Universiteit Leiden
The Netherlands

Opleiding Informatica

Deep and Shallow Features for Image Classification

Gijs Broch

Supervisors:
Michael Emmerich & Hui Wang

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

15/05/2021

Abstract

In this thesis we will look at shallow and deep intermediate features of Convolutional Neural Networks (CNN) and whether they can be used to classify images by training a Support Vector Machine with them. We are also comparing their performance and study how to choose the best layer to extract features from for image classification. We are building our model with public domain libraries and environments in Python, namely Scikit-learn and Caffe. This means we are essentially using the CNN as a data preprocessor for our classifier. A special focus of this thesis is to implement and test methods that can be used on affordable hardware and implemented in open source libraries. Moreover, in contrast to existing studies using only deep features to train Support Vector Machines, we include features of all depth levels in our study.

Contents

1 Introduction

Recent advances in deep learning made tasks such as image classification possible using methods like Convolutional Neural Networks (CNN) [Sch15]. Many methods are being explored to improve this process. One such method is using features other than the final categories. Most papers look at the deep features, extracted from layers typically on the latter half, if not even further, in the network. We will be looking at the performance of both the shallow features and the deep features and comparing them in the context of image classification. We are also showing how to set up a model using existing environments and public domains.
We will discuss how to design a program using public domain libraries. We will use pre-trained convolutional neural networks from Caffe[JSD+14] and extract intermediate features from the layers in this network. For our classifier we turn to Python library sklearn, this has many classifiers to choose from. We will be using a support vector machine but this can be substituted fairly easily. Finally from sklearn we are also using the performance metrics to see how well our program fairs. The simplified workflow of our model is shown in Figure 1. The idea is to put an image dataset through a CNN to extract the features while also generating the labels for our data. We then take the extracted features and generated labels and use them to train and evaluate a separate classifier.

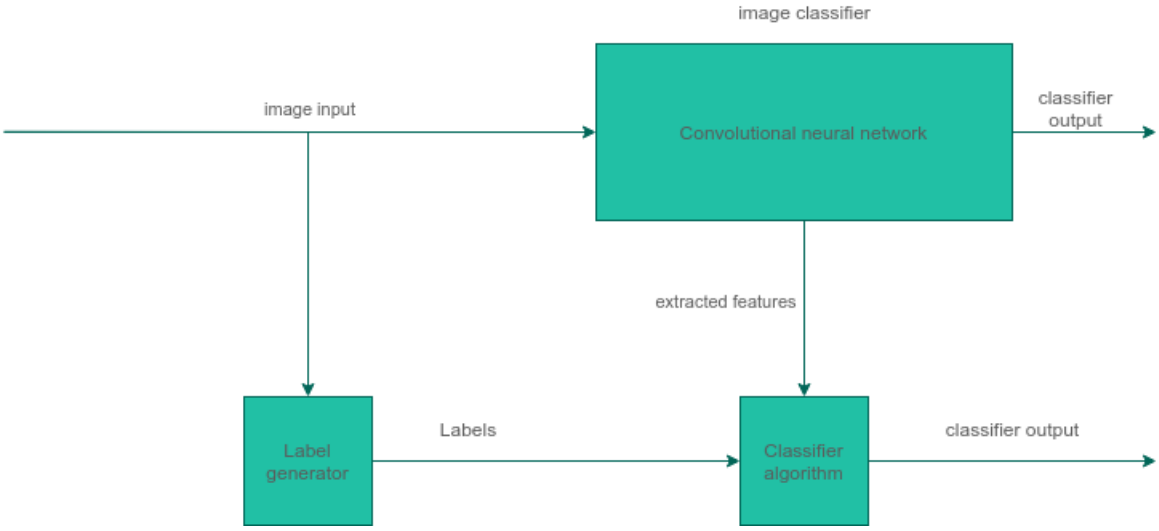

Figure 1: Simple workflow of the model

1.1 Topic and motivation

Image classification methods using deep learning are currently very popular. There are often competitions to make the best models on certain datasets such as ImageNet. The many applications of deep learning make the subject matter very broad, and building and setting up a new model or network can be quite challenging. However there are trained models publicly available with which you can experiment as well. We will be looking at image classification and give a general idea on how to get started with the practical side. Public libraries like Caffe do have tools you can use for things like feature extraction but these seem to be mainly focused on the ImageNet dataset.

Making your own code for extraction allows you more freedom of what to do with the features and what data you put through the convolutional neural network and it opens up possibilities for exploring new methods on new types of data.

1.2 Research questions

To sum up, we will be exploring how to design a program or software that extracts features from a CNN and feeds this to a different classifier, all in public domain, to answer the following questions:

- How do shallow features perform compared to deep features when used in a secondary classifier?
- What layer of the deep neural networks provides the best features for classification?
- How can the principle of deep features be implemented using public domain software components and affordable, low-end hardware?

We will be investigating whether classical machine learning methods can benefit from the auto-generated features and to which extent the depth of the layer influences the quality of the features. The concept will be implemented in a publicly available environment.

2 Definitions

Convolutional neural networks are a family of deep learning methods that use several different layers of neurons. In Caffe these layers are either convolutional layers, pooling layers or fully connected layers. In section 4 we will elaborate on the working of CNNs.
Scikit-learn [PVG+11] is an open source Python library with all kinds of machine learning algorithms as well as functions to calculate the performance of these algorithms. It is built on the Python libraries NumPy, SciPy and matplotlib. We will be referring to it as sklearn as that is what the library is called in Python.
From the Python library scikit-learn we will use the support vector machine(SVM) method SVC as well as the performance metric functions. The performance metric functions are used to calculate the accuracy, recall, and precision of the SVM, these make use of the positive and negative results from the SVM. False positives are results that came out positive but should have been negative, similarly false negatives come out as negative but should be positive. True negative and positive come out as negative and positive respectively and were predicted correctly. Accuracy is the amount of correctly predicted cases, TP and TN divided by the total amount of cases.

P	positive cases
N	negative cases
TP	true positives
FP	false postivies
TN	true negatives
FN	false negatives
sklearn	scikit-learn
CNN	convolutional neural network
SVM	support vector machine

$$Accuracy = (TP + TN)/(P + N) = (TP + TN)/(TP + FP + TN + FN)$$

Recall, or True Positive Rate, is the ratio of true positives divided by the sum of true positives and false negatives.

$$Recall = TP/(TP + FN) = TP/P$$

Precision, or Positive Predictive Value, is the ratio of true positives divided by the sum of true positives and false positives.

$$Precision = TP/(TP + FP)$$

More information and additional model evaluations metrics can be found on the webpage of sklearn [MET].

3 Related Work

In this section we will briefly discuss related work of the more general methods used like the SVM and the CNN, and go somewhat more in depth in the works that are more specific to my thesis such as transfer learning and combining machine learning methods.

