks being - in the same sense - less similar to the ILSVRC task, making it benefit less from the later representations.

When it comes to grouped features we can see in table 5.3 and by looking at figure 5.2 that there was a slight improvement in performance when concatenating features from different depths. Groups of length two, three and four in particular saw an improvement in classification accuracy and by looking at table 5.1 we can see that the best features were all of size three and bigger. This can also be viewed in figure 5.3 where you also can see the variance in each group. Groups of size bigger than four did not further improve the accuracy much and in many cases resulted in worse performance than smaller groups. Since the smaller groups are always contained within the bigger ones this phenomenon seem a bit strange. We argue that it could be due to the fact that the feature dimension gets too big for the SVM to handle as efficiently. The SVM might make less optimal trade-offs between trying to classify correctly and generalize well. Also, SVMs uses euclidean distance on wrongly classified points during training, which is a decreasingly accurate way of calculating distances as the number of dimensions grow.

Another observation we made in table 5.3 was that for the tasks that had the 9th layer as their best single features (CIFAR-10 and CelebA-Smiling), the ninth layer was not included in the best consecutive groupings. Also here we argue that a reason for this could be that the features from the 9th layer is much larger than features from earlier layers and when combining the 9th-layer features with others the dimensionality makes it so the SVM can not find as good margins. The curse of dimensionality brought more penalty than the features themselves gave value to the classification.

The reasoning that we gave about tasks that are closer to the source task should benefit more from the later semantic representations also holds for the grouped features. Concatenated features from earlier in the network is reliably better on the tasks that where farther from the ILSVRC source task. This is especially obvious in the case of CIFAR-10 in figure 5.2 where the peak of each "wave" comes at a much later point.

Whether or not these observations hold true for other deep neural networks is hard to know. It is for example not necessarily true that the first and last hidden layers generates poor results for networks twice or half as deep. In [13] a network of 7 layers (excluding the output layer) is used and similar results are demonstrated. They found that the 6th layer is on average optimal for feature extraction. In [53] it was shown that features from layers too close to the output often provided negative transfer if the target domain differed from the source domain. In agreement with these previous papers, our results tells us that depth is an important factor to explain the generality of different layers, but other factors as breadth of the network or different architectures has not been discussed as exhaustively. More research addressing these matters should be conducted before drawing too strong conclusions about how well our results generalize to other source networks.

6.1 Conclusion

We found that it is suboptimal to use features extracted from the last convolutional layer of a sufficiently deep source network, especially if the target task is not very similar to the source task. Instead the optimal feature depth depend both on how similar the source and target datasets are and on what type of problem the datasets represents. We also found that it is reliably better to combine multiple adjacent features in order to boost classification performance. When combining features one should avoid using the first and last features and instead include features closer to the middle of the network. A rule of thumb can be to try earlier features when expecting the target domain to be far from the source domain, and later ones when working on problems that you suspect are similar to the original task.

6.2 Future Work

When working on this project we thought we saw a correlation between SVM classification performance and how well low-dimensional embeddings like t-SNE where able to visually separate feature vectors. A qualitative study of how high-dimensional features could be made to research if good features could be distinguished from worse ones through low-dimensional embeddings and visualizations. A study like this would be especially relevant for the usage of extracted features in an unsupervised setting.

It would be interesting to see other methods that are similar to ours but that vary some variables. Such variables could for example be the choice of source network, exploring other ways of extracting features or choosing other target problems. While our findings show distinct trends we can not say much about how these results would vary given a different CNN architecture. In future work our method could be used on multiple other source networks to see how the feature transferability are dependent on the chosen architecture.

References

[1] Azizpour, H., Razavian, A. S., Sullivan, J., Maki, A., and Carlsson, S. (2014). From generic to specific deep representations for visual recognition. *CoRR*, abs/1406.5774.

[2] B. Tenenbaum, J., Silva, V., and C. Langford, J. (2001). A global geometric framework for nonlinear dimensionality reduction. 290:2319–23.

[3] Baldi, P. (2011). Autoencoders, unsupervised learning and deep architectures. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27*, UTLW'11, pages 37–50. JMLR.org.

[4] Bengio, Y. (2012). Deep learning of representations for unsupervised and transfer learning. In Guyon, I., Dror, G., Lemaire, V., Taylor, G., and Silver, D., editors, *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, volume 27 of *Proceedings of Machine Learning Research*, pages 17–36, Bellevue, Washington, USA. PMLR.

[5] Bingham, E. and Mannila, H. (2001). Random projection in dimensionality reduction: Applications to image and text data. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 245–250, New York, NY, USA. ACM.

[6] Bock, H.-H. (2008). Origins and extensions of the k-means algorithm in cluster analysis. *Electronic Journ@l for History of Probability and Statistics*, 4(2).

[7] Borovicka, T., Jirina Jr, M., Kordik, P., and Jirina, M. (2012). Selecting representative data sets. In *Advances in data mining knowledge discovery and applications*. InTech.

[8] Carreira, J., Agrawal, P., Fragkiadaki, K., and Malik, J. (2015). Human pose estimation with iterative error feedback. *CoRR*, abs/1507.06550.

[9] Chollet, F. (2016). Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357.

[10] Ciresan, D. C., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. *CoRR*, abs/1202.2745.

