CRANFIELD UNIVERSITY

ANDREAS SCHMIDHOFER

AUTOMATED FOOD LOG ANALYSIS

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Computational and Software Techniques in Engineering

MSc
Academic Year: 2016–2017

Supervisor: Dr Stefan Rüger
August 2017

CRANFIELD UNIVERSITY

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Computational and Software Techniques in Engineering

MSc

Academic Year: 2016–2017

ANDREAS SCHMIDHOFER

Automated Food Log Analysis

Supervisor: Dr Stefan Rüger
August 2017

This thesis is submitted in partial fulfilment of the requirements for the degree of MSc.

Abstract

Food logging can be beneficial for reaching weight loss goals. In oder to support an automated food log proper food image classification algorithms have to be developed. I propose one such algorithm that uses a pre-trained deep neuronal network called Inception as a feature generator. The focus of my work was the support of an adaptive learning rate. Using this algorithm I was able to achieve 89.4% accuracy on the Food-11 dataset.

Keywords

Food log; image classification; deep learning; gradient descent; adjusting learning-rate; deep convolutional neuronal network.

Contents

Abstract **v**

Table of Contents **vii**

List of Figures **ix**

List of Tables **xi**

List of Abbreviations **xiii**

Acknowledgements **xv**

1 Introduction **1**

2 Motivation **3**
2.1 Problem Definiton 3
2.2 Challenges 4

3 Literature Review **9**
3.1 Food Image Classification 9
3.2 Convolutional Neural Networks 10
3.3 Food Image Classification using DCNNs 12
3.4 Adaptive Learning Rate 13

4 Setup **17**
4.1 Framework 17
4.2 Environment 18
4.3 Dataset 18
4.4 Network Architecture 23
4.5 Mobile Computing 25

5 Methods **27**
5.1 Retraining 27
5.2 Feature-map Caching 29

5.3 Training Procedure 29
5.4 Stopping Criterion 30
5.5 Adaptive Learning Rate 32
5.6 Initial Learning Rate 41

6 Results **43**
6.1 Confusions 43
6.2 Training Times 48
6.3 ALRGD compared to SDG 48
6.4 ALRGD compared to AdaDelta 49
6.5 ALRGD on other Datasets 53

7 Future Work **57**

8 Conclusion **59**

References **61**

A Additional Figures **69**

B Python Code **75**

List of Figures

2.1 Examples for high intra-class variability 7
2.2 Examples for low inter-class variability 8

3.1 The structure of a CNN 11

4.1 Examples of the Food-11 dataset 21
4.2 The architecture of an Inception V3 network 24

5.1 The last layer of the DCNN 28
5.2 An example of overfitting 31
5.3 A comparison of learning rates 32
5.4 Adaptive learning rate version 1 34
5.5 Adaptive learning rate version 2 35
5.6 Adaptive learning rate version using cross entropy 39
5.7 Adaptive learning rate version using variable increase 40
5.8 Example of a high initial learning rate 42

6.1 Re-training using ALRGD 44
6.2 Examples of misclassified images 47
6.3 ALRGD vs SGD 50
6.4 ALRGD vs AdaDelta 51
6.5 ALRGD vs AdaDelta with low learning rates 52
6.6 ALRGD vs AdaDelta with high learning rates 54

A.1 ALRGD on UNICT-FD1200 dataset 70
A.2 ALRGD on UECFOOD-100 dataset 71
A.3 ALRGD on UECFOOD-256 dataset 72
A.4 ALRGD on ETH FOOD-101 dataset 73

List of Tables

4.1 A list of deep learning frameworks 17
4.2 A list of food image datasets 19
4.3 Number of Images in the Food-11 dataset. 20

6.1 The confusion matrix 45
6.2 A comparison of execution times 48
6.3 Comparison on Food-11 and UNICT-FD1200 55
6.4 Comparison on UECFOOD-100, UECFOOD-256 and ETH Food-101 . . 55

List of Abbreviations

ALRGD	Adaptive Learning Rate on Gradient Descent
BOF	Bag-of-features
CPU	Central Processing Unit
CNN	Convolutional Neural Network
DCNN	Deep convolutional Neural Network
EPFL	Ecole Polytechnique Fédérale de Lausanne
HPC	High Performance Computing
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
OS	Operating System
PFID	Pittsburgh Fast-Food Image Dataset
RAM	Random Access Memory
RFDC	Random Forest Discriminant Components
RMS	Root Mean Square
SDG	Stochastic Gradient Descent
SIFT	Scale-Invariant Feature Transform
SSD	Solid-State Drive
SVM	Support Vector Machine
UEC	University of Electro-Communications
US	United States (of America)

Acknowledgements

First and foremost I would like to thank my supervisor, Dr. Stefan Rüger, for his continuous support throughout the project. He always took time for the meetings to discuss the subject matter in depth. This provided me with helpful insights and kept me motivated to continue giving my best.

I am grateful for the involvement of Dr. Horst Bischof, my co-supervisor from Graz University of Technology, for his constructive comments and for ensuring that my project provides novelty to the field.

I also want to thank Nogaret Baptiste, my predecessor, who worked on this project last year and gave me valuable tips to get started.

Furthermore I would like to acknowledge the contribution of the Multimedia Signal Processing Group at EPFL since they provided the dataset that has been used throughout this project.

Finally I want to express my gratitude towards my parents, Annemarie and Wolfgang Schmidhofer, who supported each of my decisions and provided me with the financial support that made studying abroad possible.

Chapter 1

Introduction

Obesity is becoming a considerable problem to our society. With over a billion people affected it has been labeled the sixth most important risk factor contributing to the overall burden of disease worldwide [17]. Especially in the western world the obesity rates were going up in recent years. According to [4] we saw an increase of obesity in Europe over the past 10 years with now around one in five people of the population being obese. It is even worse in the US where one out of three people is obese as reported in [25].

Studies like [3] and those reviewed in [6] have found a correlation between self monitoring one's food intake and successful weight loss. This might be due to the fact that logging whatever you eat makes you conscious about what it is you are eating and thus you might be more likely to avoid the unhealthy choice. Another possible reason could be that monitoring requires effort and thus one might not want to go though this effort just for a quick snack and therefore eat less.

Smartphones are becoming more and more common. Nowadays almost everybody owns a smartphone. That creates the opportunity of mobile applications to reach a lot of people.

One such application could help with monitoring food intake. If the user only has to

take a picture of the food instead of noting down the time of consumption, the type and amount of food etc. it would be less of an inconvenience. This would mean that the initial resistance of starting self-monitoring would be lower and therefore more people might start self-monitoring.

This application has to be able to perform several tasks. It has to do allow the user to easily take pictures of food, it has to perform food localisation, food classification, estimate amount and composition and provide functionality for logging and analysis.

Deep learning algorithms have been successfully applied to many fields such as speech recognition [20], face recognition [39], bioinformatics [9], image classification [26] etc. The major disadvantage of deep neuronal networks is that they are computationally intensive to train. Due to the fact that computing resources became cheaper and more easily accessible this is less of an obstacle anymore. There has been a lot of research done in the field of deep learning in the past five years and it is still an ongoing trend.

In my thesis I will be focusing on the aspect of food image classification using deep learning. In particular I propose an algorithm that uses an adaptive learning rate in order to decrease the model training time. Chapter 2 illustrates the challenges of food image classification. Chapter 3 provides background as well as the state of the art. In Chapter 4 I justify the choices made on framework and dataset selection. Chapter 5 states my approach and defines the algorithm. Chapter 6 discusses the results and compares them to results found in existing literature. Finally, I will elaborate on possible further steps in chapter 7 and conclude in chapter 8.

Chapter 2

Motivation

2.1 Problem Definiton

The longterm goal is reliably recognise food on a given image. This ability could be used in many applications such as a food logging application that tells the user whether or not a specific meal is healthy to be consumed or not, a price estimation application that can be used in a food market or an application that helps find recipes. In order to perform these tasks the application needs to be able to predict the type and the amount of food.

To ensure good usability it shall only require capturing an image of the food. The user can be requested to take an image of just the region of interest such that no additional localisation needs to be performed.

An essential part required for the decision making is the food classification. When the class of food is known it is easier to estimate portion sizes and to predict how healthy the food is.

Ideally, the classes would resemble the categories of a food pyramid. In [12] Dixon et al. propose the *Food Guide Pyramid* to help decide what to eat on a daily basis. This pyramid consists of the following categories (in order of how much of it should be con-

sumed)

- Bread, Cereal, Rice & Pasta
- Vegetables
- Fruits
- Milk, Yogurt & Cheese
- Meat, Poultry, Fish, Dry Beans, Eggs & Nuts
- Fats, Oils & Sweets

When it comes to healthiness of food it is more important to know they rough category than the exact type of food within the category. For example it is more relevant to know whether the presented food item is bread or meat than it is to know whether it is beef or pork.

The Food-11 dataset described in section 4.3 has similar classes than those presented in the food pyramid. Therefore this dataset is used throughout this paper.

If a meal contains portions of different classes it should be classified using the unhealthiest portion. Many meals will consist partly of one type and partly of another type found in the food pyramid. One example would be a burger that consists of bread, some vegetables, cheese and meat. Whenever that is the case the meal should be classified using the class that is highest in the food pyramid. For the burger this would be the meat.

For some applications a rough categorisation might not be sufficient. In those cases an additional classification step can be added within the domain of a category. The classification task should be a lot simpler if the category is already known.

2.2 Challenges

Food image classification is not a trivial task. This is due to the fact that pictures of food are so diverse. On a low level each picture consists of a certain number of pixels

(usually in the range of tens of thousands) and three colour values per pixel. This is a lot of information and allows for $\left(256^3\right)^{\text{width}*\text{height}}$ possible images. Of course most of those would not be food images anymore but it demonstrates that an approach that handles all possible cases is impossible. Similarly, there is not any one pixel that decides which class the image belongs to but a set of pixels that make up the food portion of the image.

On a higher level one can analyse textures, colour histograms or other features like SIFT [28]. However, those do not allow an accurate classification for two main reasons: high intra-class variability and low inter-class variability.

2.2.1 High Intra-Class Variability

Images in the same class can be very different. There are many factors that account for this. Some of those are linked to how the picture is taken, for instance the perspective and the lighting conditions. On the other hand it also depends on the presentation food itself. One and the same dish can be prepared to look very different. A 5-star chef will decorate a meal differently than a school canteen.

Another reason for high intra-class variability is associated to the class definitions. If the classes are very broad then a lot of different dishes which might not look alike are in the same class. This is displayed in figure 2.1 where a plate of cheese on a wooden table is compared to a glass of milk on a white surface. Those two images look very different even though they are in the same class in the Food-11 dataset. The second example is a chocolate cake compared to macarons. The image of the chocolate cake is almost black and white while the macarons are very colourful yet both images are classified as desserts.

2.2.2 Low Inter-Class Variability

Images in different classes can look similar. This can happen because the same ingredients are used for different types of food but also because some parts of a dish just look very similar to food of a different class. Another possible case is that the surroundings match, for instance if the food was presented on the same (or similar) plate or table. It can also be the case that a dish would fit in several classes and therefore it might be found in more than one class.

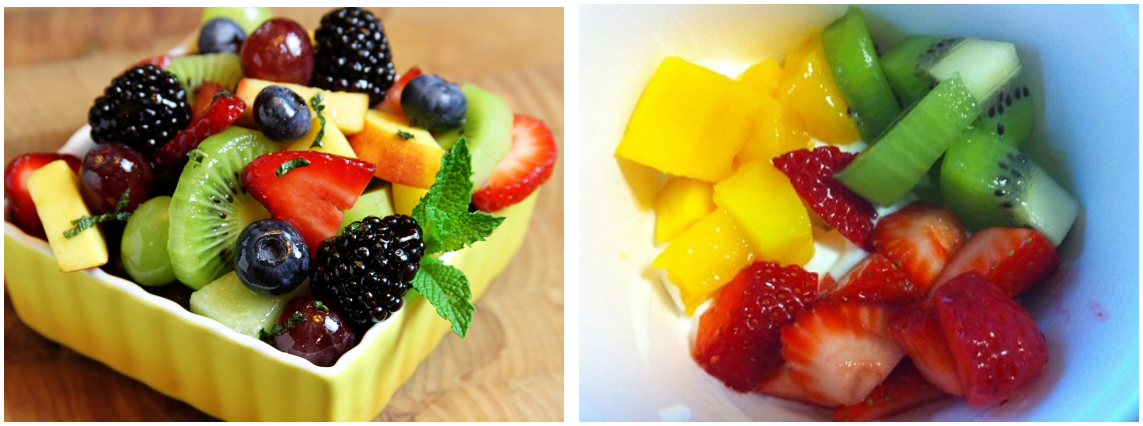
Figure 2.2 provides examples of low inter-class variability in the Food-11 dataset. The bread has a similar colour than the fried food and both have fries as a side-dish. The dessert has the same fruits as those found in a picture of the fruits and vegetables class. The ice-cream underneath the fruits - which makes the food item a dessert - is almost entirely covered by the fruits.

(a) In the class of dairy products: a plate of cheese (left) and a glass of milk (right).

(b) In the class of desserts: a chocolate cake (left) and some macarons (right).

Figure 2.1: Two examples of high intra-class variability in the Food-11 dataset.

(a) A picture of the class bread (left) compared to a picture of the class fried food (right).



(b) A picture of the class fruits and vegetables (left) compared to a picture of the class desserts (right).

Figure 2.2: Two examples of low inter-class variability in the Food-11 dataset.

Chapter 3

Literature Review

This chapter highlights previous work that has been carried out in the field of food image classification.

3.1 Food Image Classification

Back in 2009 Chen et al. [8] performed food image classification using SVM classifiers. They used two different techniques for feature extraction. The first one was to use colour histograms and the other one was to use a bag of SIFT-features. On their own dataset (the PFID) they achieved accuracies 11% and 24% respectively.

Using the same dataset Yang et al. [41] used pairwise local features to improve the classification. They managed to obtain up to 28% accuracy. Furthermore they combined classes of the PFID to obtain 7 major food classes. Using their method with those class definitions they reported an accuracy of 78%.

As discussed by [30] the low success rate shows how hard the task of food image classification really is. A higher percentage was usually only achieved when the number of classes was decreased. The key to successful food image classification is a good feature

extraction.

An optimized bag-of-features model was proposed by Anthimopoulos et al. in 2014 [2]. They used a dataset of 5000 images in 11 classes. Different colour and texture descriptors were analysed and they found that using SIFT on an HSV colour space performs best achieving an accuracy of 77.8%.

Another approach to classify food images is the use of Random Forests. In [5] Bossard et al. created a dataset called Food-101 which consists of 101 classes and approximately 1000 images per class. Their RFDC approach reached an average accuracy of 50.76%. This outperformed alternative classification methods such as SVMs and BOF. The only evaluated method that could not be outperformed was using a CNN which reached an average accuracy of 56.40%.

3.2 Convolutional Neural Networks

Convolutional Neuronal Networks are deep feed-forward neuronal networks. They have been particularly successful in image recognition because they do both unsupervised feature extraction and classification.

As described in [10] a CNN consists of a series of convolution and pooling layers followed by a set of fully connected layers. The first layer is a convolution layer which has a number of kernels (also called filters) that it can learn. It takes a rectangular subsets of the image that correspond to the kernel size (for example using a width and height of 5 pixels the kernel size is 5x5x3 because the image is 3 layers deep as it has 3 colours) as an input. For each position in the image the kernels are applied and the output is sent though an activation function that produces non-linearities. The kernels are the same for each location they are applied on (weight-sharing). The resulting output volume has the width and height of the image dimensions and a depth of the number of kernels.

The following layer is a pooling (also called subsampling) layer. It performs max or mean pooling (taking either the maximum or the mean value of the input) on rectangular subsets of the previous layer to reduce the size of the feature map. There can be a number of convolution and pooling layer pairs in a CNN. The next convolution layer always takes the result from the previous layer as an input (only the first layer operates on the image). After the convolution and pooling layers there is one or more fully connected layers. An example of a typical CNN can be found in figure 3.1. The convolution and pooling layers are responsible for feature extraction while the fully connected layers perform the classification.

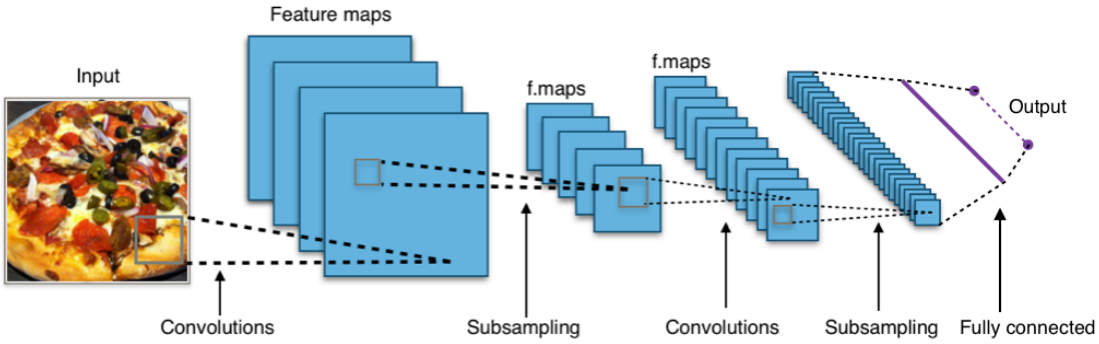


Figure 3.1: The typical structure of a CNN.
Source: Wikimedia. `https://commons.wikimedia.org/w/index.php?curid=45679374`
The original image was slightly modified.