3.1 Convolutional neural networks

A convolutional neural network is a class of deep artificial neural networks that is used in deep learning. The simplest method of deep learning is the multilayer perceptron where the layers are often fully connected to each other and are therefore prone to overfitting. Prevention of overfitting is called regularization, and typical ways of regularization include penalizing parameters or removing some of the connections. CNNs take a different approach using the hierarchical patterns in data to assemble patterns of increasing complexity using smaller patterns.
It's rather difficult to attribute the invention of the CNN to one person or paper, it is rather an extension of artificial neural networks many people contributed to. The first artificial neural network that used convolution and that was considered deep was Fukushima's Neocognitron [FM82]. First published in Japanese in 1979, Elsevier published what seems to be a translation that was received in 1981 and published in 1982. CNNs are large and complex, and it would be too much to go into detail on how they work here, Schmidhuber [Sch15] made a historic overview of deep learning in neural networks. Guo et al. [GLO+16] reviewed the state-of-the-art in deep learning algorithms back in 2015, the sections that cover CNNs should give a better understanding on how they work. The simplified general structure of a CNN is depicted in Figure 2 and its building blocks consist of convolutional layers, pooling layers, ReLU layers, loss layers and fully connected layers. The convolutional layers are the starting layers, consisting of a set of learnable kernels creating the feature maps. The pooling layers essentially down-samples the feature maps or input images. Commonly these are put periodically in between convolutional layers or just after the convolutional layers. ReLU layers often come after the pooling layers, the Rectified Linear Unit, and is basically a corrective layer. Several functions can be used for the correction, but the most general one sets negative values in the activation map back to zero. Loss layers specify how the training penalizes differences between predicted output and the actual labels of the input. Several loss functions can be used depending on the task. Finally the fully connected layers, these are basically regular multilayer perceptrons at the end of the network to do the final classification.

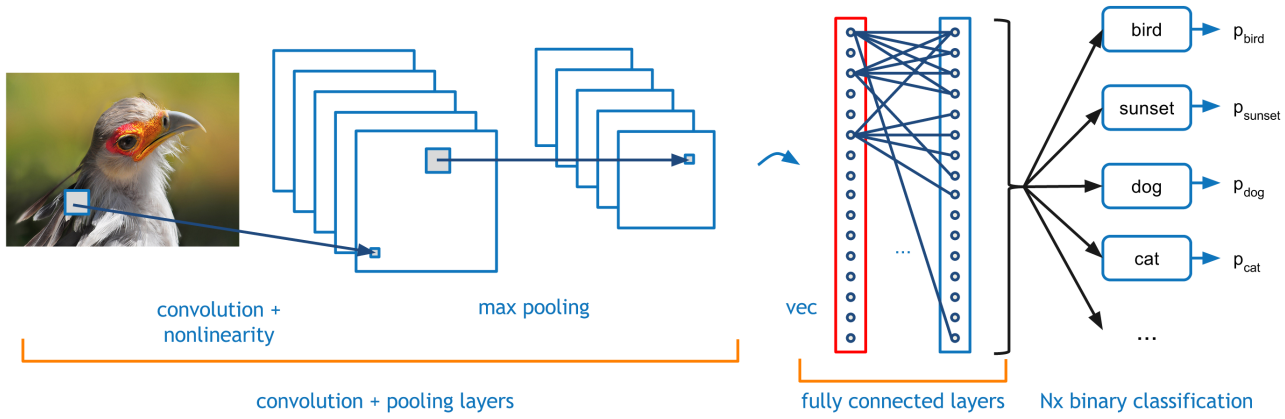

Figure 2: A general structure of a CNN. Source: code.flickr.net.

3.2 Support vector machine

Support vector machines or support vector networks as they were called when first conceived by Vapnik and Cortes [CV95] is a machine learning method developed in 1995. It was originally made for two-group classification problems but more modern ones can handle multiclass problems as well. SVMs use an algorithm wherein the input data is turned into vectors and linearly mapped to a high dimensional space. Then the algorithm, in the case of a two-group classification problem, tries to divide this space in two, separating the two classes with a hyperplane. This hyperplane is a manifold of one dimension less than the space it is placed in, so for a 2-dimensional space this would be a line, for a 3-dimensional space a plane etc. There are several methods with which a good separation is achieved but the most intuitive one tries to maximise the margin or distance between the hyperplane and the nearest data point on each side. This so-called maximum-margin hyperplane is the most intuitive because it leaves the most room for error inside the separation margin. The data points, or vectors that are on the margin line as seen in Figure 3, are referred to as the support vectors. After deciding on the hyperplane, the new input that has to be classified has its class determined by checking on which side of the hyperplane its input vector is placed in the multi-dimensional space.

3.3 Transfer learning

One of the uses of extracting intermediate features from CNNs is transfer learning. In the medical field diagnosis can be difficult, using a deep learning method to predict a patient's affliction using the visual output of a spectrograph or other such equipment can be very useful. But the medical field does not always have millions of data at hand so training a deep learning method like a CNN can be problematic. This is where transfer learning comes into use. A network trained on a large and varied dataset like ImageNet can recognise generic image data, meaning its features in the layers have a general use. These features once extracted can then be used to train a different classifier, like a support vector machine or another (deep)neural network. Nguyen et al. [NLLC18] use a combination of three CNNs to extract features from for transfer learning on a microbiotic dataset in the biomedical field. They also concatenate these features for the transfer learning and train a

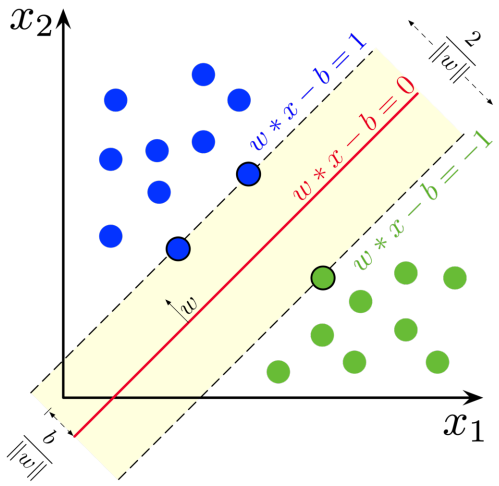

Figure 3: A 2-dimensional SVM with hyperplane and support vectors. Source: en.wikipedia.org.

separate classifier with this. The concatenation of the features provides the separate classifier with more data to train on per image.

3.4 Combining classification methods

With all the data that is being shared in the world nowadays, machine learning is used in more and more fields. So naturally there is a lot of research to further this field and combining machine learning with other methods or other models of machine learning is a common approach. Several approaches are considered in Fernández-Varela et al. [FVHPAEMB17] on automatic detection of EEG arousals. They use two ensemble methods that combine individual classifiers with the best results in their experiments, in their case an ensemble of Classification Trees to make a Random Forest and an ensemble k-Nearest Neighbors(k-NN) to make k-NNE. An ensemble isn't much of a combined approach, it takes the two results of the individual classifier and uses their different errors to correct each other. Their other approaches are a model suggested by Shortlife and Buchanan [SB75] and a linear combination.
They had slightly mixed results but overall saw an improvement in results. Both of the combined approaches had less error and higher specificity and both outperformed the ensemble methods. The ensemble methods however weren't as successful. The Random Forest only had slightly better results than the Classification Tree and the k-NNE obtained higher error than the individual k-NN. The reason for the worse performance in k-NNE was that it couldn't use the same distance function as the individual k-NN the reason being that as an ensemble k-NNE deals with slightly different feature subsets on which the same distance function does not always apply.