[11] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[12] Domingos, P. (2012). A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87.

[13] Donahue, J., Jia, Y., Vinyals, O., Hoffman, J., Zhang, N., Tzeng, E., and Darrell, T. (2013). Decaf: A deep convolutional activation feature for generic visual recognition. *CoRR*, abs/1310.1531.

[14] Dong, M., Pang, K., Wu, Y., Xue, J.-H., Hospedales, T., and Ogasawara, T. (2017). *Transferring CNNs to Multi-instance Multi-label Classification on Small Datasets*.

[15] Ghiles (2016). Somewhat noisy linear data fit to both a linear function and to a polynomial of 10 degrees. although the polynomial function is a perfect fit, the linear version can be expected to generalize better. in other words, if the two functions were used to extrapolate beyond the fit data, the linear function would make better predictions. Wikimedia Commons, File: Overfitted Data.png.

[16] Goldberg, Y., Zakai, A., Kushnir, D., and Ritov, Y. (2008). Manifold learning: The price of normalization. *J. Mach. Learn. Res.*, 9:1909–1939.

[17] Grundström, J., Chen, J., Ljungqvist, M., and Åström, K. (2016). *Transferring and compressing convolutional neural networks for face representations*, volume 9730, pages 20–29. Springer Verlag, Germany.

[18] Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier.

[19] Heer, J., Bostock, M., and Ogievetsky, V. (2010). A tour through the visualization zoo. *Queue*, 8(5):20:20–20:30.

[20] Hofmann, M. (2006). Support vector machines – kernels and the kernel trick. pages 10–15.

[21] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

[22] Huh, M., Agrawal, P., and Efros, A. A. (2016). What makes imagenet good for transfer learning? *CoRR*, abs/1608.08614.

[23] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R. B., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093.

[24] Keim, D., Kohlhammer, J., Ellis, G., and Mansmann, F. (2010). *Mastering The Information Age – Solving Problems with Visual Analytics*.

[25] Kiefer, J. and Wolfowitz, J. (1952). Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466.

[26] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report.

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

[28] Källén, H., Molin, J., Heyden, A., Lundström, C., and Åström, K. (2016). *Towards Grading Gleason Score using Generically Trained Deep convolutional Neural Networks*, pages 1163–1167. IEEE–Institute of Electrical and Electronics Engineers Inc.

[29] LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.

[30] Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *CoRR*, abs/1312.4400.

[31] Liu, Z., Luo, P., Wang, X., and Tang, X. (2015). Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*.

[32] Mcculloch, W. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147.

[33] Minsky, M. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA.

[34] Nicoguaro (2016). A scatter plot of samples that are distributed according a multivariate (bivariate) gaussian distribution centered at (1,3) with a standard deviation of 3 in the (0.866, 0.5) direction and of 1 in the orthogonal direction. the directions represent the principal components (pc) associated with the sample. Wikimedia Commons, File: GaussianScatterPCA.svg.

[35] Oquab, M., Bottou, L., Laptev, I., and Sivic, J. (2014). Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, pages 1717–1724, Washington, DC, USA. IEEE Computer Society.

[36] Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318.

[37] Poggio, T. (2012). This image represents the problem of overfitting in machine learning. the red dots represent training set data. the green line represents the true functional relationship, while the red line shows the learned function, which has fallen victim to overfitting. Wikimedia Commons, File: Overfitting_on_Training_Set_Data.pdf.

[38] Puntanen, S., Styan, G. P. H., and Isotalo, J. (2011). *Eigenvalue Decomposition*, pages 357–390. Springer Berlin Heidelberg, Berlin, Heidelberg.

[39] Real, R. and Vargas, J. M. (1996). The probabilistic basis of jaccard's index of similarity. *Systematic Biology*, 45(3):380–385.

[40] Roweis, S. T. and Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326.

[41] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2013). Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229.

[42] Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA.

[43] Shlens, J. (2014). A tutorial on principal component analysis. *CoRR*, abs/1404.1100.

[44] Shlens, J. (2016). Schematic diagram of inception-v3. File: image03.png.

[45] Simonyan, K. and Zisserman, A. (2014). Two-stream convolutional networks for action recognition in videos. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 568–576. Curran Associates, Inc.

[46] Steinwart, I. and Christmann, A. (2008). *Support vector machines*. Springer Science & Business Media.

[47] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. *CoRR*, abs/1409.4842.

[48] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826.

[49] Torralba, A., Fergus, R., and T. Freeman, W. (2008). 80 million tiny images: A large data set for nonparametric object and scene recognition. 30:1958–1970.

[50] Torrey, L., Shavlik, J., Walker, T., and Maclin, R. (2010). *Transfer Learning via Advice Taking*, pages 147–170. Springer Berlin Heidelberg, Berlin, Heidelberg.

[51] Van Der Maaten, L. (2014). Accelerating t-sne using tree-based algorithms. *J. Mach. Learn. Res.*, 15(1):3221–3245.

[52] Van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605.

[53] Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? *CoRR*, abs/1411.1792.

[54] Zeiler, M. D. and Fergus, R. (2013). Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901.

[55] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2017). Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012.

Master's Theses in Mathematical Sciences 2018:E24
ISSN 1404-6342

LUTFMA-3347-2018

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/