One of the first highly successful CNNs was developed by Krizhevsky et al. in 2012 [26]. Their CNN consists of five convolutional layers (some of which are followed by max-pooling layers) and three fully-connected layers. The first layer has 96 kernels of size 11x11x3, the second convolutional layer 256 kernels of size 5x5x48, the third has 3x3x256 and the following convolutional layers have a size of 3x3x192. The fully connected layers have 4096 neurons each. This setup allowed them to achieve 62.5% accuracy on the ILSVRC-2010 [31] dataset (containing 1000 classes) which outperformed all other methods at that time. This trained CNN was codenamed AlexNet and has later

been used by other researchers.

Since they are deep neuronal networks, CNNs are often refered to as DCNNs.

3.3 Food Image Classification using DCNNs

The key to good food image classification is gathering a lot of different features. Deep convolutional neuronal networks are surprisingly good at that. Without them being explicitly trained to do so, the first layers usually perform oriented edge, blob and pattern detection. Another reason why DCNNs are successful is that they can be trained to perform recognition, localisation and detection in the same network [34]. When enough training data is provided they can also learn to ignore surrounding noise. Thus they are not susceptible to missing segmentation.

Kawano et al. in 2014 [23] used the DCNN from [26] (AlexNet) to perform food image classification on the UEC Food-100 dataset [29]. For each picture they extracted the network signals just before the last layer of the DCNN to create their feature vector. Then they used a SVM for classification. In combination with conventional features coded into Fisher Vectors they achieved a best classification accuracy of 72.3%. This showed that DCNN features can be successfully used for food image classification.

A year later, in 2015, Kawano and Yanai used DCNNs in [40]. Their approach was to pre-train with a huge dataset (ImageNet) and fine-tune using a small dataset (around 100 training images per class). For fine-tuning they used the UEC Food-100 and the UEC Food-256 (which they proposed in [24]) and gained classification accuracies of 78.77% and 67.57% respectively. This approach is both simple and effective which is why I decided to use it as a base for my research.

Farooq et al. [16] performed image classification on the PFID using AlexNet. Since the dataset only contains a total of 1098 images they just used the DCNN to extract fea-

tures. For classification a linear SVM was used. They obtained accuracies of 70.13% for the original 61 classes and 94.01% on the more general 7 classes. Once again showing that features extracted from DCNN are more valuable than conventional features.

The current best classification results on the datasets UEC Food-100, UEC Food-256 and ETH Food-101 [5] were accomplished by Hassannejad et al. in 2016 [18]. They used a (54 layers) DCNN with an Inception V3 architecture [38] which was pre-trained on the ImageNet dataset and they fine-tuned it by changing the classification layers. This gave them top-1 accuracies of 81.45% (UECFOOD-100), 76.17% (UECFOOD-256) and 88.28% (ETH Food-101) which are the best published results to date.

3.4 Adaptive Learning Rate

Adaptive learning rates are mainly used to decrease computation time. As stated in [32] a backpropagation algorithm cannot converge to a regular minimum if the learning rate η is too high. However, by decreasing the learning rate more iterations are required to converge. Thus if the learning rate is too small the computational resources available might not be sufficient to obtain the desired result. To adjust the learning rate to the curvature of the cost function a dynamically adapting learning rate can be used.

One simple way of adapting the learning rate was proposed by Salomon and van Hemmen in [33]. They defined the new learning rate as

$$\eta_{t+1} = \begin{cases} \eta_t \zeta & \text{if } E(w_t - \eta_t e_k \zeta) \leq E(w_t - \eta_t e_k / \zeta). \\ \eta_t / \zeta & \text{otherwise.} \end{cases} \tag{3.1}$$

where E is the cost function, e_k is the unit vector in direction of $\nabla E(x_k)$ and ζ is a constant. There are two possibilities: either the learning rate is increased (by a factor) or it is

decreased (by a factor) depending on which result of the cost function is lower. In simple terms it applies the learning rate that performs better. Each step the learning rate adapts itself this way. The constant ζ is said to be optimal at 1.8.

In 2011, Duchi et al [13] developed an algorithm called AdaGrad. It is based on gradient descent and collects information from gradients over several iterations to set the learning rate. It updates the learning rate per dimension using the following rule

$$\Delta\eta_t = -\frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}} g_t \tag{3.2}$$

where $\Delta\eta$ is the change in learning rate, g_t is the gradient of step t (so the denominator calculates the $\ell 2$-norm of all previous gradients) and η is the global learning rate. Although the algorithm gave good results it still had a hyper-parameter that had to be hand-tuned (the global learning rate).

A year later in 2012 Zeiler [42] proposed AdaDelta, an algorithm that addressed the issue of hyper-parameter selection and the fact that AdaGrad had (inherently) decaying learning rates. To do so it only takes the gradients of the past n iterations into account (rather than all previous gradients). Since even storing n gradients is inefficient the accumulation is implemented as an exponentially decaying average of the squared gradients. This average is calculates as follows

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2 \tag{3.3}$$

where ρ is a decay constant. Moreover they calculated

$$\mathrm{RMS}[g]_t = \sqrt{E[g^2]_t + \varepsilon} \tag{3.4}$$

with ε as a constant which ensures progress is being made in the first iteration (where

$\Delta\eta_0 = 0$) and when previous updates are small. The update rule is defined as

$$\Delta\eta_t = -\frac{\text{RMS}[\Delta\eta]_{t-1}}{\text{RMS}[g]_t} g_t \tag{3.5}$$

The AdaDelta algorithm has been proven successful when applied to large neuronal networks [11].

Adaptive learning rates have been used in the field of image classification. Singh et al. [35] used a layer-per-layer approach where each layer had its own learning rate. They calculated the learning rates using

$$\eta_{t+1} = \eta_t \left(1 + \log\left(1 + \frac{1}{\|g_t\|_2}\right)\right) \tag{3.6}$$

When the gradients g_t are large the result is the previous learning rate η_t but when gradients are small (ie. low curvature) the learning rate is scaled up to ensure faster learning. This approach can be used for any optimisation technique that used a global learning rate. Therefore it works with SDG and AdaGrad. It has been applied on the ImageNet dataset using AlexNet and a SDG optimiser resulting in a 15% reduction in training time (over vanilla SDG).

No literature was found that mentioned adaptive learning rates on the topic of food image classification specifically. This is why I wanted to approach this topic in my thesis.

Chapter 4

Setup

4.1 Framework

Since implementing all the deep learning functionalities from scratch would be far beyond the scope of this project, an existing deep learning framework is used. Table 4.1 lists some of the most popular deep learning frameworks. I have never worked on deep learning algorithms before so I do not have any experience in any of the frameworks. As a personal preference I select Python as the programming language to be used. There are two frameworks, namely TensorFlow and Theano, which are native in Python and two more, Caffe and Torch, that provide Python wrappers.

Framwork	**Language**	**Python Wrapper**
TensorFlow	Python	-
Caffe	C++	PyCaffe
Torch	Lua	PyTorch
Theano	Python	-
DL4J	Java	-

Table 4.1: A list of the most common publicly available deep learning frameworks.

I have tried both Caffe [22] and TensorFlow [1] and came to the conclusion that TensorFlow suits me better. The main criterion for this choice was the stability. TensorFlow works out-of-the-box whereas Caffe caused some initial difficulties. Furthermore Caffe does not officially support Python3. TensorFlow provides better documentation and examples and has an easy to use visualisation tool called TensorBoard.

TensorFlow is not the most performance optimised framework but it has proven to be sufficient for this project.

4.2 Environment

All of the trainings and simulations have been performed on a personal computer. Specifically a Lenovo™ ThinkPad T450s with the following specifications:

- CPU: Intel® Core™ i5-5300U CPU @ 2.30GHz × 4
- Memory: 11GB RAM
- Graphics: Intel® HD Graphics 5500 (Broadwell GT2)
- Storage: Toshiba THNSFJ25 256GB SSD
- OS: Ubuntu 16.04 LTS 64-bit

I developed in Python version 3.5.2 with TensorFlow version 1.1.0. Since the GPU does not support Cuda, all prosessing was done on the CPU.

To generate plots I used `pyplot` of Matplotlib [21] in version 2.0.1.

4.3 Dataset

To perform the classifications a set of food images is required. I had two possible choices to acquire them: a) create my own dataset or b) use an existing dataset. Creating a dataset would provide me with the opportunity of selecting exactly the classes I require. However, dataset creation is a lot of work because each image has to be labeled and a lot of images

are required. Moreover, using a novel dataset means that the results are not comparable. For this reason I decided against creating a new dataset.

4.3.1 Choice of Dataset

There are several food image datasets publicly available. Some of them have been mentioned in chapter 3. Table 4.2 contains a list of available food image datasets.

As discussed in section 2.1 the dataset for this application should have classes representing the categories of the food pyramid. Most of the available datasets have more narrow class definitions and therefore a higher number of classes.

Name	Year	Number of Classes	Number of Images	Proposed in
PFID	2009	61	1098	[8]
UEC FOOD-100	2012	100	9060	[29]
ETH Food-101	2014	101	101 000	[5]
UEC FOOD-256	2014	256	31 397	[24]
no-name	2014	11	4868	[2]
UNICT-FD889	2015	889	3583	[15]
UNICT-FD1200	2016	1200	4754	[14]
Food-11	2016	11	16 643	[36]

Table 4.2: A selection of food image datasets that can be found in the literature.

One of the datasets that has suitable class definitions was proposed in [2]. It contains 11 classes: Bread, Breaded food, Cheese, Egg products, Legumes, Meat, Pasta, Pizza, Potatoes, Rice, and Vegetables. However, it consists of less than 5000 images which is not ideal in a deep learning context.

The Food-11 dataset [36] was published in 2016 by the Multimedia Signal Processing Group at EPFL. It contains a total of over 16000 images in 11 classes. The images were acquired partly from other datasets like the ETH Food-101 and UEC FOOD-256 and

partly from social media platforms. The dataset is divided into three subsets: a training set (which contains roughly 60% of the images), a validation set (containing 20%) and an evaluation set (which contains the remaining 20%).

The dataset contains a total of 16643 images. The distribution of those images into the 11 classes as well as into the training, validation and evaluation set can be found in table 4.3. Figure 4.1 shows some example pictures of all the different classes of the dataset.

Class	**ID**	**training-set**	**validation-set**	**evaluation-set**	**total**
Bread	0	994	362	368	1724
Dairy products	1	429	144	148	721
Desserts	2	1500	500	500	2500
Eggs	3	986	327	335	1648
Fried food	4	848	326	287	1461
Meat	5	1325	449	432	2206
Noodles/Pasta	6	440	147	147	734
Rice	7	280	96	96	472
Seafood	8	855	347	303	1505
Soup	9	1500	500	500	2500
Vegetable/Fruit	10	709	232	231	1172
Total	11	9866	3430	3347	16643

Table 4.3: The number of images in each class of the Food-11 dataset.

During development I primarily used the Food-11 dataset. I also different datasets for testing as discussed in section 6.5. Unless stated otherwise all trainings in this thesis were performed using Food-11.

(a) bread

(b) eggs

(c) desserts

(d) cheese

(e) fried food

(f) meat

(g) seafood

(h) rice

(i) noodles

(j) soups

(k) fruits & vegetables

Figure 4.1: Some examples of the 11 classes in the Food-11 dataset.

4.3.2 Drawbacks

Although the Food-11 dataset fits best for the task from all the datasets analysed it is far from ideal. There are several details in the class definitions that are not optimal.

My primary criticism is towards the class of bread. Even though bread it is low in the food pyramid and thus one should be allowed to consume more of it, the bread class contains some very unhealthy products. The bread class seems to be defined in a way that as soon as a food item contains some form of bread it is classified as such. This means that pizza, burgers and wraps are classified as bread even though for most of them the bread portion is less than 50% of the meal. For the given application it would be better if burgers would be classified as meat, pizza as cheese (or meat or sea-food depending on the topping) and wraps as egg, vegetable or meat depending on the filling.

The fruits and vegetables class should be split into two classes. Fruits and vegetables are necessary for a balanced and healthy diet [7]. The Food-11 dataset contains only one class for both fruits and vegetables. If the application cannot distinguish between those two classes it opens the possibility for a user to just eat fruits. This is bad because fruits on average have higher sugar levels [27] than vegetables. Essentially a green salad is put in the same class as a fruit salad which is not reasonable from a food intake analysis perspective.

The soup class is redundant. A soup is essentially (mashed) food with water. The task is not to analyse water intake otherwise a drink classification should be implemented. There are many types of soups ranging from vegetable soups over cream and fish soups to soups with meat or noodles. Those should be classified with the corresponding label: fruits & vegetables, dairy, seafood, meat or noodles.

All the disadvantages of the Food-11 dataset mentioned in this section concern the application not the classification process. Properties that influence the classification are

image quality, number of pictures and correct labels (among others). All of those are reasonable with this dataset. Since this paper focuses on the classification aspect it is justifiable to proceed with the Food-11 dataset.

4.4 Network Architecture

There are many pre-trained DCNNs available. One of them, the AlexNet, was mentioned in section 3.2. It has won the ILSVRC 2010 [31] which is an image classification challenge on the ImageNet dataset.

GoogLeNet [37] is an incarnation of the Inception architecture. It has won the ILSVRC in 2012. This architecture has been updated in 2015 to Inception V3 [38].

In their paper [30] Mezgec and Seljak proposed their own architecture called NutriNet to perform food and drink classification. More importantly, they tested other architectures and observed that NutriNet outperforms AlexNet, has comparable results to GoogLeNet and is only outperformed by ResNet [19].

Even though ResNet has won the ILSVRC 2015, I decided to use Inception V3 for this project. At the time of decision making no direct comparison between ResNet and Inception V3 was available. Furthermore, TensorFlow already supported a Inception V3 architecture so it just had to be imported as a module.

The Inception V3 architecture is shown in Figure 4.2. This architecture introduced so called Inception modules which not only go in depth but also in width. This allows to have smaller kernels (3x3 instead of 5x5 or 7x7) which is less computationally expensive. The convolution and pooling layers are quite complicated but it is not necessary to understand it in detail since it is only used for feature extraction. The important layers for this project are the last two: the fully connected and the softmax layer. The rest will be left untouched.

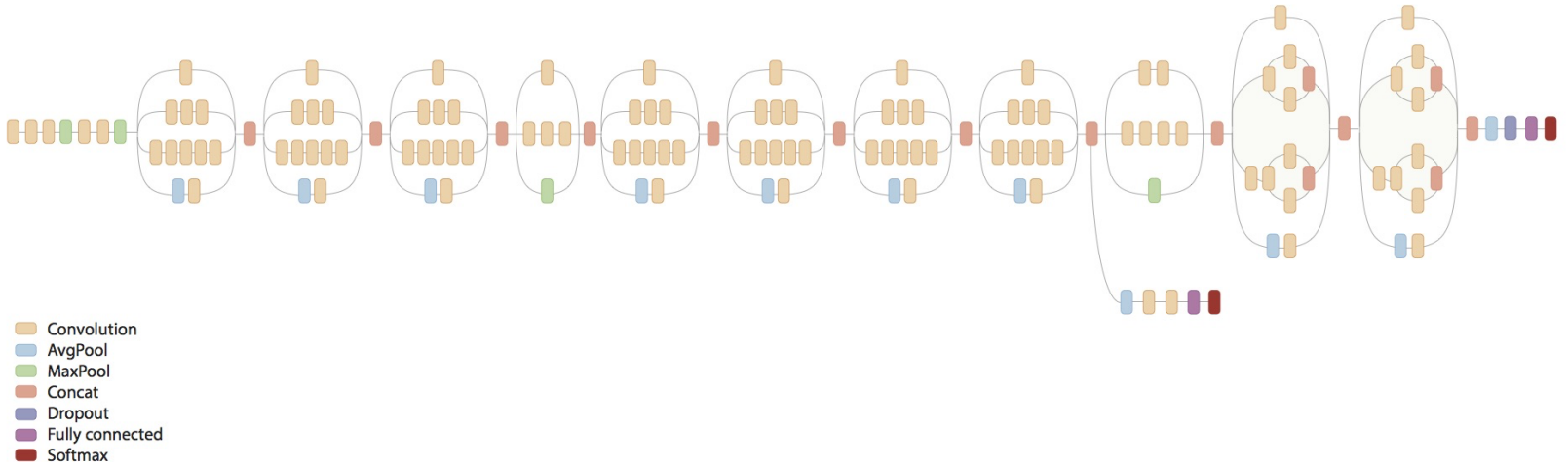


Figure 4.2: The Inception V3 architecture as implemented in TensorFlow.
Source: `https://github.com/tensorflow/models/blob/master/inception/g3doc/inception_v3_architecture.png`

4.5 Mobile Computing

The classification step has to be efficient enough that it can be performed on a mobile phone. Once a model has been trained using TensorFlow it can be stored with all its weights. This model can then be optimized to reduce its size and copied into a mobile application. TensorFlow provides support for Android and iOS. I developed an Android application[1] as a prove of concept. The classification on a OnePlus2 device running Android 6.0.1 takes approximately 5 seconds.