More closely related to the experiment in this paper is the research of McAllister et al. [MZBM18] in which they also extract features and use them to train machine learning classifiers. The classifiers they use are artificial neural networks, SVM, Random Forest, Naive Bayes. They use pretrained ResNet-152 and GoogleNet CNNs that were trained on ImageNet to extract features from four image datasets containing pictures of food. ResNet-152 and GoogleNet are much larger CNNs than the ones we're using with 152 and 22 layers respectively. Ours has 8 learnable layers, it might have

more layers in the network but these are the only ones we can extract from. They state that they evaluate the layers for extraction from the end of the pooling layers. Generally speaking this means that the layers considered are in the latter half of the network and thus deep features and not shallow features.
They use Weka, a software platform containing several machine learning algorithms developed at the University of Waikato, New Zealand, we however are using sklearn. Just looking at the accuracy of the classifiers in their results, they all score fairly high on the different datasets. The datasets used are:

- Food-5K, a balanced real world data set of food and non-food items with possible noise.
- Food-11, an unbalanced set of 11 major food groups with noise.
- RawFooT-DB, a balanced data set of 68 classes of food items without noise.
- Food-101, a balanced data set of 101 food categories with 1000 images each that are cleaned of noise
- UNICT-FD889, used to evaluate food/non-food models, contains 889 distinct dishes with high food variance but with little noise
- Caltech-101, a non-food based data set used to evaluate food/non-food models.

For the SVM, using either the RBF or the polynomial kernel, they often achieve over 95% accuracy, the biggest outlier being on the Food-101 dataset. On this multiclass food dataset the SVM with RBF kernel only achieved 68.98% accuracy. On this dataset a lot of misclassification happened between the food groups, however the SVM with RBF was still the most accurate of the methods used in this experiment. This shows that either these models struggle with datasets that contain a lot of different classes or that the images of different classes share a lot of similarities. The other outliers still achieve an accuracy in the 80 to 90 percent range, showing that extracted deep features can be used to recognise food items. What we do differently is that we also evaluate the shallow features and we're using a subset of ImageNet.

3.5 Open questions

Of course these methods are not perfect and there are points of contention to be made. Such as cases where deep learning does not work or the fact that deep learning can be very computationally intensive making it not a viable method for the task.

3.5.1 Adversarial examples

Adversarial examples are misclassification of samples that were designed to fool the classifier. These samples can appear to humans like regular samples, but the algorithm sees them very differently or in some cases the opposite, where humans can see the difference clearly but the machine can not. Sometimes deep learning methods can learn the wrong things, an example of that would be the artificial intelligence project CLIP designed by OpenAI. Results from a paper of Goh et al. [G+21] made the rounds on internet fora recently with jokes that we should not fear a robot uprising, because the AI called CLIP was easily fooled into thinking an apple was an iPod. As seen in in

Figure 4 CLIP was fooled with a piece of paper covering part of the apple with the word iPod written on it. In this case the multimodal neurons that were originally trained to categorize images

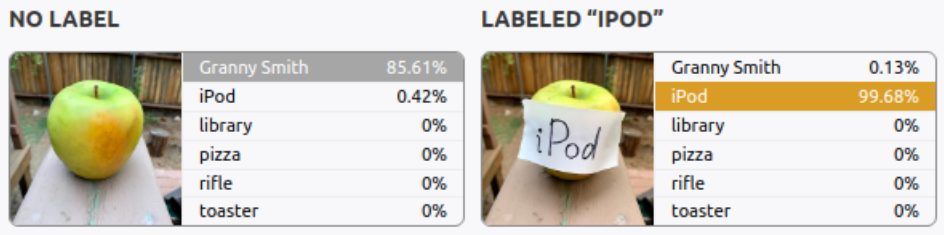

Figure 4: Part of the results from the paper of Goh et al. Source: multimodal-neurons.

had learned how to read text and generally gave the words he saw precedence over the objects he saw. This was also seen when the word pizza had been edited on a picture of a chihuahua several times and therefore saw it as pizza.
A paper by Yuan et al. [YHZL19] also outlines several vulnerabilities during the testing and deploying stage. In the deploying stage this could be dangerous. Self-driving cars for example could be fooled by fake stop signs and suddenly come to a halt in the middle of traffic. For humans it is naturally much easier to point out examples that we see correctly but the machine does not. While not deep learning, one example of where humans have difficulty spotting what's wrong and a machine does not is a spam filter on your email. Hand crafted filters can be fairly easily fooled by masking the email or other methods. In the case of phishing emails for example, if it is made well enough it can relatively easily fool the recipient, but because the mail does not fit the criteria of the filter it is not marked as spam.

3.5.2 Computationally intensive

A big reason to not use deep learning in a lot of cases is that it is computationally intensive and according to Thompson et al. [TGLM20] the models are scaling more rapidly than known lower bounds from theory. This suggests that improvements might be possible but also that if things continue to progress this way, it will rapidly become economically and technically prohibitive. Not even mentioning environmentally with the power costs. For example AlexNet was trained in 2012 for five to six days on two GPUs, in 2017 it took ten days to train ResNeXt-101 on eight GPUs.

4 Methodology

In this section we discuss our methods, hardware and data. As previously stated we want to use publicly available libraries so that anyone can use it. This extends to our data and hardware as well. Image datasets can relatively easily be made by yourself with a script and Google image search, so we are slightly bending the rules there a little for sake of convenience. Hardware is also an interesting point with machine learning. Naturally high-end machines have better performance but not everyone has access to those so we want to make sure that our software works on lower end machines as well.

4.1 Dataset

We use a subset of the ImageNet [RDS+15] 2012 dataset, using the brain coral and sea anemone subsets. Both sets have 1300 training images and 50 test images, totalling 2600 training images and 100 test images. These 2 subsets were chosen for their size, being among the larger subsets of ImageNet. They are also both of aquatic life so the datasets are not so dissimilar that the classifier would have an easy time on recognising the difference. We are however not using the wordnet that comes with ImageNet, we are only using it to locate the corresponding images in the validation set to use as our test images. The images from the coral and anemones will be saved to their own folder. When we start loading we also generate a vertical array of ones for the coral and zeros for anemones. We then concatenate these two vertically to create our label array. Because we first loaded one set and then the other we know that these entries line up properly when fed to our CNN and SVM.
Now ImageNet is not completely public domain, to get a copy of the dataset you need to request it on their website with a university email that they recognise. If you do not have such an email address but you do attend a university that uses ImageNet in their research you can also approach one of the research groups there to receive a copy. There are other datasets available like CIFAR but those are saved into a database format, so you have to first find a way to extract the images. You would also need to save them if you only want to use a subset. You can also make your own dataset by writing a script to download images from a google search query. As long as you save the different group of pictures to a different, preferably empty, directory it is easy to assign them labels once you start loading them into your program.

4.2 Caffe

Caffe [JSD+14], developed by Berkeley AI Research, is a deep learning framework made with expression, speed and modularity in mind. Many community contributors helped develop it and many pretrained models are available for use in their so-called Model Zoo. Along with TensorFlow, Caffe is one of the more widely used open source deep learning platforms. TensorFlow can sometimes have large changes between updates while Caffe stays relatively the same. This makes Caffe a more attractive option for people new to deep learning. That being said, while Caffe is still useful as a learning tool to gain an understanding of deep learning, it is no longer being developed. Caffe2 has already been developed, its libraries have now also been deprecated and newer ones have now been implemented in PyTorch.

4.2.1 Caffe modes

Caffe can be run in two modes, CPU-only and CPU with GPU support. The GPU supported mode has better performance but also requires a GPU compatible with CUDA, a parallel computing platform developed by NVIDIA. There are a couple of downsides to this. First of all because CUDA is developed by NVIDIA, GPUs developed by AMD do not support it, this means you are limited to graphic cards developed by NVIDIA. Second, not all of NVIDIA GPUs, especially older ones, support CUDA either.
CPU-only mode has worse performance but its independence on the GPU means that it could be run on any system. This is especially useful if you can run it on a high end machine like a server. You can develop it on your PC and just transfer it to the server to run it. We are using

CPU-only mode, primary reason being that our hardware does not support CUDA. (more on that in section 4.4) But it is also easier to compare performance when only using the CPU. There are less components to take into consideration, not just the GPU but also the data transfer between CPU and GPU.

4.2.2 Model of choice

There are plenty of models to choose from the model zoo of Caffe. We settled one of the models from the following paper by Chatfield et al. [CSVZ14] for several reasons. First of all we want something that is trained on ImageNet. This might create a bias because we are also using a subset of ImageNet but since we are not using the training data for validation it should be fine. Using the same data that the model was trained with should still give us good features to train our classifier with.
We also want the model to be accurate, several of Chatfield's networks score a top-5 error rate of around 14%. Top-5 error rate is one of the benchmarks used during the ImageNet competitions, 14% is a reasonably high score especially for an eight layer CNN. We take the Slow model (CNN-S) which was made with accuracy over speed in mind. The fine-tuned models are not available but we do not necessarily want the fine-tuned models. We might run the risk of overfitting with the possible bias on the dataset combined with a fine-tuned CNN. We are then putting the features extracted from the CNN through another classifier, a more classical method of machine learning, like an SVM in our case. Ideally this classifier will be fine-tuned but in our case we are only checking the first results with a largely default SVM. Because this paper has several models that are trained somewhat similarly we could possibly swap out for a different model whose primary difference is being built for speed over accuracy.

4.3 Scikit-learn

Scikit-learn [PVG+11], otherwise known as sklearn, is a freely available Python library with many machine learning components. We are using the SVC, Support Vector Classifier, component for our proof of concept, but it can easily be replaced by another method, for instance a RandomForest. SVMs are widely used for classification problems and the one in sklearn is easy to implement. We are using the default settings to get a general idea of the performance of the extracted features but with the linear kernel from SVC because it is faster than the default rbf kernel. Naturally these settings will not give us the best performance as the classifier has not been optimised for our problem. Fine-tuning of the parameters would yield better results. Sklearn has three different SVMs available that are largely similar, but could be considered when fine-tuning the model. SVC is the most basic one, which is why we are using it. LinearSVC has a faster implementation of the linear kernel, as such it does not have the parameter for kernel that SVC has. NuSVC is similar to SVC but uses a different formulation and therefore has a slightly different set of parameters. We will not discuss the differences in this thesis but the details can be found on the website of sklearn [SKL] All three of the different SVMs can be expanded to become multiclass SVMs but we are sticking with a binary classification in our thesis.

As mentioned in section 1, we are also using sklearn's performance metric functions that are designed to be used with all the different machine learning methods available in sklearn. We are

looking at accuracy, precision and recall for our performance metrics. Recall from section 2 that accuracy is a general statistic to measure the percentage how many cases were correctly classified. Precision to measure how often a positive prediction was actually true. As a result it also measures how many false positives there were when predicting the test set. And lastly the recall is used to check the true positive rate. Measuring how many of the positive cases in the test set were actually predicted positive and more importantly, measuring how many false negatives were encountered. Both false positives and false negatives can be harmful, and depending on the application, one is more harmful than the other. For example an anti-virus on a computer can identify executables as harmful even though they came from a trustworthy source or were compiled by the user. These false positives are indesirable but what would be worse is if an actual malicious program was seen as not harmful. This false negative would be more harmful than the false positive.
A program that predicts events in the stock market or other form of gambling is the opposite. A false negative would lead you to not put in the money thus losing out on some profit. But a false positive would lead you to put in the money and not only would you not make profit, you would lose the money you invested.

4.4 Hardware

Deep learning methods benefit greatly from high end hardware with all the calculations it has to do and data it has to parse and potentially save. When doing experiments with extracted features, we would want to save those features so that we do not have to run to the CNN each time we want to test something. Since we are making use of public domain our methods for feature extraction should also work on the average computer setup or maybe even low end hardware. That being said, we are using a roughly 8 year old laptop with 8GB of RAM and an Intel Core i5-4210H CPU, 2.90GHz quad core running Ubuntu 16.04. As previously stated in 4.2.1 you can use the GPU to boost the performance but we are only using the CPU in this thesis. While the laptop used in this thesis has an NVIDIA GPU it does not support the GPU programming environment.

5 Experiments

In this section we go over the setup of our experiment, we present our results and challenges that came up during the execution of the experiment.

5.1 The model

We are using a pretrained model from this paper [CSVZ14] for our experiment and extract the features from all the layers to see which performs best. The extraction from the 4th and 7th layer can be seen in Figure 5. Other than flattening the data from the convolutional layers we do not process these features. When reading in the images we reshape them all to the same size, this is to make sure they all fit through the CNN and also ensure that the output is uniform in shape. If the output is not uniform then the SVM will not accept it. We then train an SVM on this data and evaluate it with a smaller set of images of which we also extracted the features. As previously mentioned our experiment uses a subset of the ImageNet 2012 dataset, using the brain coral and sea anemone subsets. Both sets have 1300 training images and 50 test images, totalling 2600 training