[1] `https://github.com/aschmidhofer/UpYourVeggies`

Chapter 5

Methods

In this chapter I discuss the step by step development of the algorithm.

5.1 Retraining

Training a DCNN from scratch is impractical. It would require millions of pictures and would take several weeks to train (even on an HPC system). Section 3.3 of my literature review showed that it is feasible to use a pre-trained network and just change the classification layer (the last layer). I decided to use an Inception V3 architecture which was trained on the ImageNet dataset. To apply it to my domain the last layer had to be changed to contain 11 instead of the 1000 nodes since there are 11 classes in the Food-11 dataset.

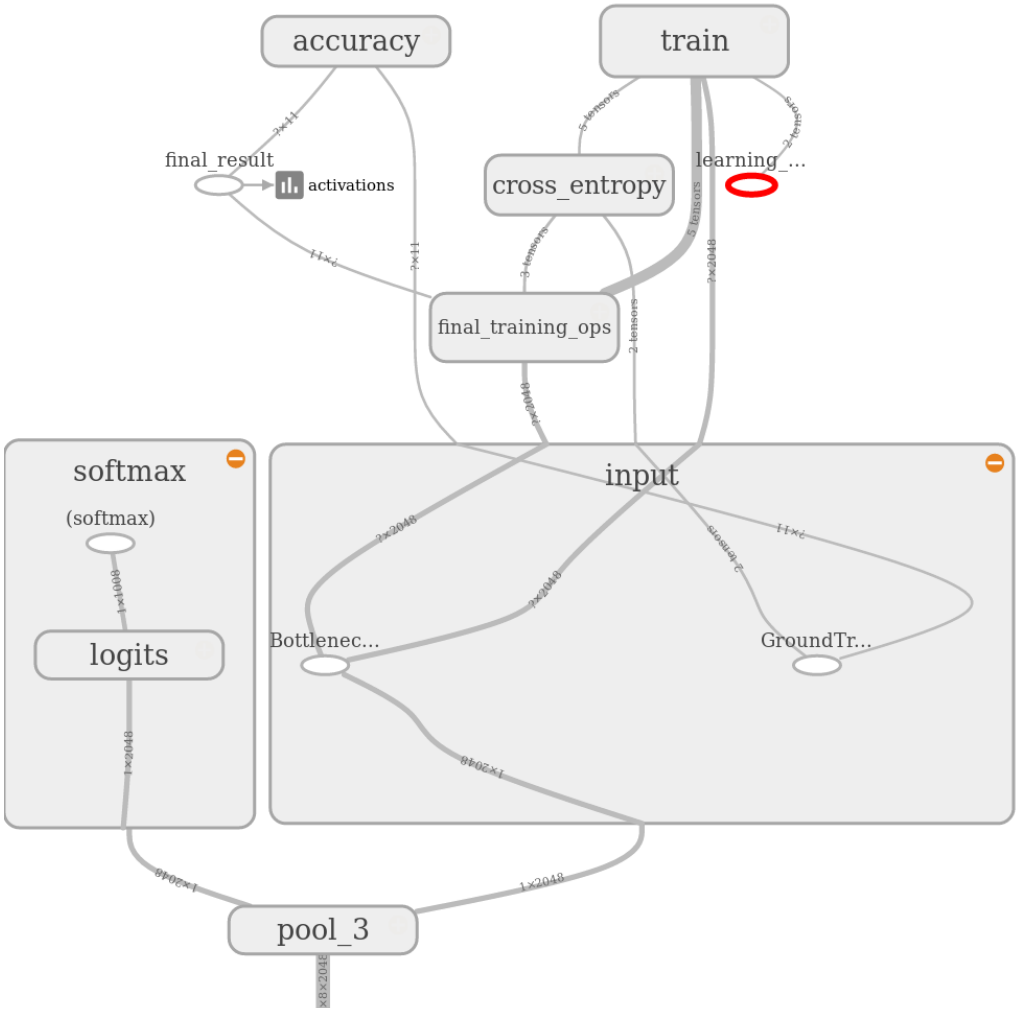


Figure 5.1: The architecture of the last layer of the DCNN as shown in TensorBoard. `pool_3` is the last pooling layer of the Inception V3 network, it provides the 2048 values that make up the `bottleneck`. The `GroundTruth` is provided at training time. `train` contains the optimiser algorithm (eg. SDG) and the red eclipse represents the adjusted learning rate that is fed into the network. `final_result` contains the output of the network.

Figure 5.1 shows the modified structure of the last layer of the network. Hidden from this figure are the previous Inception V3 layers. They are the same as shown in figure 4.2. It can be observed that the last layer takes the values of the last pooling layer (the so called `bottleneck`) and uses them as a feature-map. The ground truth is provided in the form of a 1-dimensional boolean matrix: it contains a 1 at the index of the class the image belongs to and zeros for the remaining 10 entries. The node `final_result` contains 11 confidence values (one for each of the classes) which can be used in combination with the ground truth to calculate the accuracy.

5.2 Feature-map Caching

Since a `bottleneck` for an image is used several times throughout the training process it is efficient to cache it. The Inception layers do not change, only the newly added classification layer is re-trained. Thus all of the `bottlenecks` can be calculated once and then just read from file.

5.3 Training Procedure

The Food-11 dataset is already split in three subsets. This ensures that the results of different researchers are comparable. The training set is used for training, the validation set is used to ensure that the training does not adjust too heavily on the training set (overfittig) and the evaluation set is used to provide yet another step to check that the results are not relying on the validation set.

In order for the adaptive learning rate to work reliably a batch-learning approach is used. This means that all of the 9866 images in the training set are used in each training step. After each training step the validation accuracy is calculated using all 3430 images

in the validation set. Training and validation are repeated for several iterations. In the very end of the procedure the evaluation set of 3347 images is tested on the model to calculate the final accuracy.

All accuracies are calculated using

$$\text{accuracy} = \frac{\text{number of correctly predicted classes}}{\text{total number of images tested}} \tag{5.1}$$

5.4 Stopping Criterion

With the given setup the training could be continued indefinitely. Until a certain point the results improve but if training is continued beyond this point the network will go into overfitting. Overfitting is when the network adjusts too tightly to the training set and results for a more general set get worse. figure 5.2 shows an example of overfitting where the training accuracy keeps increasing but there is no improvement for the validation set.

There are two ways to stop the training and avoid overfitting. The easiest is to provide the algorithm with a fixed number of iterations that it should go though. However, finding the correct number of iterations to sufficiently train the network and at the same time not go into overfitting can be tricky. The ideal number of training steps depends on many factors such as the size of the dataset, the learning rate, the batch size etc. This is why I decided on using the second possibility: applying a stopping criterion.

The simplest stopping criterion is to continue training until the validation accuracy decreases. However, since the network has many neurons it can happen that the validation accuracy drops even when the general trend is still rising. To overcome this issue I compare the last k iterations. Let $V(t)$ be the validation accuracy at iteration t, then the

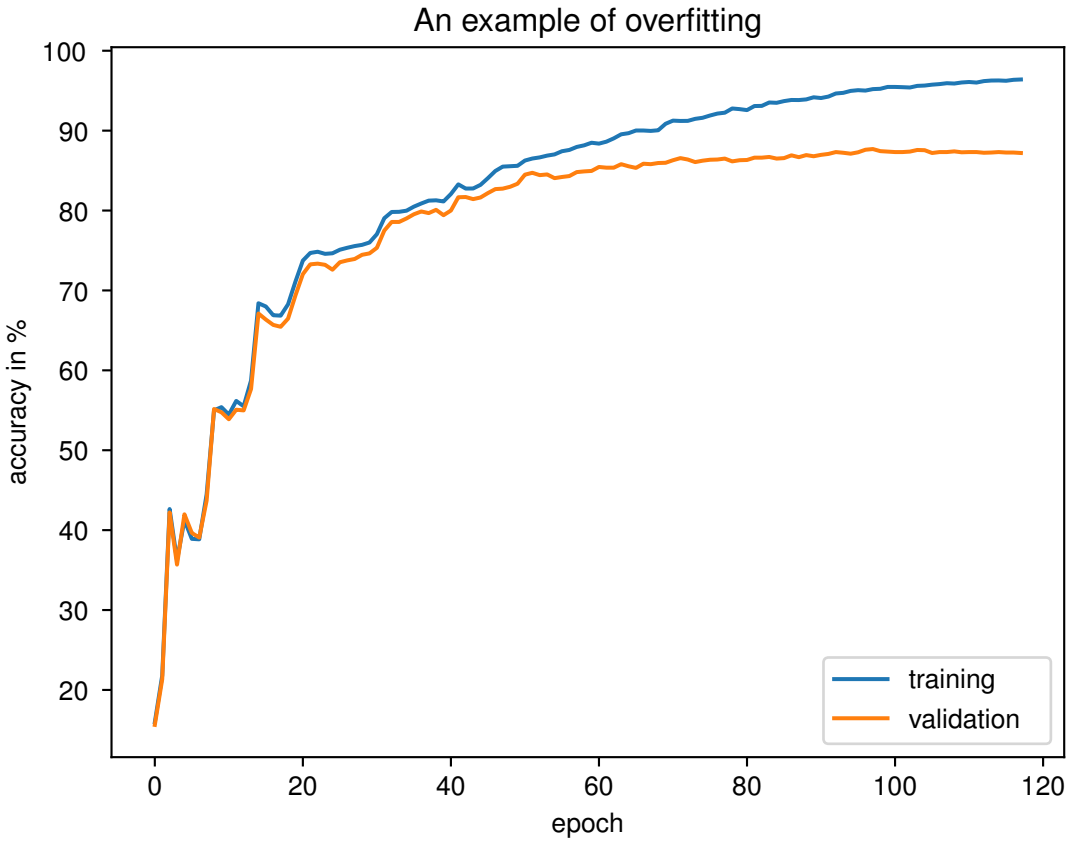


Figure 5.2: This plot shows what happens when training proceeds into overfitting. The training accuracy continues to grow towards 100% while the validation accuracy stays the same.

stopping criterion is defined as:

$$\text{if } \forall i \in [\![1,k]\!] : V(t) \leq V(t-i) \tag{5.2}$$

Only if the current validation accuracy is smaller than those from the previous k steps the training is supposed to stop. The constant k should be adjusted to the size of the dataset. A bigger dataset produces fewer oscillations and therefore a smaller k can be used. For the Food-11 dataset $k = 10$ has proven to work well.

5.5 Adaptive Learning Rate

The backward propagation step uses stochastic gradient descent (SGD) to update the weights. A learning rate has to be defined in order for SGD to work. If the learning rate is too small the training process will take unnecessarily long. If it is too big then the results are undesirable. Figure 5.3 shows the usage of different learning rates on the Food-11 dataset.

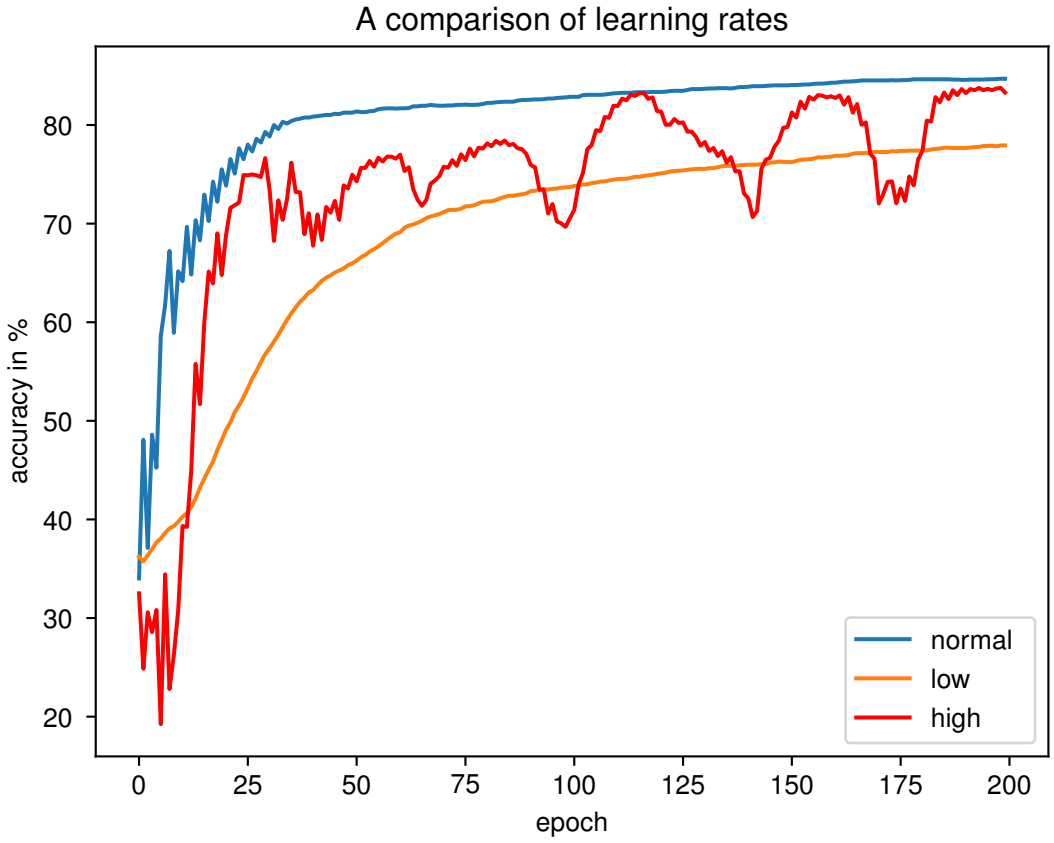


Figure 5.3: This plot shows the validation accuracies of three trainings using SDG with different learning rates. A reasonable learning rate (0.1) is shown in blue. A learning rate that is too low (0.01) an therefore does not learn as quickly is shown in orange and a learning rate that is too high (1.0) is shown in red.

Finding a good learning rate can be a lengthy trial and error process. It would be

better if the algorithm would adjust its learning rate automatically. I propose an adaptive learning rate gradient descent algorithm. For better readability it shall be referred to as ALRGD.

5.5.1 Algorithm Development

I started out by using Salomon's algorithm. As an error function I used the training accuracy, the higher the accuracy, the smaller the error. Let $A_t(\eta)$ be the training accuracy at iteration t using the learning rate η then I defined the new learning rate η_{t+1} as follows:

$$\eta_{t+1} = \begin{cases} \frac{\eta_t}{\zeta} & \text{if } A_{t+1}(\eta_t \zeta) < A_{t+1}(\frac{\eta_t}{\zeta}). \\ \eta_t \zeta & \text{otherwise.} \end{cases} \tag{5.3}$$

The constant ζ was arbitrarily chosen to be 1.5. Any values for ζ between 1.2 and 1.8 gave similar results. As shown in figure 5.4 the algorithm seemed promising because the validation accuracy stayed close by the training accuracy. An initial learning rate of 10 was used and the learning rate decreased to stay between 0.5 and 4.5. After 45 steps of training it stopped with an evaluation accuracy of 82.5%.