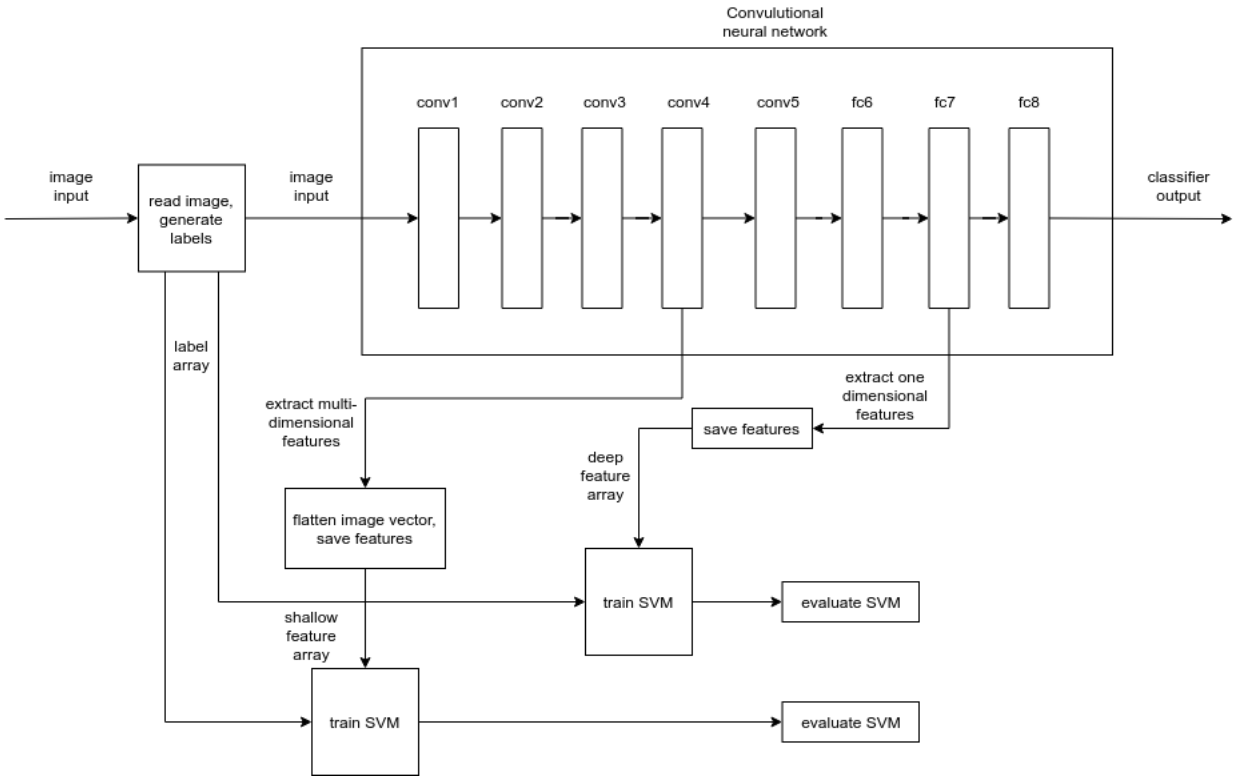

Figure 5: Detailed workflow of the model

images and 100 test images. These 2 subsets were chosen for their size, being among the larger subsets of ImageNet.

5.2 Results

The first test was done using the channelwise image mean of ImageNet provided in the Caffe libraries. A channelwise image mean is a method to preprocess the training data to give the network some additional information to work with, it consists of a vector with the average 3 colour values, RGB, of every pixel of every image in the dataset. In the early stages of this research it was unclear whether this image mean was only needed during training of the model or during all stages. Many tutorials online that use the pretrained models to extract features just use the one supplied by Caffe or make no mention of it. So the first tests in this thesis were done with this image mean. When it was clear that a mean had to be generated on our own dataset we decided that we can compare the results to show and understand what the differences would be between using the wrong and the correct image mean.

5.2.1 First experiment

	conv3	conv4	conv5	fc6	fc7	fc8
Accuracy	0.85	0.78	0.77	0.66	0.72	0.71
Precision	0.8182	0.7593	0.7755	0.6600	0.7037	0.6780
Recall	0.90	0.82	0.76	0.66	0.76	0.8

Table 1: First results with the default SVM with linear kernel from sklearn using the image mean provided by Caffe

As previously stated this experiment used the wrong image mean, the one supplied with Caffe is an image mean over the entirety of ImageNet whereas we only use a small subset of ImageNet. As seen in Table 1 on average the convolutional layers seem to perform better than the fully connected layer with the third layer performing the best. It should be noted however that the data in the convolutional layers are much larger. The single vector extracted from the fully connected layer 6, "fc6", has 4096 values stored in it, making the feature matrix of all the images a size of 2600×4096. The 2-dimensional vectors that have been flattened such as the third layer, "conv3", however has in a single flattened vector 147968 values. Thus the matrix has a size of 2600×147968. Layers 1 and 2 have even larger output making them use even more memory, so much in fact that our 8GB of RAM is not enough to save the features from both layer 1 and 2. We can however extract the features from the test set as it is much smaller. We then see that layer 1 and layer 2 would have had a feature of size 2600×1140576 and 2600×278784 respectively. Layer 8 is also an outlier, While fc6 and fc7 are the same size, fc8 is only 2600×1000 making it the smallest.
Table 2 gives a rough estimation of the sizes of the features as well as the load and execution times. The features of the test set of layer 3 saved to a numpy array are 118,4MB, the features for layer 2 are 223,0MB and for layer 1 it is 912,5MB. Layer 2 is almost double that of layer 3 and layer 1 is almost 8 times the size of layer 3. We can get a better estimation using the test data we could extract however. The size of the of the test is 100 images and the size of the training set is 2600

	conv1	conv2	conv3	conv4	conv5	fc6	fc7	fc8
Size test set(MB)	912.5	223.0	118.4	118.4	118.4	3.3	3.3	0.80
Size training set(MB)	±23724.0	±5798.7	3077.7	3077.7	3077.7	85.2	85.2	20.8
Load time(ms)	±216840	±53001	28566	28634	28410	786	774	231
training time(ms)	-	-	957863	941083	926830	17583	20815	8463
prediction time(ms)	-	-	27866	27172	26060	531	560	139

Table 2: Memory and time statistics for the first results

images. We can easily estimate the size of the training feature array of layers 1 and 2 by multiplying their test feature arrays by 26. This gives us roughly 5.8GB for layer 2 and 23.7GB for layer 1. In terms of memory this is much too inefficient so we can just ignore layers 1 and 2.

The runtime of the software has been separated into three categories to gain a better understanding of the time involved in each step of the process. The loading time to measure how long it takes to load in the training set, training labels, test set and test labels. Training time refers to how long it takes to fully train the SVM and finally prediction time is how long the prediction takes with the hundred images from the test set. We can see that the fully connected layers have much shorter load, execution and prediction times with the final layer coming out ahead with a runtime totalling only 8833 millisecond or roughly 9 seconds. Layers 6 and 7 hover around 20 seconds, and layers 3 through 5 hover around 16 minutes.
We can estimate the loading time by calculating the average loading speed in MB/ms with the simple formula:

$$A = \frac{1}{n} \times \sum_{i=1}^{n} \frac{M_i}{T_i}$$

Where A is the average loading speed, n is the number of terms, M_i is the value of the memory in MB(test set and training set) of the term and T_i is the loading time of the term in ms. So we get:

$$\frac{1}{6} \times (\frac{3196.1}{28566} + \frac{3196.1}{28634} + \frac{3196.1}{28410} + \frac{88.5}{786} + \frac{88.5}{774} + \frac{21.6}{231} = 0.1094076551$$

Now we simply have to divide the total memory of the training and test sets of layers 1 and 2 with the loading speed to get an estimate of their loading times.

$$T = \frac{M}{A} \text{ layer 1:} \frac{23724.0}{0.1094076551} = 216840 \text{ layer 2:} \frac{5798.7}{0.1094076551} = 53001\text{ms}$$

With A being the average loading speed in MB/ms we just calculated, T the loading time in ms and M the combined memory of the test and training features in MB.
For the execution and prediction time there is too much variance to calculate them. There is a lot of variance with the costs of calculations, some cost more time than others and with more data there is also more time needed to move that data around. Loading times are not that important though outside of an experimenting environment. In practice, once you finished your model, you would have a pipeline from CNN straight to the SVM without first storing the features on a disk or drive. The training and prediction time are more important to measure performance and we can see that the data taken from the convolutional layers take considerably longer than the data from the fully connected layers.

5.2.2 Second experiment

	conv3	conv4	conv5	fc6	fc7	fc8
Accuracy	0.84	0.77	0.77	0.67	0.72	0.72
Precision	0.8148	0.7455	0.7755	0.6667	0.7037	0.6897
Recall	0.88	0.82	0.76	0.68	0.76	0.8

Table 3: First results with the default SVM with linear kernel from sklearn using the image mean generated on our dataset.

Table 3 shows the results when we are using the correct image mean and while the changes are minor, they show an interesting pattern. Compared to first results in Table 1, here we see a slight decrease in the performance in the earlier layers and a slight increase in the later layers, with layer conv5 being the exact same in performance. In our case this suggests that the error in using the wrong mean becomes more apparent in the later layers. Figures 6, 7 and 8 show the differences in

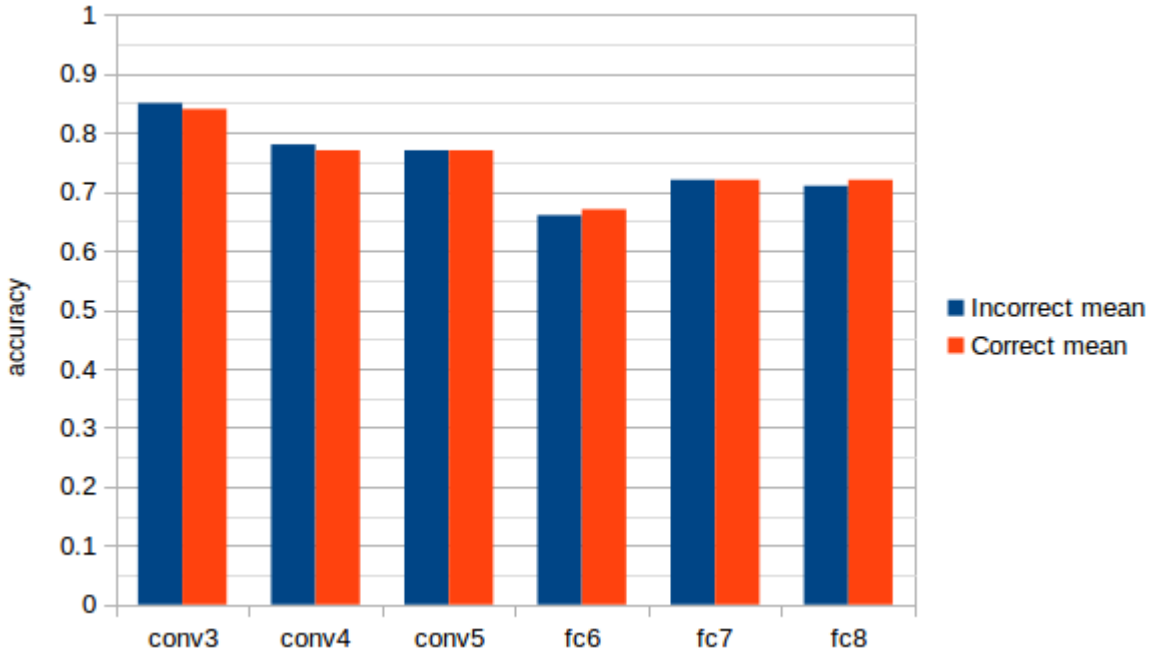

Figure 6: Comparison of the achieved accuracy

performance. We get relatively similar performance however, possibly because the image mean we generated is generated on a subset of the data that the other mean was generated on. Regardless of whether or not the performance would have been higher with the ImageNet mean, it is better practice to use a mean generated on your own data than it is to use an ad hoc mean. The ad hoc mean is not relevant to the dataset being used, whereas a mean generated on the dataset is. Table 4 has the loading, training and prediction times for the second run using the correct mean. The sizes for the files containing the features were the same so predictably the performance times would also