Per definition of the algorithm the learning rate fluctuates. However, the most appropriate learning rate might already be found. So I changed the algorithm to allow it to stay at the same learning rate. The new learning rate definition is therefore:

$$\eta_{t+1} = \begin{cases} \eta_t & \text{if } A_{t+1}(\eta_t \zeta) < A_{t+1}(\eta_t) \text{ and } A_{t+1}(\frac{\eta_t}{\zeta}) < A_{t+1}(\eta_t). \\ \frac{\eta_t}{\zeta} & \text{if } A_{t+1}(\eta_t \zeta) < A_{t+1}(\frac{\eta_t}{\zeta}). \\ \eta_t \zeta & \text{otherwise.} \end{cases} \tag{5.4}$$

The result can be observed in figure 5.5: The learning rate stays below 0.2 and then

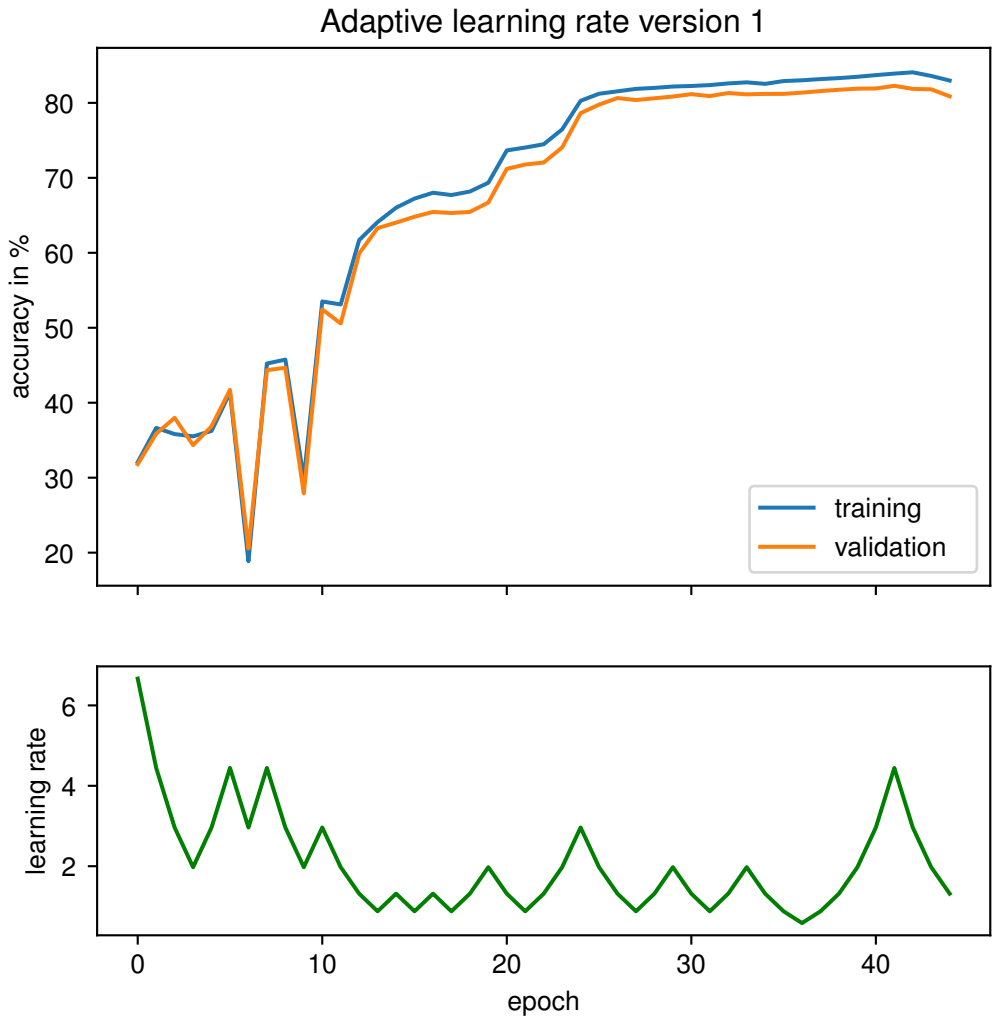


Figure 5.4: The upper plot shows the training and validation accuracies of the ALRDG algirithm in the first version. The lower plot shows the corresponding learning rates that were used.

suddenly spikes. This is a sign that the algorithm is unstable. This is the case because the training accuracy cannot be used as an optimizer. Even if the model quality increases the training accuracy might decrease simply because of the composition of the training set.

To overcome this instability I had to rethink the approach. The solution is as simple

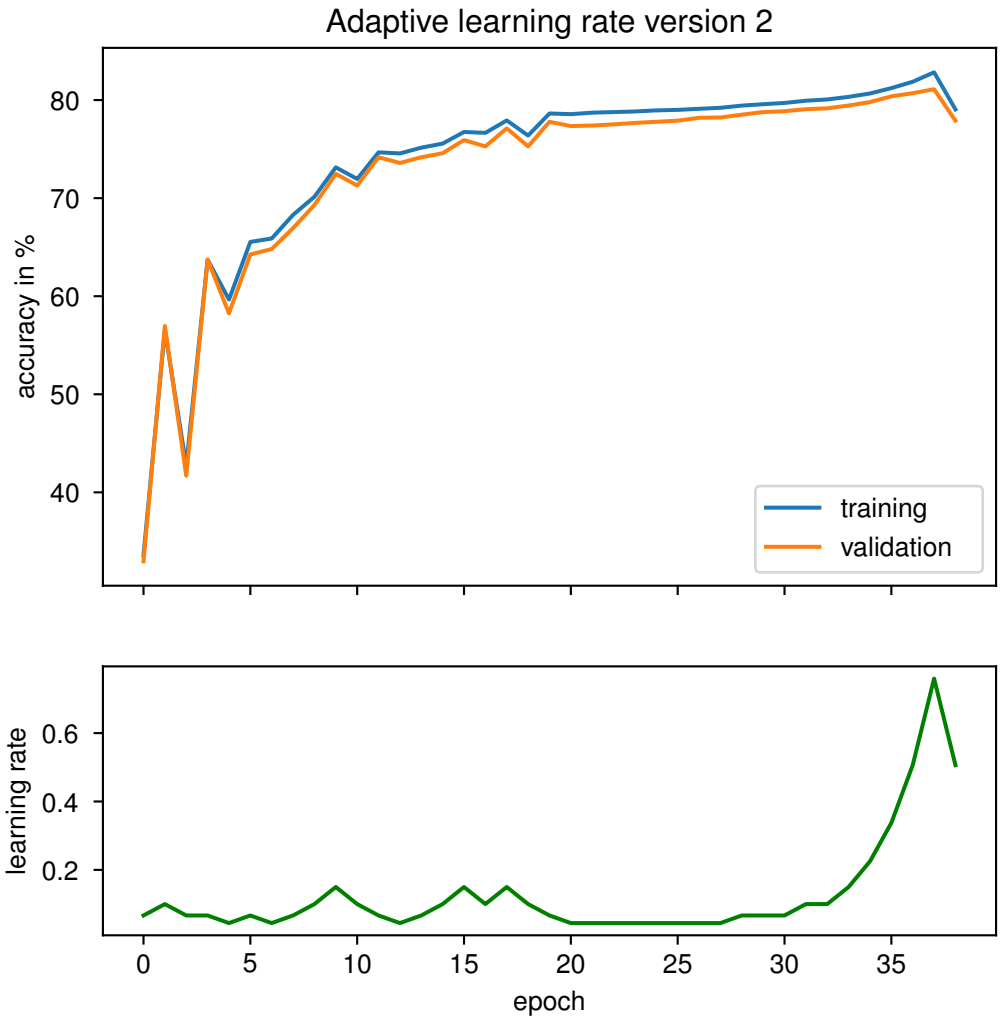


Figure 5.5: Showing the accuracies and learning rate progress of version two.

as it should have been obvious in the beginning: Use the cross entropy for comparison. After all, this is what to network is minimizing.

With $c_t(\eta)$ being the cross entropy at step t using learning rate η, the new learning

rate is

$$\eta_{t+1} = \begin{cases} \eta_t & \text{if } c_{t+1}(\eta_t) < c_{t+1}(\eta_t \zeta) \text{ and } c_{t+1}(\eta_t) < c_{t+1}(\frac{\eta_t}{\zeta}). \\ \frac{\eta_t}{\zeta} & \text{if } c_{t+1}(\frac{\eta_t}{\zeta}) < c_{t+1}(\eta_t \zeta). \\ \eta_t \zeta & \text{otherwise.} \end{cases} \tag{5.5}$$

Figure 5.6 shows the result of a training with an initial learning rate of 0.01. It should be noted that the boundaries the learning rate stays in are well defined compared to previous versions. This attests for a more stable algorithm. The evaluation accuracy after 245 epochs is 87.8%.

Observing the learning rate development I noticed that oscillations increase throughout the training. In the beginning of the training η bounces between the two values 0.08 and 0.11. At around 50 iterations it settled on 0.11 and afterwards it started fluctuating between 0.11 and 0.17. After epoch 130 it also started jumping to 0.25. This behavior indicates that the ideal learning rate for this problem is slightly increasing throughout the training. However, since the learning rate is only increased and decreased by a multiplicative factor, it cannot reach higher values of η with the same accuracy as lower ones. From a given initial learning rate η_0 the possible values for η_t are given by the exponential $\eta_0^{\zeta k}$ where k is an integer.

Intuitively, the oscillations of η should become smaller during training (and not bigger). So I implemented the next change in the algorithm: a variable ζ. The idea is that since in the end the cross entropy does not change as much anymore, the learning rate should also not change as much. In the beginning when a lot of progress is made decreasing the cross entropy the tested learning rates should cover a wider spectrum to find the best learning rate more quickly. To achieve this ζ is set to be proportional to the change in cross entropy.

$$\zeta_{t+1} = 1 + a + m\frac{(c_{t-1}(\eta_{t-1}) - c_t(\eta_t))}{c_t(\eta_t)} \tag{5.6}$$

where a is an additive constant and m is a multiplicative constant. The 1 in the formula is required since ζ is a multiplicative value (and for multiplication 1 is the identity element). The additive constant a is added such that there is a possible change in learning rate even if little or no progress is made on the cross entropy. The suggested value for a is 0.2. The multiplicative constant m defines how much a change in cross entropy affects the possible learning rates. Since this constant is dependent on the absolute value of the cross entropy, it has to be adjusted to the problem. Ideally you want an average value of ζ_t between 1.2 and 2.0 (initially it can be higher). For the batch-learning approach on the Food-11 dataset I suggest $m = 10$. The new learning rate has to be calculated accordingly:

$$\eta_{t+1} = \begin{cases} \eta_t & \text{if } c_{t+1}(\eta_t) < c_{t+1}(\eta_t \zeta_{t+1}) \text{ and } c_{t+1}(\eta_t) < c_{t+1}(\frac{\eta_t}{\zeta_{t+1}}). \\ \frac{\eta_t}{\zeta_{t+1}} & \text{if } c_{t+1}(\frac{\eta_t}{\zeta_{t+1}}) < c_{t+1}(\eta_t \zeta_{t+1}). \\ \eta_t \zeta_{t+1} & \text{otherwise.} \end{cases} \tag{5.7}$$

Figure 5.7 shows the result of a training using the algorithm with changing ζ. It can be observed that the oscillations become smaller towards the end of the training. This also positively influences the accuracy. After only 160 training steps an evaluation accuracy of 87.0% was reached.

Even though this approach has proven to work, it assumes that the cross entropy is monotonically decreasing. In other words it assumes that $(c_{t-1}(\eta_{t-1}) - c_t(\eta_t))$ is always positive. What if that is not the case? Granted, the harm done is small since even if $\zeta_{t+1} < 1$ there will still be a low learning rate $\eta_t \zeta_{t+1}$ and a high learning rate $\frac{\eta_t}{\zeta_{t+1}}$. However, as discussed in [32] a stabilisation should be performed. The simplest stability term is defined as

$$\eta^*_{t+1} = \frac{\eta_t}{\zeta^{k+1}} \tag{5.8}$$

using the smallest $k \in \{0,1,2\dots\}$ such that $c_{t+1}(\eta^*_{t+1}) \leq c_t(\eta_t)$. Since in my algorithm ζ is not a constant it should be replaced by a constant, yielding

$$\eta^*_{t+1} = \frac{\eta_t}{2^{k+1}} \tag{5.9}$$

Applying those changes results in the final version of the ALRGD algorithm.

5.5.2 Final Version

Using ζ_{t+1} from equation 5.6 and η^*_{t+1} from equation 5.9 the adaptive learning rate is defined as

$$\eta_{t+1} = \begin{cases} \eta^*_{t+1} & \text{if } c_{t-1}(\eta_{t-1}) < c_t(\eta_t). \\ \eta_t & \text{else if } c_{t+1}(\eta_t) < c_{t+1}(\eta_t \zeta_{t+1}) \text{ and } c_{t+1}(\eta_t) < c_{t+1}(\frac{\eta_t}{\zeta_{t+1}}). \\ \frac{\eta_t}{\zeta_{t+1}} & \text{else if } c_{t+1}(\frac{\eta_t}{\zeta_{t+1}}) < c_{t+1}(\eta_t \zeta_{t+1}). \\ \eta_t \zeta_{t+1} & \text{else.} \end{cases} \tag{5.10}$$

During an average training with around 500 steps the case of η^*_{t+1} is applied less than 5 times.

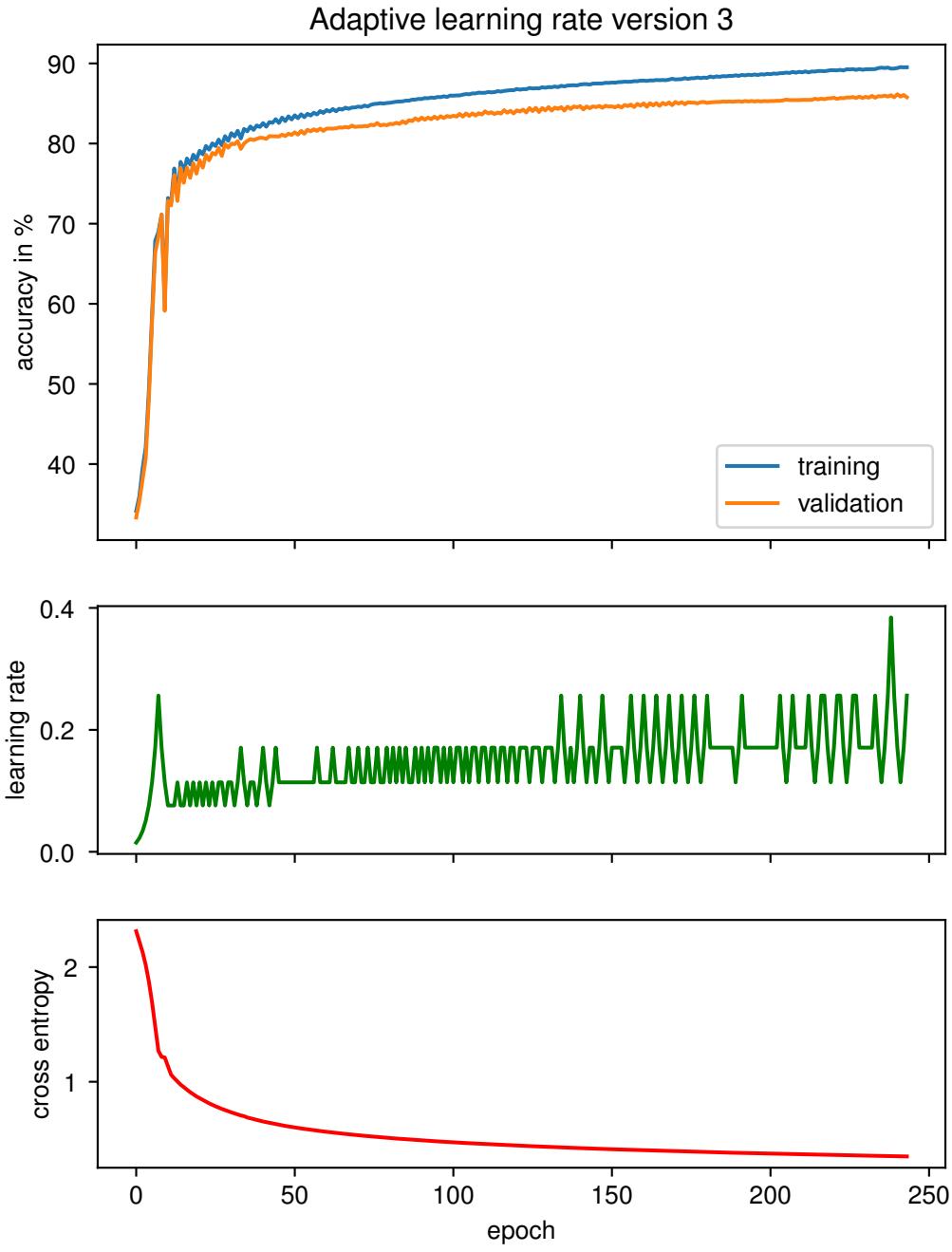


Figure 5.6: This plot shows the accuracies, learning rate and cross entropy development of the training using ALRGD version three.