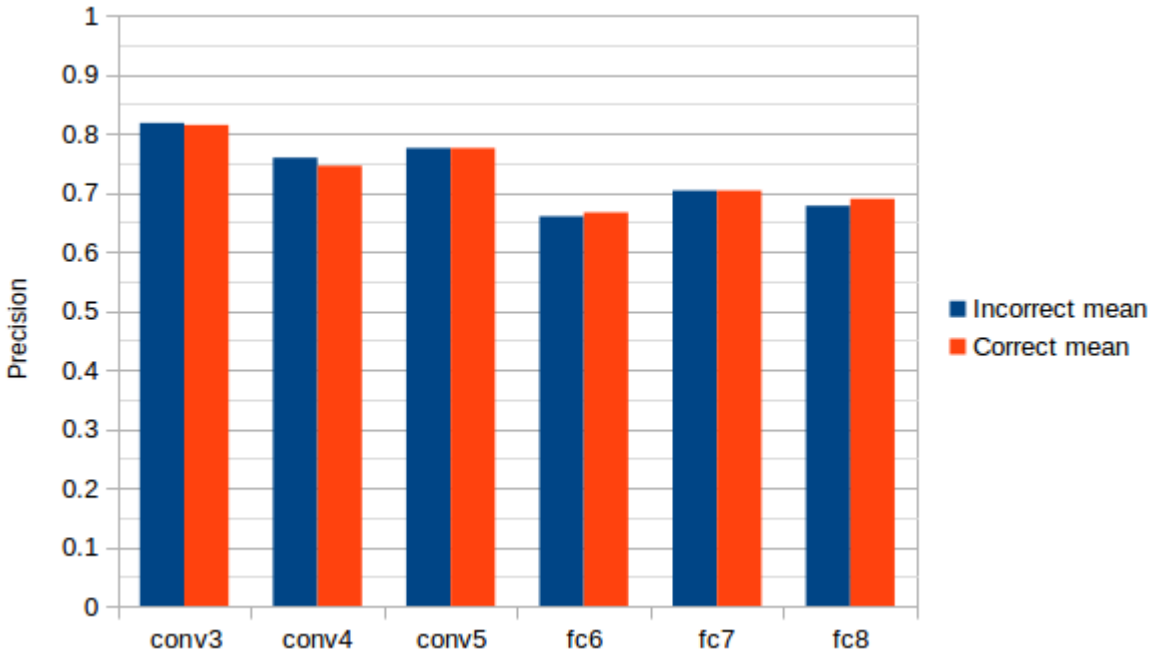

Figure 7: Comparison of the achieved precision

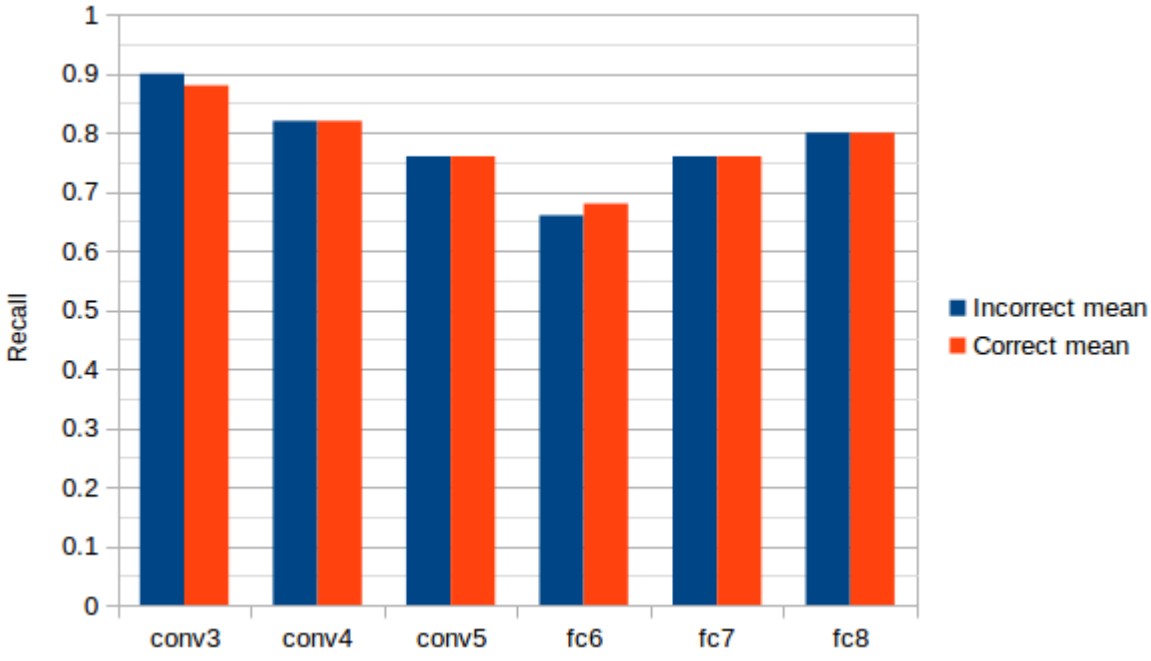

Figure 8: Comparison of the achieved recall

	conv3	conv4	conv5	fc6	fc7	fc8
Load time(ms)	11776	30960	27153	824	888	224
Training time(ms)	954125	933979	926724	17984	20784	8848
Prediction time(ms)	29410	27896	27532	527	557	139

Table 4: Memory and time statistics for the first results

be similar. The one outlier being the loading time for conv3, but this was because that had recently been accessed in a previous test run.

5.2.3 Discussion

When visualizing an SVM you can often see a clear divide between the data, this can be seen in the example in Figure 3. But when dealing with data that has a high dimensionality, visualizing in itself is challenging. We can not visualize anything higher than three dimensions, but we can use techniques to reduce the dimensionality. Principal component analysis, or PCA [JC16], is a technique for reducing dimensionality and minimizing information loss. It does so by creating new uncorrelated variables that successively maximize variance. This however still has its limits as seen in Figure 9. As you can see the data points are all clustered together, making it look like the decision boundary was decided arbitrarily. Even when reducing the dimensionality of our data, it is not linearly separable in some low dimensional space.

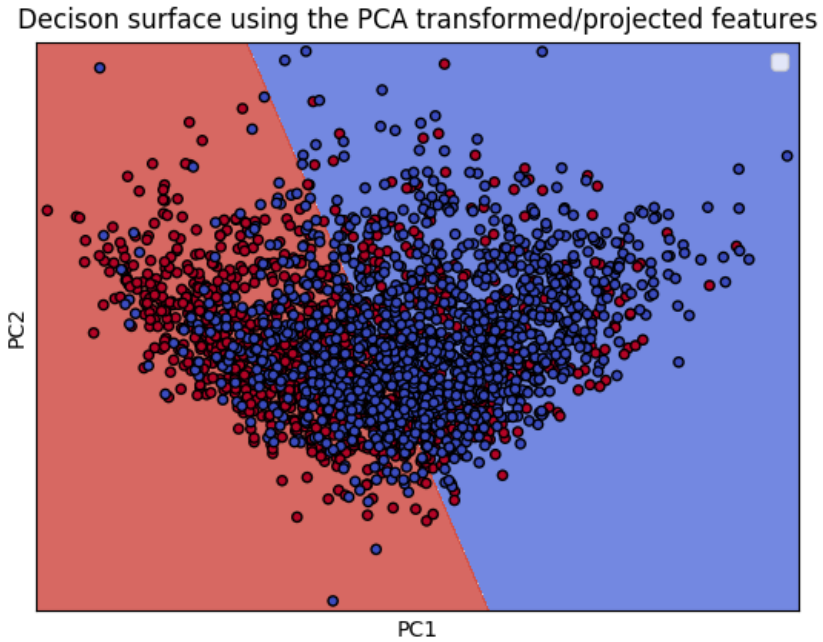

Figure 9: A visual of a PCA on the features of layer fc8

Because we are running on old hardware, the computation speed is naturally lower than if you were to run this on a current generation budget PC. We are also using desktop specs and not regular desktop PC specs making it even slower. Although we did not measure it accurately, the extraction of the features takes a considerable amount of time for 2600 images. The performance is somewhat better on the fully connected layers as these take up less memory, but it took roughly two hours to extract the features from convolutional layer 5 for example. With enough RAM you can extract several features at the same time, and it should be said that this process can be made more memory and possibly more time efficient by using a more efficient data structure. Like using some form of database instead of a numpy array. Using a database would then also decrease the load times during the SVM process.