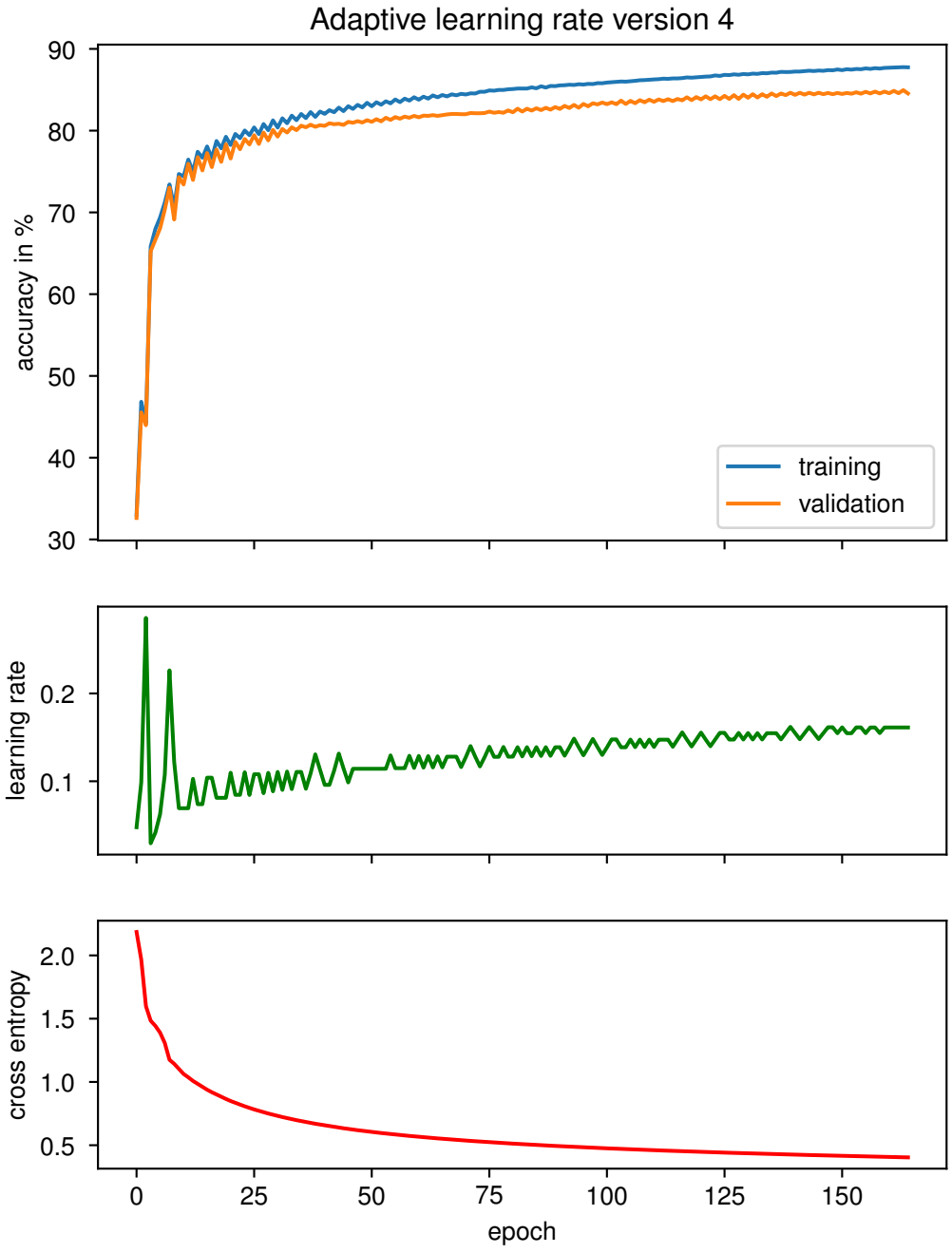


Figure 5.7: This plot shows the accuracies, learning rate and cross entropy development of the training using ALRGD version four. In this version the amount the learning rate can change is given by the difference in cross entropy. Thus the further into the training, the smaller the changes. An overall slight upward trend in learning rate is noticeable.

5.6 Initial Learning Rate

The adaptive learning rate update depends on the learning rate of the previous iteration. Thus the algorithm has to be provided with an initial learning rate η_0.

If the initial learning rate is too small it takes a bit longer to to find a good value but if it is too large it might negatively effect the whole training. This is due to the fact that if a training step is performed with a too high learning rate, it might shift the weights too far to a flat area of the activation function. Even if this happens at only the first training step the final accuracy will be significantly lower.

One such example is shown in figure 5.8. The cross entropy stays at value of 50 when it usually advances below 1 and the learning rate ends up at 10 (when it usually converges to 0.20). After 260 epochs the evaluation accuracy is only 86.5%.

The solution is to apply the same technique as for the stabilisation defined in equation 5.9.

$$\eta_0^* = \frac{\eta_0}{2^k} \tag{5.11}$$

using the smallest $k \in \mathbb{N}_0$ such that $c_0(\frac{\eta_0}{2^{k+1}}) > c_0(\frac{\eta_0}{2^k})$. The user-set initial learning rate η_0 is continuously divided by two as long as the cross entropy for the resulting learning rate decreases. As soon as the cross entropy increases the previous value for the learning rate is applied. This means that the adjusted initial learning rate η_0^* can also be the provided learning rate η_0 (for instance if the value is very small).

There is no need to apply this procedure to increase the initial learning rate as the algorithm adjusts quickly enough.

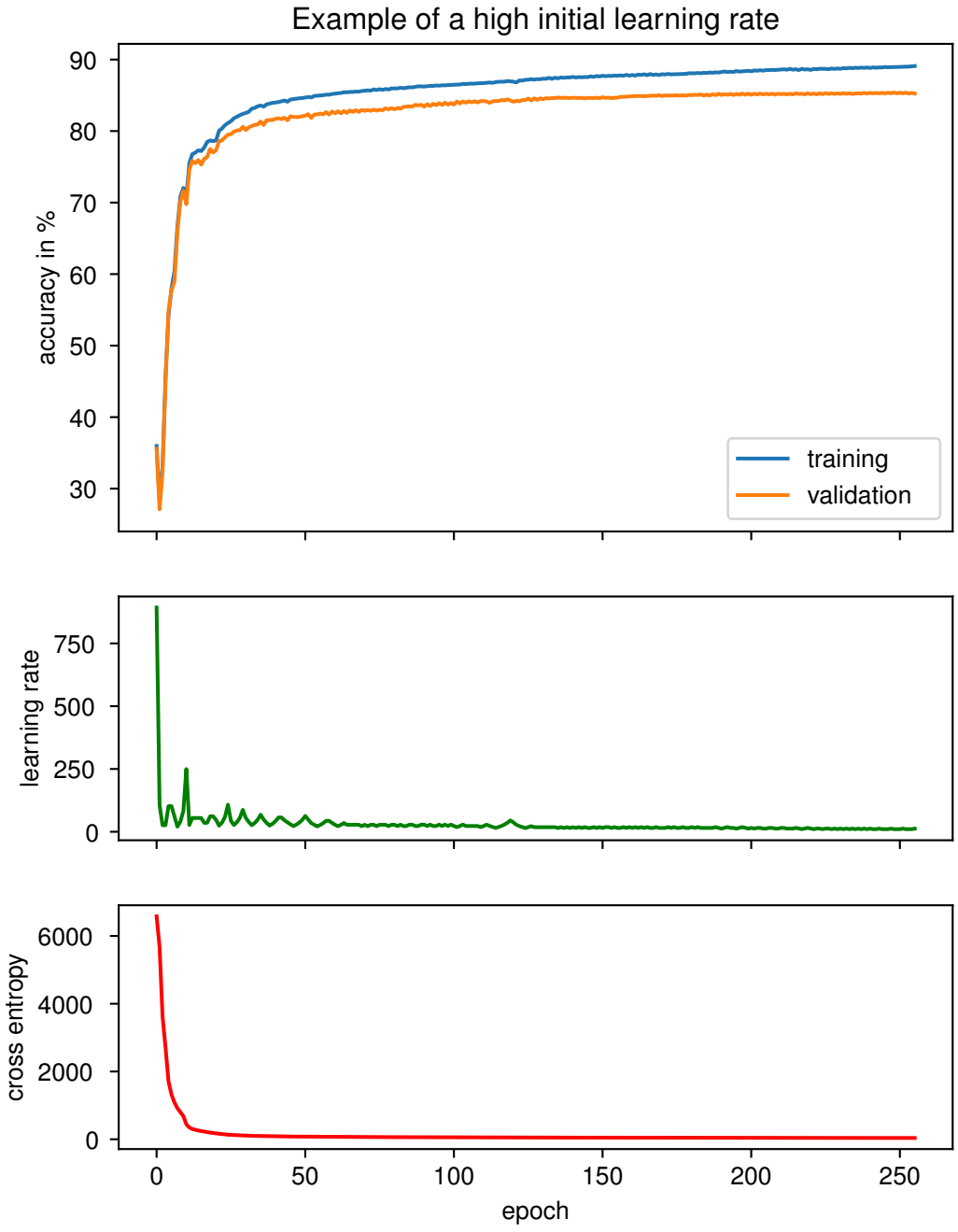


Figure 5.8: This plot shows the accuracies, learning rate and cross entropy development of the training without adjusting the initial learning rate. An initial learning rate of 10 000 was specified. Even though the learning rate decreased rapidly in the first few steps it never went under the value of 10.

Chapter 6

Results

The re-training process has been performed repeatedly on the Food-11 dataset with alternating parameters to analyse the capabilities of the proposed algorithm.

The development of training and validation accuracy, learning rate and cross entropy throughout a training is shown in figure 6.1. After 530 epochs the training ended due to the stopping criterion. An evaluation accuracy of 89.4% was achieved.

In their paper [36] Singla et al. used the pre-trained DCNN GoogLeNet [37] and fine-tuned the last 6 layers to the Food-11 dataset. They reported an accuracy of 83.5%. This means my approach gains almost 6% over the authors method. For an image classification problem this increase is quite noticeable.

6.1 Confusions

Since not all of the images are classified correctly it can be interesting to study the misclassifications. The confusion matrix in table 6.1 shows the distribution of predictions of a certain class. Several observations can be made using the confusion matrix.

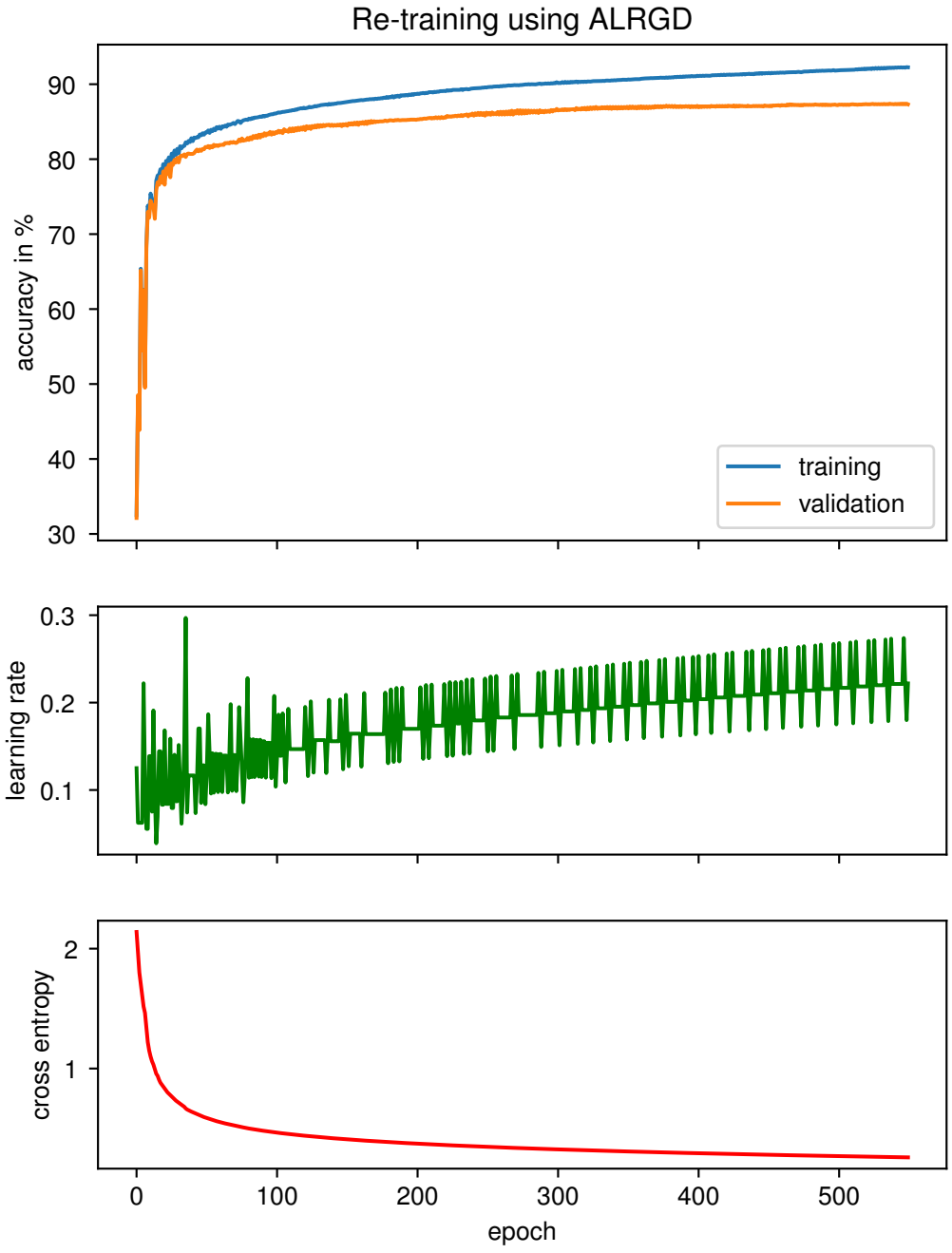


Figure 6.1: This plot shows a typical training progress. The final evaluation accuracy of this specific run is 89.4%

		Prediction										
		dairy	seafood	meat	eggs	noodles	bread	dessert	rice	soup	fruit&veg	fried
Actual	dairy	74.3	5.4	0.7	1.4	0.0	1.4	12.2	0.0	1.4	0.7	2.7
	seafood	0.0	89.4	1.7	2.0	0.0	1.3	3.6	0.3	0.3	1.0	0.3
	meat	0.0	1.2	91.4	1.9	0.0	1.2	1.6	0.0	0.5	0.2	2.1
	eggs	0.6	1.5	1.2	85.7	0.0	5.4	3.0	0.0	0.3	0.0	2.4
	noodles	0.0	0.0	0.0	0.0	97.3	0.0	0.0	0.0	2.0	0.0	0.7
	bread	0.8	0.5	1.1	6.2	0.0	84.8	2.2	0.3	0.0	0.3	3.8
	dessert	1.8	2.8	2.0	2.8	0.0	1.0	87.6	0.0	1.0	0.4	0.6
	rice	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0
	soup	0.0	1.4	0.2	0.2	0.0	0.6	0.8	0.0	96.8	0.0	0.0
	fruit&veg	0.4	1.7	0.9	1.3	0.0	0.0	2.2	0.0	0.0	93.5	0.0
	fried	0.3	1.0	5.6	3.1	0.0	2.1	4.2	0.3	0.0	0.0	83.3

Table 6.1: The confusion matrix. A row represents the actual class (truth) and a column the prediction (classification by the algorithm). All values are given in percent.

The dessert class is a popular target class. Especially dairy products are often classified as desserts but also most other classes have some images misclassified as desserts. My hypothesis that would explain this is that dessert images are very diverse and because of that the system has a hard time finding features that all the images have in common. The result is that the class could become the *no classification* class. Since there is no not-food class all of the unrecognised images might be classified as desserts. Furthermore, along with the soup class the dessert class is the biggest class containing 2500 images. This increases the pure statistical likelihood that an image is classified as dessert. The reason why this is not a problem for the soup class is that the soup class has very distinct features: circles and eclipses filled with a constant colour.

Another observation that can be made using the confusion matrix is that some pairs of classes are often confused with each other. One such pair is meat and fried food. This is because some meat dishes have the same colour as the fired food and also the side dishes sometimes match. Another pair is bread and eggs. I would also contribute this confusion to the similar colours and the fact the eggs are very often served on bread.

The rice class is predicted best. With only 472 images rice is the smallest class in the dataset but the high classification accuracy is due to the specific texture of rice. The same argument can be made for noodles even though some are confused with soup. This is partly because noodles on the plate are very round (which is a feature of soups) and partly because some soups contain noodles.

Figure 6.2 shows images of the dataset that were misclassified along with images from the class they were misclassified as that look alike.

(a) dairy products classified as desserts

(b) bread classified as eggs

(c) fried food classified as meat

(d) noodles classified as soup

Figure 6.2: Some examples of misclassified images (left) that look similar to an image of the predicted class (right). Those are the top 3 most misclassified classes (a-c) and an example of noodles being predicted as soup which shows the network learned that soups have round features.

6.2 Training Times

An interesting criterion for algorithm comparison is the execution time. An execution of the ALRGD algorithm usually takes between 5 and 8 hours on the system specified in section 4.2. Table 6.2 shows the training times for three different optimization algorithms. It can be observed that my algorithm requires three times the amount of time for the same problem size. This is because it performs the training step three times: one per learning rate for same, high and low η.

Algorithm	**Execution time (min)**
SGD	6
AdaDelta	6
ALRGD	18

Table 6.2: A comparison of execution times to calculate 20 iterations on the Food-11 dataset.

Those are the execution times provided the bottlenecks were already created. The bottleneck creation phase takes approximately half the time of a re-training process. On my system the bottleneck creation for the Food-11 dataset took approximately 3 hours. This is a throughput of 2 images per second.

6.3 ALRGD compared to SDG

In this section the adaptive learning rate algorithm is compared to gradient descent with a constant learning rate.

I tested both algorithms on the Food-11 dataset using 200 training steps. The results are shown in figure 6.3. It can be observed that ALRGD performs better in the beginning but at step 75 SGD managed to catch up. In the end both algorithms reached similar evaluation accuracies: 87.4% (ALRGD) and 87.7% (SGD). So there is no accuracy gain

when using the adaptive learning rate. However, it has to be mentioned that the constant learning rate of 0.16 for SGD was discovered using ALRGD on the same problem (and observing which learning rate it converges to). Without the adaptive learning rate several trainings with different learning rates would have to be performed to achieve the same result.

6.4 ALRGD compared to AdaDelta

In this section my ALRGD is compared to AdaDelta [42], another algorithm which uses an adaptive learning rate.

Figure 6.4 shows the comparison on the Food-11 dataset with 500 training steps. ALRGD can compete with AdaDelta. It even performs slightly better in the beginning. The final evaluation accuracies are in the same narrow range: 89.3% (ALRGD) and 89.1% (AdaDelta).

Considering the execution times AdaDelta should be the preferred algorithm since it can produce a similar result in a third of the time. Nonetheless, my algorithm also has its advantages. When an appropriate learning rate is unknown ALRGD is the safer choice. To demonstrate this trainings have been performed with very low and very high initial learning rates. An initial learning rate of 0.001 was applied and the results are shown in figure 6.5. My algorithm starts a bit slower than usual (when given a normal initial η) but after a few iterations it found a good learning rate of around 0.2 providing an acceptable end result (87.3%). AdaDelta however fails to adapt quickly enough and stays at low accuracies (18.9% in the end).

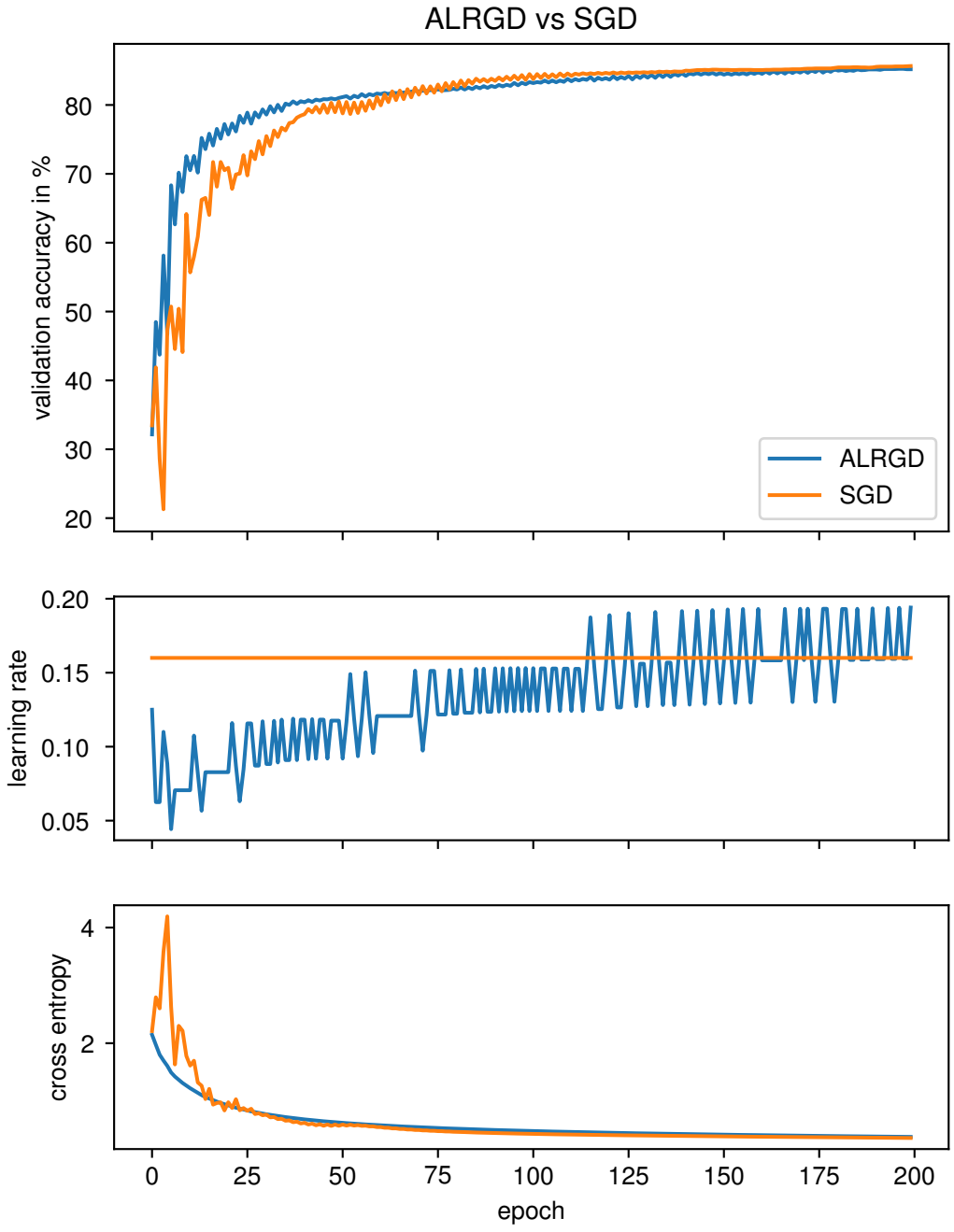


Figure 6.3: This plot compares the ALRGD algorithm (blue) to gradient descent (orange). It shows validation accuracy, learning rate and cross entropy development for both trainings. Gradient descent used a constant learning rate of 0.16

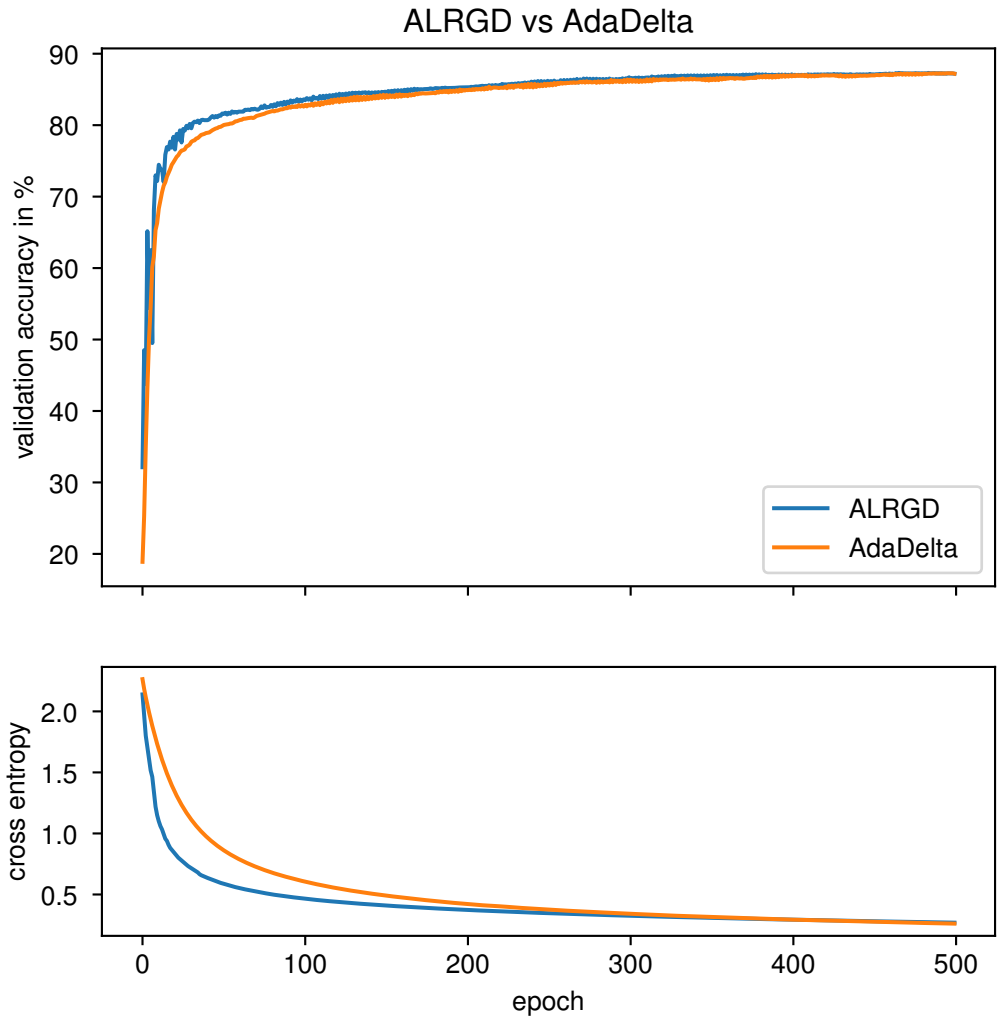


Figure 6.4: This plot compares the ALRGD algorithm (blue) to AdaDelta (orange). It shows validation accuracy and cross entropy development for both trainings. Both algorithms were provided with an initial learning rate of 1.0

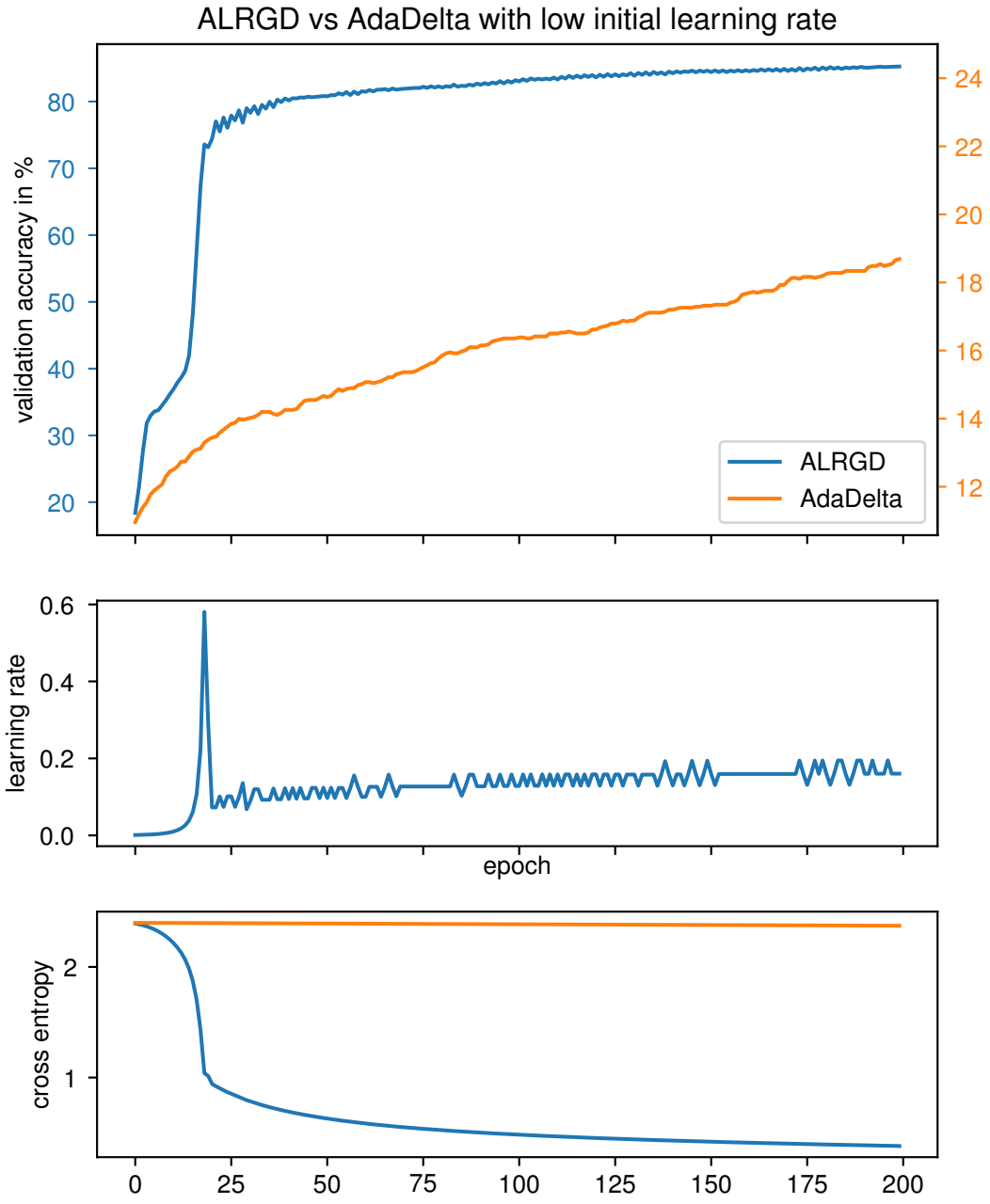


Figure 6.5: This plot compares the ALRGD algorithm (blue) to AdaDelta (orange) using an initial learning rate of 0.001. Note that in the accuracy plot the y-axis has two different scales. The learning rate of ALRGD overshoots but manages to settle in the usual area. There is no learning rate data available for AdaDelta.

Since my algorithm goes through the initial adaption phase described in section 5.6 it is not susceptible to too high initial learning rates. The console output of my implementation for an execution with an initial learning rate of 100 might look as follows

```
2017-08-15 14:35:47: Same LR Train accuracy = 32.6%, cross entropy = 770.259827
2017-08-15 14:36:04: LR = 50.000000, Train accuracy = 32.6%, cross entropy = 385.131683
2017-08-15 14:36:21: LR = 25.000000, Train accuracy = 32.6%, cross entropy = 192.569748
2017-08-15 14:36:39: LR = 12.500000, Train accuracy = 32.6%, cross entropy = 96.293030
2017-08-15 14:36:56: LR = 6.250000, Train accuracy = 32.6%, cross entropy = 48.163162
2017-08-15 14:37:14: LR = 3.125000, Train accuracy = 32.6%, cross entropy = 24.115374
2017-08-15 14:37:34: LR = 1.562500, Train accuracy = 32.6%, cross entropy = 12.127821
2017-08-15 14:37:50: LR = 0.781250, Train accuracy = 32.6%, cross entropy = 6.213081
2017-08-15 14:38:06: LR = 0.390625, Train accuracy = 32.6%, cross entropy = 3.427958
2017-08-15 14:38:23: LR = 0.195312, Train accuracy = 32.6%, cross entropy = 2.345481
2017-08-15 14:38:40: LR = 0.097656, Train accuracy = 32.6%, cross entropy = 2.123763
2017-08-15 14:38:56: LR = 0.048828, Train accuracy = 32.7%, cross entropy = 2.183693
2017-08-15 14:38:56: proceeding with LR = 0.097656
```

The learning rate is halved until a minimal cross entropy of 2.12 is reached. At this point the algorithm continues as if it was provided with an initial learning rate of 0.098. This console output also shows why the approach based on train accuracy did not work: train accuracy can increase even though cross entropy increases.

A comparison of ALRGD and AdaDelta on a learning rate of 100 can be found in figure 6.6. It shows that my algorithm performs as expected but AdaDelta has huge fluctuations in accuracy. After 200 steps it reached 84.2% which might sound reasonable but this value could drop if more iterations are performed. ALRGD on the other hand ended with a stable 87.4%.

6.5 ALRGD on other Datasets

I tested my algorithm on datasets other than the Food-11. This was done in order to ensure that it is general enough and not to require this specific dataset. None of the algorithm parameters were changes for the trainings in this section. Only the dataset was swapped.

Table 6.3 shows the accuracies obtained when using the UNICT-FD1200 dataset. Each row represents a dataset and each column an algorithm. In their paper Farinella et al. use a

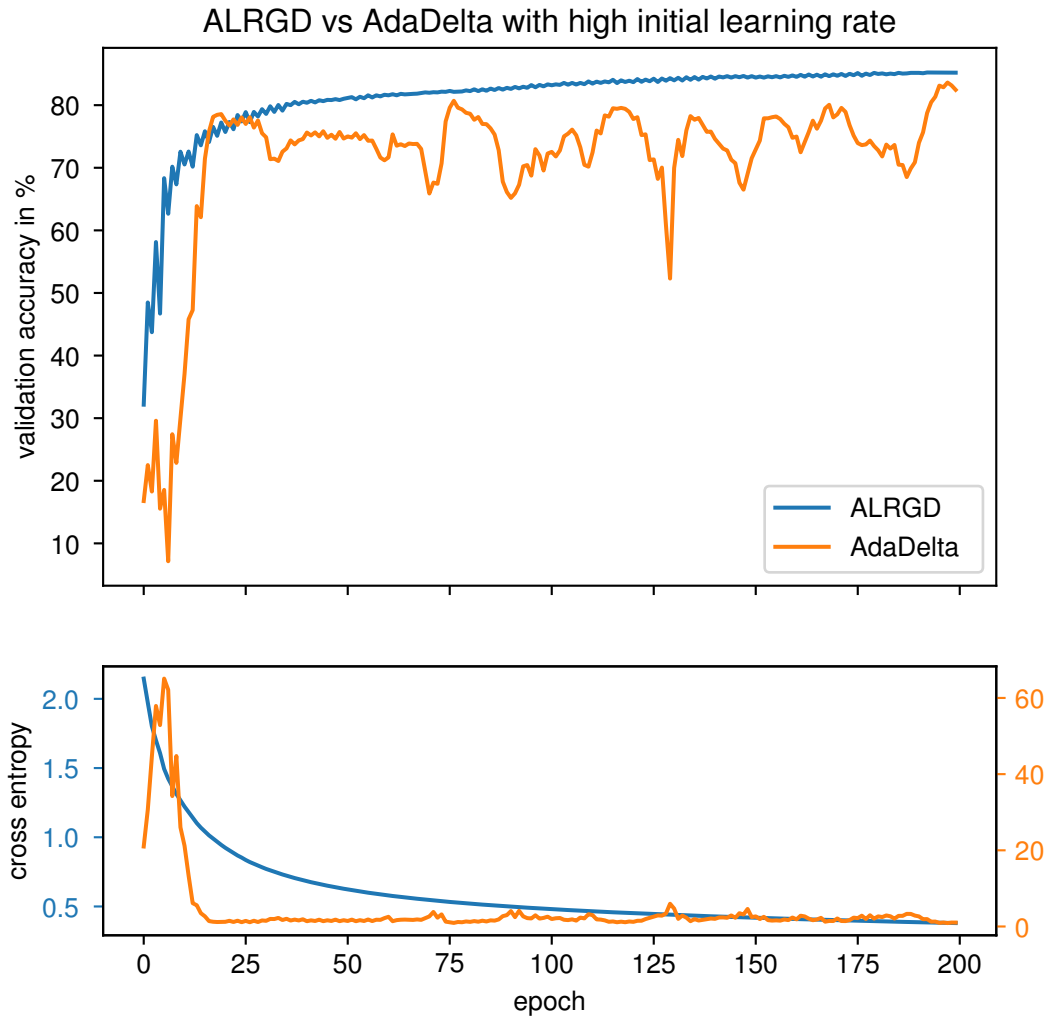


Figure 6.6: This plot compares the ALRGD algorithm (blue) to AdaDelta (orange) using an initial learning rate of 100. Note that in the cross entropy plot the y-axis has two different scales.

bag of textons approach an reached accuracies of over 90%. My algorithm only managed 75% but it is still an improvement over the CNN approach that they compared against in their paper which only reached 51%. The reason why CNN approaches (including mine) do not work well on this dataset is the number of images. There are only 4754

images in the dataset in originally 1200 classes. Those classes have been combined into 8 broader classes: Appetizer, Main Course, Second Course, Single Course, Side Dish, Dessert, Breakfast and Fruit. Those are types of dishes and not necessarily types of food which might also have influenced the result.