As we can see, layer 3, conv3, has the best performance on the data, even after the decrease from using the correct mean. But the shallow layers also use a lot more memory and have a longer execution time. When doing further training of the SVM with fine-tuning and possibly larger dataset, the memory inefficiency will prove to be the largest issue. The time for training and prediction will be less of an issue, in both future training and application. Training could be done on a server or overnight on a PC as for prediction, there are not a lot of scenarios where you want to predict a lot of images at once. And even in that scenario, roughly half a minute for a hundred images is perfectly acceptable in most applications.
Among the more memory efficient fully connected layers, layers 7 and 8 have the best performance, with fc7 having a better accuracy and precision, but a worse recall. The execution time for fc7 is also higher than of fc8. For large datasets the fully connected layers are more appealing, with 3.1GB of data per 2600 images on the convolutional layers you would already have 12.4GB with roughly ten thousand images. At this point layers 1 and 2 should not even be considered anymore, they use way too much memory. The performance increase just might not be worth it with that memory usage. Not to mention that this is a default SVM, and fine-tuning the fully connected layers might outperform the convolutional layers.

6 Conclusions

From the first results of our experiments it can be concluded that the shallow features have better performance when put through the SVM compared to the deep features. That being said, this comes at the cost of using a lot more memory and it would be more efficient to use the deep features on larger datasets simply because of that. In Caffe there does not seem to be an option to cut the network short after a certain layer so you can not decrease computation time either by using the shallow features. In fact because you are handling larger pieces of data, this actually takes more time than the deep features. Using more memory efficient data structures might alleviate this a little but it does not completely remove this difference. So ultimately the deep features are more efficient and usable.
As for which layer has the best features to use for image classification, that is still debatable. If the memory overhead is not an issue or if you have a small dataset, conv3 gives the most accurate results without finetuning the classifier. If that is an issue however, layers fc7 and fc8 are both good candidates. There is a small difference between these two performance wise and fc8 has smaller feature vectors than fc7.

A tutorial for using our program can be found in appendix A and the code can be found on github.

7 Further research

As we were only looking at first results, there is a lot that can be improved and extended on here. As stated earlier, we can use a more memory efficient data structure like a database to save our features to cut down on load and possibly execution times. We are using a subset of ImageNet on a network trained on ImageNet, this might cause overfitting, so using a different dataset would be worth taking a look at. We have only tested one network built in Caffe, we can check the performance on features extracted from other networks to see if there are any notable differences. You can even combine the features of different networks to train a classifier. Or combine the shallow and deep features in some way and put that through a classifier. Fine-tuning the SVM, there is no such thing as a free lunch in deep learning. The default settings on a learning algorithm are practically never the optimal settings for the problem at hand. Trying other classifiers and comparing their performance to the SVM, there might be classifiers better suited for this task. Currently we are just reshaping the images, we can also try cropping part of the image for partial image recognition. Chatfield et al.[CSVZ14] had, as previously mentioned, several other but similar CNNs in his paper. We used the more accurate of the models available to us but it would be interesting to compare with the features extracted from CNNs that were more geared toward speed. Sklearn SVMs do not seem to be multithreaded, but they can be when using an additional library. The model could be improved with regards to execution time.

References

[CSVZ14] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the Devil in the Details: Delving Deep into Convolutional Nets. In *British Machine Vision Conference*, 2014.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, Sep 1995.

[FM82] Kunihiko Fukushima and Sei Miyake. Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, 15(6):455–469, 1982.

[FVHPAEMB17] Isaac Fernández-Varela, Elena Hernández-Pereira, Diego Álvarez Estévez, and Vicente Moret-Bonillo. Combining machine learning models for the automatic detection of EEG arousals. *Neurocomputing*, 268:100–108, 2017. Advances in artificial neural networks, machine learning and computational intelligence.

[GLO+16] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S. Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48, 2016. Recent Developments on Deep Big Vision.

[G+21] Gabriel Goh, Nick Cammarata , Chelsea Voss , Shan Carter, Michael Petrov, Ludwig Schubert, Alec Radford, and Chris Olah. Multimodal Neurons in Artificial Neural Networks. *Distill*, 2021. https://distill.pub/2021/multimodal-neurons.

[JC16] Ian T. Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202, 2016.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[MET] 3.3. Metrics and scoring: quantifying the quality of predictions scikit-learn 0.24.2 documentation. https://scikit-learn.org/stable/modules/model_evaluation.html. (Accessed on 06/22/2021).

[MZBM18] Patrick McAllister, Huiru Zheng, Raymond Bond, and Anne Moorhead. Combining deep residual neural network features with supervised machine learning algorithms to classify diverse food image datasets. *Computers in Biology and Medicine*, 95:217–233, 2018.

[NLLC18] Long Nguyen, Dongyun Lin, Zhiping Lin, and Jiuwen Cao. Deep CNNs for microscopic image classification by exploiting transfer learning and feature concatenation. pages 1–5, 05 2018.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[SB75] Edward H. Shortliffe and Bruce G. Buchanan. A model of inexact reasoning in medicine. *Mathematical Biosciences*, 23(3):351–379, 1975.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[SKL] 1.4. support vector machines scikit-learn 0.24.2 documentation. https://scikit-learn.org/stable/modules/svm.html. (Accessed on 06/22/2021).

[TGLM20] Neil C. Thompson, Kristjan H. Greenewald, Keeheon Lee, and Gabriel F. Manso. The Computational Limits of Deep Learning. *CoRR*, abs/2007.05558, 2020.

[YHZL19] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial Examples: Attacks and Defenses for Deep Learning. *IEEE Transactions on Neural Networks and Learning Systems*, 30(9):2805–2824, 2019.

A Instructions

Our instructions assume you have some experience writing code and know some basics of Python. The bare minimum you need to do is change some paths in the code to get the program to work. For anything more involved like fine-tuning the SVM we assume you have enough knowledge of Python to change anything in the code to suit your goals.

A.1 Installing Caffe

First you need to acquire the Caffe library. Our code was made for Ubuntu so that `installation` is what we are following.
For Ubuntu 17.04 and higher there is an apt-get command you can follow for the installation of Caffe and another for installing the dependencies. I tried this once on a virtual machine and couldn't run my code, so your results may vary.
I recommend sticking with the manual download of Caffe from their github and following the instructions for Ubuntu 16.04 and lower. Be mindful that not all the steps are in the white boxes on their website. A few of the steps are in the text surrounding the boxes, these can be easily overlooked. A link to their github with the library files is on the left side of their installation page.

A.2 Running the program

Once you have successfully installed Caffe, our programs can be found on `github` The programs need to be run from a terminal or command line, navigate to the directory where the program is saved execute these commands:

- `python3 extraction.py`
- `python3 svm.py`

The file `extraction.py` is for running the feature extraction program, in its current configuration it will save the features and labels in a subdirectory. The file `svm.py` is used to train an SVM on the features. There is also the file `svm_with_PCA_graph.py` if you want to look at a visualisation of the data, but since this alters the data it does not compute the metrics. The code contains comments that explain some of the components and tell you which paths that you need to edit to satisfy your paths/folder structure. The code does not support multiclass classifiers. To make the code support multiclass, you need to change how the labels are generated to make sure there are more than 2 different labels. You also need to write your own code for the classifier or alter the existing SVM one so that it supports multiclass.
With the current setup you want to you folder structure to be something like:
working directory
|-dataset
|—training set
|——images of class 1
|——images of class 2
|—evaluation set
|——images of class 1
|——images of class 2