Dataset	ALRGD	CNN in [36]	CNN in [14]	BoT in [14]
Food-11	**89.4%**	83.5%	-	-
UNICT-FD1200	**74.9%**	-	51.4%	93.0%

Table 6.3: A comparison of accuracies by different methods on the Food-11 and UNICT-FD1200 datasets.

Three more datasets have been tested, namely UECFOOD-100, UECFOOD-256 and ETH Food-101 (the number of classes are indicated after the dash). The results are shown in table 6.4. It can be notices that my approach only manages to outperform the baseline methods that do not use DCNNs. There is still a 5% gap to DCNN strategies from 2015 and a significant difference to current state-of-the-art techniques. There are two reasons why those other approaches outperform mine. Either they retrained more layers for classification or they performed some pre-processing on the images. Both of those strategies are more computationally expensive. However, I do consider them as future work.

Dataset	ALRGD	FV in [23]	CNN in [40]	RFDC in [5]	CNN in [18]
UECFOOD-100	**73.8%**	72.3%	78.8%	-	81.4%
UECFOOD-256	**62.2%**	-	67.6%	-	76.2%
ETH Food-101	**52.4%**	-	-	50.8%	88.3%

Table 6.4: A comparison of accuracies by different methods on the UECFOOD-100, UECFOOD-256 and ETH Food-101 datasets.

Chapter 7

Future Work

The algorithm I propose in this thesis has its strengths but also possibilities for improvement. One of its biggest shortcomings is the efficiency. In further development the update step could be reviewed to find a solution that does not require three full trainings. Maybe the approach should even be changed to an incremental update (as in AdaDelta) such that only one training per step is performed.

Another way to improve performance is to change the update intervals. In the current version the learning rate is updated at every step. During the training the updates become less and less important. So to save computation time one could experiment with different update strategies. For instance update for the first $n_0 = 10$ steps and then only every $n_k = 10$ steps (adjust n_0 and n_k to fit the problem). Similarly an update rate that decreases linearly (or even exponentially) over epochs could work as well.

The network architecture of the feature map creation could be changed. There are other pre-trained DCNNs available such as the ResNet and I expect that new ones will be released. The central part for the classification to work are the extracted features. Therefore other DCNNs should be compared against the Inception V3 architecture.

My proposed algorithm does not rely on the underlying back-propagation algorithm.

Thus gradient descent could be substituted with another algorithm that uses a constant or global learning rate. Whether or not this would yield better results is to be examined.

I have chosen a batch-learning approach to ensure a stable learning rate adaptation. However, an approach that comes close to online-learning could be more efficient. Online-learning would use only one image per training but a combination could use a batch size of any number of images of the training set. Further work could explore optimal batch sizes and even the possibility of adapting batch-sizes (changing the batch-size throughout the training).

The choice of dataset mainly depends on the application. Since I created this thesis for research purposes the exact classes did not matter so much. However, when a real life application is developed a good class definition is essential. When no dataset with acceptable classes can be found a new dataset should be created. If every class contains at least 100 images the given classification approach should work.

Another major improvement would be to support multiple classes per image. Therefore a segmentation strategy has to be implemented. This could range from distinguishing between side-dish and main dish on a single plate or detecting and locating multiple food items in an image, to a more fine-grained segmentation that splits the parts of a single food item (such as a burger) into its components (meat, bread, lettuce etc.). A classification can then be applied on the segmented parts. Evidently, the training has to be performed on appropriately segmented samples for this strategy to work.

To improve the accuracy obtained by the system some input distortions could be applied, such as: image cropping, image resizing, changing brightness/contrast/saturation/hue. This should make the classification more reliable. The reason I avoided this is that it significantly increases the training time since bottleneck caching cannot be applied anymore.

Chapter 8

Conclusion

When it comes to image classification features extracted by a DCNN seem to be the way to go. Depending on the problem size the classification can be performed with conventional methods such as SVM or by using a deep neuronal network. DCNNs have proven effective especially for large datasets containing several thousand images.

The task of food image classification is not trivial since food images can be very diverse. On a classification problem with 11 classes my approach achieves an accuracy of 89.4% which is a 7% increase to a similar method published by the authors of the dataset in 2016. However, my approach only re-trains the last layer of a DCNN and cannot compete with state-of-the art solutions that re-train several layers.

Adaptive learning rates overcome the issue of finding a good learning rate. Thus they can save time and effort. The algorithm I proposed looks promising as it matches the accuracies of similar algorithms such as AdaDelta. Some more improvements have to be made for it to be more efficient.

In the current state the ALRGD algorithm can be used to find an appropriate learning rate for a problem. Afterwards the training can be performed using plain gradient descent with that learning rate on a simulation with more training steps.

Since training and classification process can be split, the computationally heavy training can be performed on a computer or even an HPC system. The simple classification can then be performed on resource scarce systems such as mobile phones.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] M. M. Anthimopoulos, L. Gianola, L. Scarnato, P. Diem, and S. G. Mougiakakou. A food recognition system for diabetic patients based on an optimized bag-of-features model. *IEEE Journal of Biomedical and Health Informatics*, 18(4):1261–1271, July 2014.

[3] Raymond C. Baker and Daniel S. Kirschenbaum. Self-monitoring may be necessary for successful weight control. *Behavior Therapy*, 24(3):377 – 394, 1993.

[4] J.E. Blundell, J.L. Baker, E. Boyland, E. Blaak, J. Charzewska, S. de Henauw, G. Frühbeck, M. Gonzalez-Gross, J. Hebebrand, L. Holm, V. Kriaucioniene, L. Liss-

ner, J.-M. Oppert, K. Schindler, A.M. Silva, and E. Woodward. Variations in the prevalence of obesity among european countries, and a consideration of possible causes. *Obes Facts*, 10(1):25 –37, 2017.

[5] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101 – mining discriminative components with random forests. In *European Conference on Computer Vision*, 2014.

[6] Lora E. Burke, Jing Wang, and Mary Ann Sevick. Self-monitoring in weight loss: A systematic review of the literature. *Journal of the American Dietetic Association*, 111(1):92 – 102, 2011.

[7] Callistus Bvenura and Dharini Sivakumar. The role of wild fruits and vegetables in delivering a balanced and healthy diet. *Food Research International*, 99:15 – 30, 2017.

[8] M. Chen, K. Dhingra, W. Wu, L. Yang, R. Sukthankar, and J. Yang. Pfid: Pittsburgh fast-food image dataset. In *2009 16th IEEE International Conference on Image Processing (ICIP)*, pages 289–292, Nov 2009.

[9] Davide Chicco, Peter Sadowski, and Pierre Baldi. Deep autoencoder neural networks for gene ontology annotation predictions. In *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, BCB '14, pages 533–540, New York, NY, USA, 2014. ACM.

[10] Stergios Christodoulidis, Marios Anthimopoulos, and Stavroula Mougiakakou. *Food Recognition for Dietary Assessment Using Deep Convolutional Neural Networks*, pages 458–465. Springer International Publishing, Cham, 2015.

[11] Yann N. Dauphin, Harm de Vries, Junyoung Chung, and Yoshua Bengio. Rmsprop and equilibrated adaptive learning rates for non-convex optimization. *CoRR*, abs/1502.04390, 2015.

[12] L. B. Dixon, F. J. Cronin, and S. M. Krebs-Smith. Let the pyramid guide your food choices: capturing the total diet concept. *J. Nutr.*, 131(2S-1):461S–472S, Feb 2001.

[13] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[14] Giovanni Maria Farinella, Dario Allegra, Marco Moltisanti, Filippo Stanco, and Sebastiano Battiato. Retrieval and classification of food images. *Computers in Biology and Medicine*, 77:23 – 39, 2016.

[15] Giovanni Maria Farinella, Dario Allegra, and Filippo Stanco. *A Benchmark Dataset to Study the Representation of Food Images*, pages 584–599. Springer International Publishing, Cham, 2015.

[16] Muhammad Farooq and Edward Sazonov. *Feature Extraction Using Deep Learning for Food Type Recognition*, pages 464–472. Springer International Publishing, Cham, 2017.

[17] David W Haslam and W Philip T James. Obesity. *The Lancet*, 366(9492):1197 – 1209, 2005.

[18] Hamid Hassannejad, Guido Matrella, Paolo Ciampolini, Ilaria De Munari, Monica Mordonini, and Stefano Cagnoni. Food image recognition using very deep convolutional networks. In *Proceedings of the 2Nd International Workshop on Multimedia*

Assisted Dietary Management, MADiMa '16, pages 41–49, New York, NY, USA, 2016. ACM.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[20] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012.

[21] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, May 2007.

[22] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[23] Yoshiyuki Kawano and Keiji Yanai. Food image recognition with deep convolutional features. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, UbiComp '14 Adjunct, pages 589–593, New York, NY, USA, 2014. ACM.

[24] Yoshiyuki Kawano and Keiji Yanai. *Automatic Expansion of a Food Image Dataset Leveraging Existing Categories with Domain Adaptation*, pages 3–17. Springer International Publishing, Cham, 2015.

[25] Flegal KM, Carroll MD, Kit BK, and Ogden CL. Prevalence of obesity and trends in the distribution of body mass index among us adults, 1999-2010. *JAMA*, 307(5):491–497, 2012.

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

[27] Betty W. Li, Karen W. Andrews, and Pamela R. Pehrsson. Individual sugars, soluble, and insoluble dietary fiber contents of 70 high consumption foods. *Journal of Food Composition and Analysis*, 15(6):715 – 723, 2002.

[28] David G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, pages 1150–, Washington, DC, USA, 1999. IEEE Computer Society.

[29] Y. Matsuda, H. Hoashi, and K. Yanai. Recognition of multiple-food images by detecting candidate regions. In *2012 IEEE International Conference on Multimedia and Expo*, pages 25–30, July 2012.

[30] Simon Mezgec and Barbara Koroušić Seljak. Nutrinet: A deep learning food and drink image recognition system for dietary assessment. *Nutrients*, 9 7, 2017.

[31] 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*, 115(3):211–252, Dec 2015.

[32] S. M. Rüger. A class of asymptotically stable algorithms for learning-rate adaptation. *Algorithmica*, 22(1):198–210, Sep 1998.

[33] Ralf Salomon and J. Leo van Hemmen. Accelerating backpropagation through dynamic self-adaptation. *Neural Networks*, 9(4):589 – 601, 1996.

[34] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229, 2013.

[35] B. Singh, S. De, Y. Zhang, T. Goldstein, and G. Taylor. Layer-specific adaptive learning rates for deep networks. pages 364–368, Dec 2015.

[36] Ashutosh Singla, Lin Yuan, and Touradj Ebrahimi. Food/Non-food Image Classification and Food Categorization using Pre-Trained GoogLeNet Model. In *Madima'16: Proceedings Of The 2Nd International Workshop On Multimedia Assisted Dietary Management*, pages 3–11, New York, 2016. Assoc Computing Machinery.

[37] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, June 2015.

[38] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[39] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, June 2014.

[40] K. Yanai and Y. Kawano. Food image recognition using deep convolutional network with pre-training and fine-tuning. pages 1–6, June 2015.

[41] S. Yang, M. Chen, D. Pomerleau, and R. Sukthankar. Food recognition using statistics of pairwise local features. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2249–2256, June 2010.

[42] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

Appendix A

Additional Figures

This appendix contains figures of trainings that were not included in the thesis but might be of interest for further analysis.

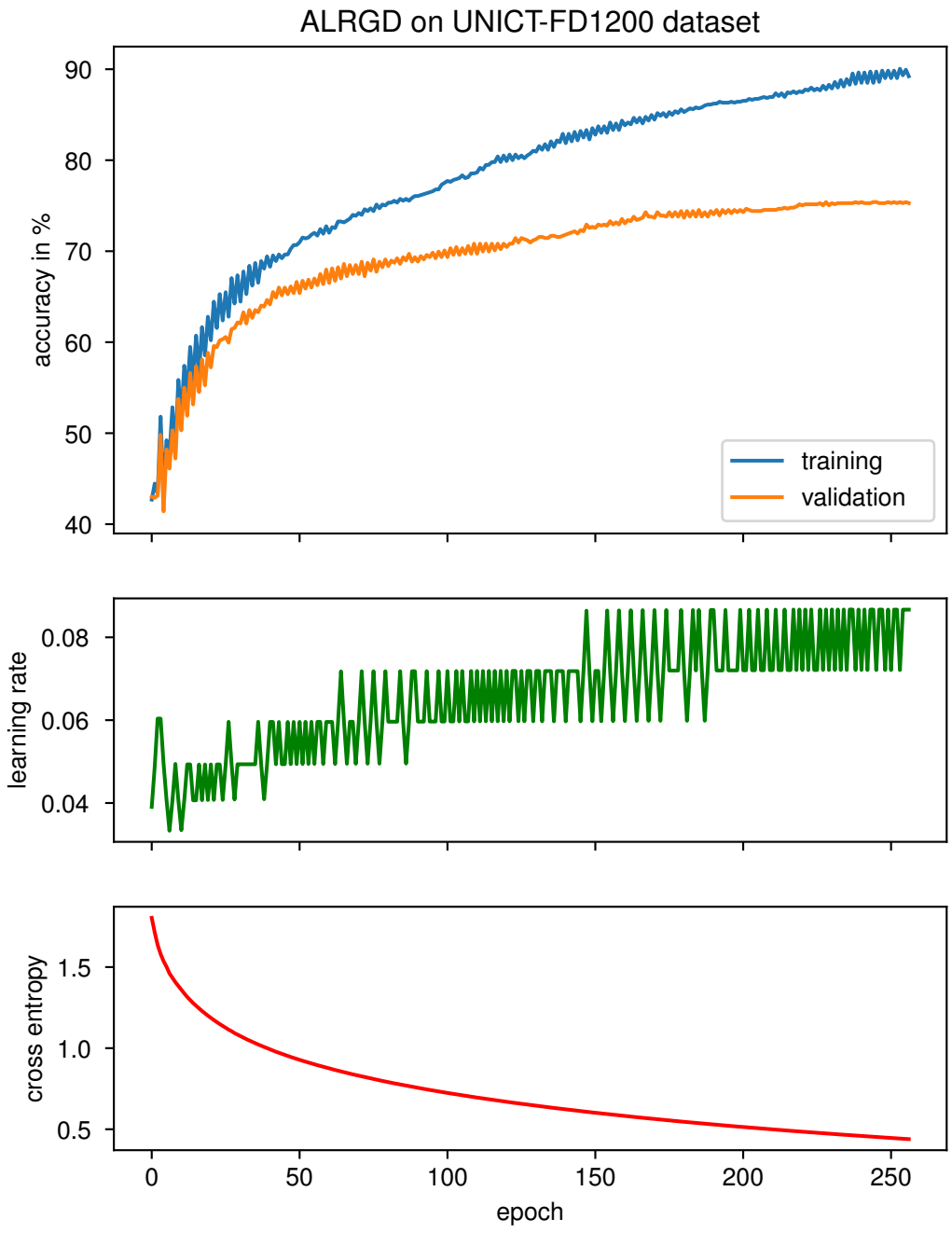


Figure A.1: This plot compares shows the training and validation accuracy, learning rate and cross entropy of a training on the UNICT-FD1200 dataset. This is very much comparable to a training on the Food-11 except that the learning rate is lower.

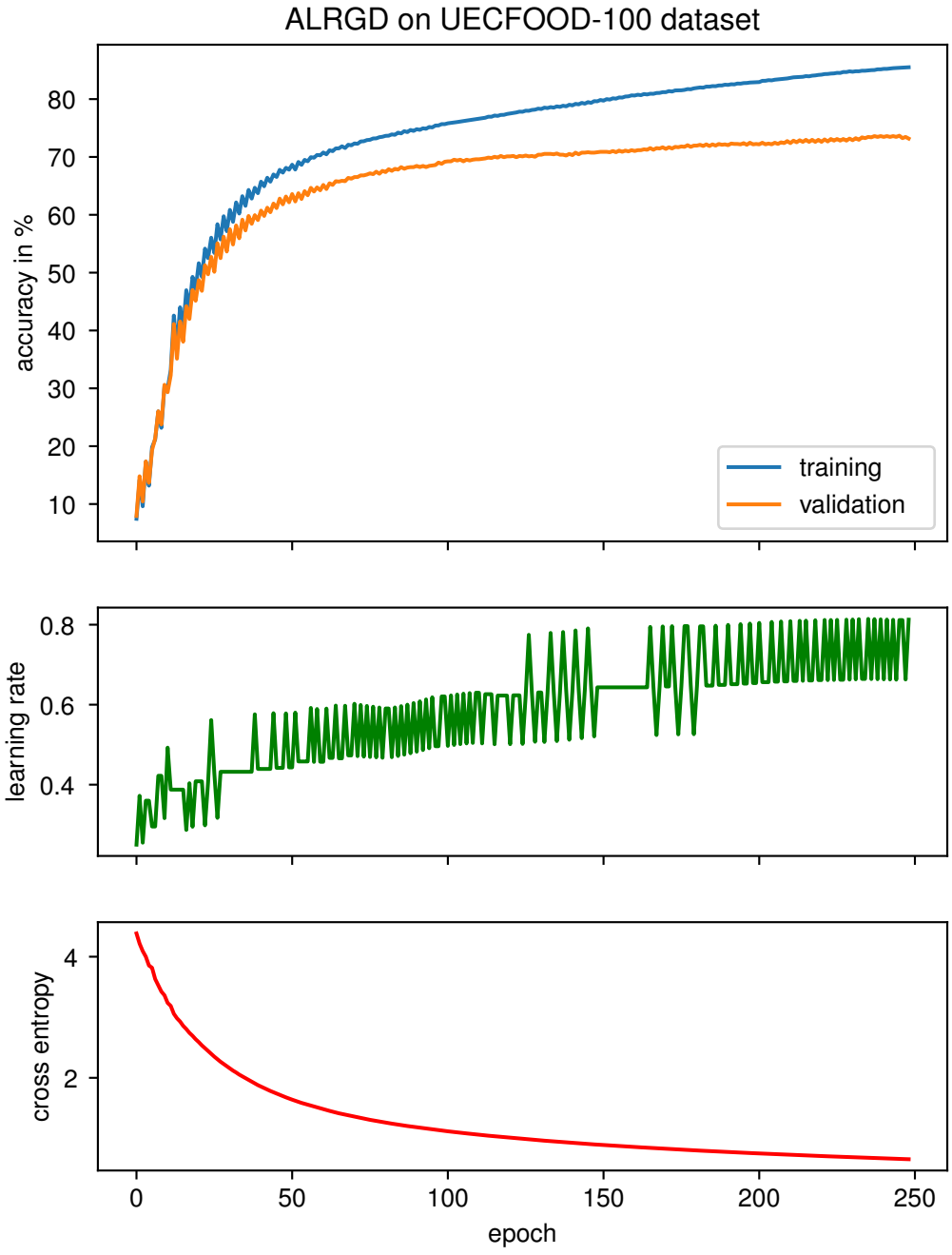


Figure A.2: This plot compares shows the training and validation accuracy, learning rate and cross entropy of a training on the UECFOOD-100 dataset. The final accuracy is 73.8%.

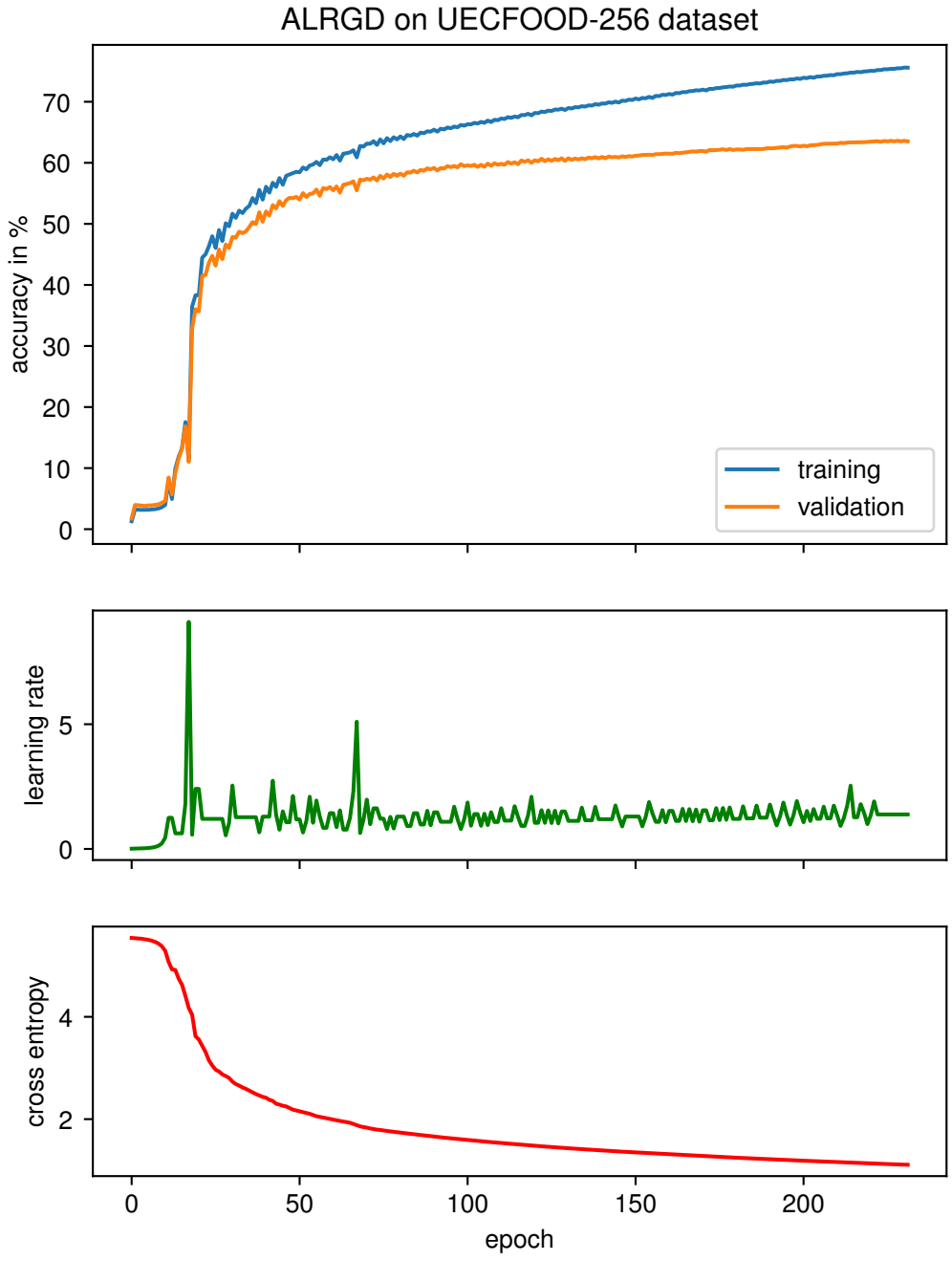


Figure A.3: This plot compares shows the training and validation accuracy, learning rate and cross entropy of a training on the UECFOOD-256 dataset. The initial learning rate was set too low for this dataset (0.01) that is why the beginning is not as steep. Still, a final accuracy of 62.2% has been achieved.

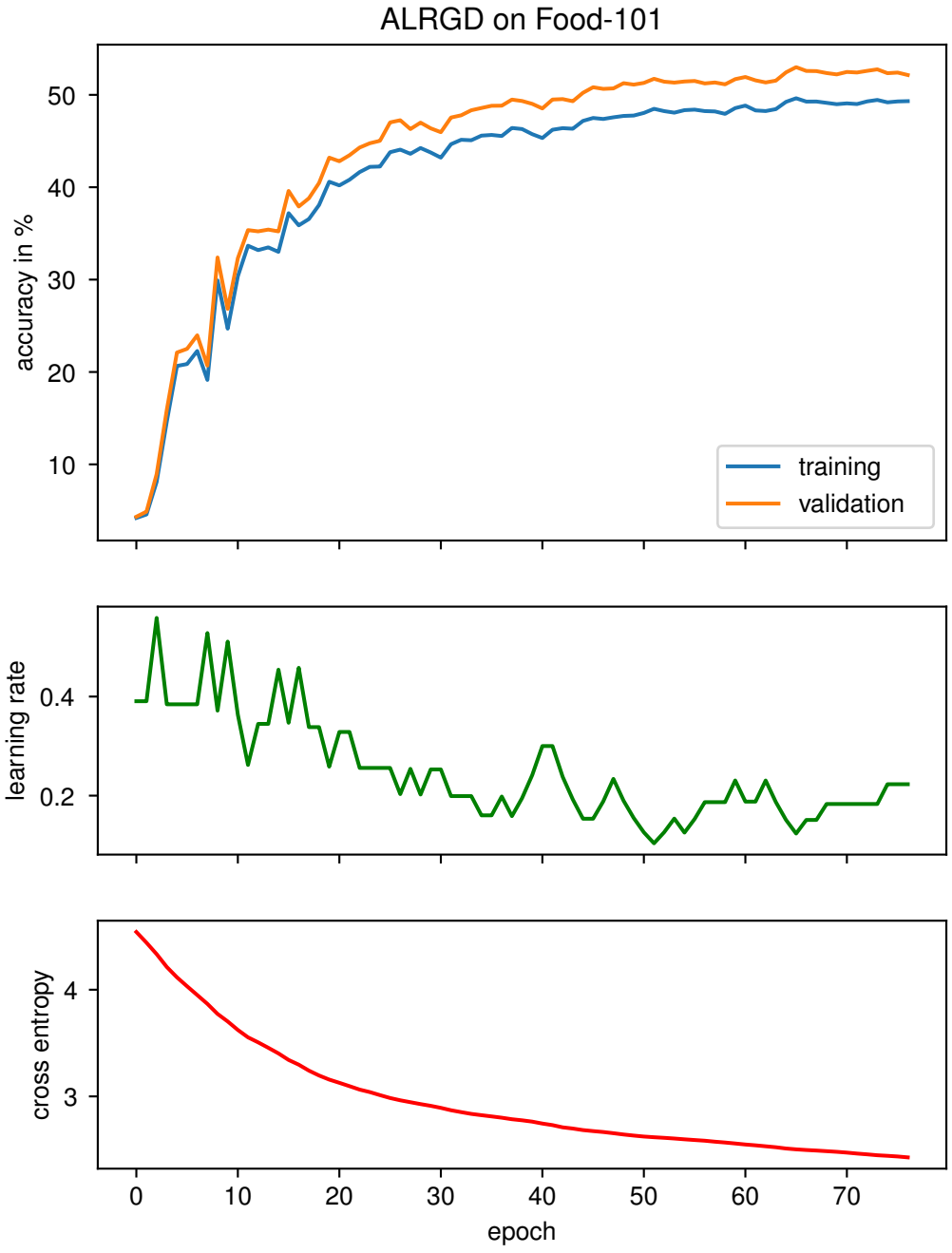


Figure A.4: This plot compares shows the training and validation accuracy, learning rate and cross entropy of a training on the ETH FOOD-101 dataset. The final accuracy is 52.4%.

Appendix B

Python Code

My implementation is based on TensorFlow example code[1]. The modified code can be found on GitHub[2] as well as attached below.

[1] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/image_retraining/retrain.py

[2] https://github.com/aschmidhofer/cranfieldthesis/blob/master/src/retrain.py

```
# Copyright 2015 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================
#
# This file was modified by Andreas Schmidhofer, July 2017
#
# (most modification contain a comment tagged with AS)
#
# ==============================================================================

"""Simple transfer learning with an Inception v3 architecture model.

With support for TensorBoard.

This example shows how to take a Inception v3 architecture model trained on
ImageNet images, and train a new top layer that can recognize other classes of
images.

The top layer receives as input a 2048-dimensional vector for each image. We
train a softmax layer on top of this representation. Assuming the softmax layer
contains N labels, this corresponds to learning N + 2048*N model parameters
corresponding to the learned biases and weights.

Here's an example, which assumes you have a folder containing class-named
subfolders, each full of images for each label. The example folder flower_photos
should have a structure like this:

~/flower_photos/daisy/photo1.jpg
~/flower_photos/daisy/photo2.jpg
...
~/flower_photos/rose/anotherphoto77.jpg
...
~/flower_photos/sunflower/somepicture.jpg

The subfolder names are important, since they define what label is applied to
each image, but the filenames themselves don't matter. Once your images are
prepared, you can run the training with a command like this:


```bash
bazel build tensorflow/examples/image_retraining:retrain && \
bazel-bin/tensorflow/examples/image_retraining/retrain \
    --image_dir ~/flower_photos
```

Or, if you have a pip installation of tensorflow, `retrain.py` can be run
without bazel:

```bash
python tensorflow/examples/image_retraining/retrain.py \
    --image_dir ~/flower_photos
```

You can replace the image_dir argument with any folder containing subfolders of
images. The label for each image is taken from the name of the subfolder it's
in.

This produces a new model file that can be loaded and run by any TensorFlow
program, for example the label_image sample code.

To use with TensorBoard:

By default, this script will log summaries to /tmp/retrain_logs directory
```

```
Visualize the summaries with this command:

tensorboard --logdir /tmp/retrain_logs

"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
from datetime import datetime
import hashlib
import os.path
import random
import re
import struct
import sys
import tarfile
import pickle #by AS

import numpy as np
from six.moves import urllib
import tensorflow as tf

from tensorflow.python.framework import graph_util
from tensorflow.python.framework import tensor_shape
from tensorflow.python.platform import gfile
from tensorflow.python.util import compat

FLAGS = None

# These are all parameters that are tied to the particular model architecture
# we're using for Inception v3. These include things like tensor names and their
# sizes. If you want to adapt this script to work with another model, you will
# need to update these to reflect the values in the network you're using.
# pylint: disable=line-too-long
DATA_URL = 'http://download.tensorflow.org/models/image/imagenet/inception
    -2015-12-05.tgz'#'http://download.tensorflow.org/models/image/imagenet/
    inception-v3-2016-03-01.tar.gz'

# pylint: enable=line-too-long
BOTTLENECK_TENSOR_NAME = 'pool_3/_reshape:0'
BOTTLENECK_TENSOR_SIZE = 2048
MODEL_INPUT_WIDTH = 299
MODEL_INPUT_HEIGHT = 299
MODEL_INPUT_DEPTH = 3
JPEG_DATA_TENSOR_NAME = 'DecodeJpeg/contents:0'
RESIZED_INPUT_TENSOR_NAME = 'ResizeBilinear:0'
MAX_NUM_IMAGES_PER_CLASS = 2 ** 27 - 1  # ~134M
MAX_STEPS = 100000
DYNAMIC_STEPS_LOOKBACK = 16 # used by stopping criteria
CHANGING_LEARNING_RATE_ADJUST_PERCENTAGE_MODIFYER = 10.0 # changes the amount of \
    zeta depending on cross entropy difference
CHANGING_LEARNING_RATE_MIN = 1.2 # the static amount of \zeta (must be >= 1.0)
MIN_LEARNING_RATE = 0.001
OPTIMIZERS = ['ALRGD'] # list of optimizers to run

def create_image_lists(image_dir): # note: percentages not supported anymore
  if not gfile.Exists(image_dir):
    print("Image directory '" + image_dir + "' not found.")
    return None
  result = {}
  # AS - this part was changed to support a directory per category
  for cat in ['training', 'testing', 'validation']:
    category_dir = os.path.join(image_dir,cat)
    if not gfile.Exists(category_dir):
      print("Category directory '" + category_dir + "' not found.")
      return None

    sub_dirs = [x[0] for x in gfile.Walk(category_dir)]
    # The root directory comes first, so skip it.
    is_root_dir = True
    for sub_dir in sub_dirs:
      if is_root_dir:
        is_root_dir = False
        continue
```

```
      extensions = ['jpg', 'jpeg', 'JPG', 'JPEG']
      file_list = []
      dir_name = os.path.basename(sub_dir)
      if dir_name == category_dir:
        continue
      print("Looking for images in '" + dir_name + "'")
      for extension in extensions:
        file_glob = os.path.join(category_dir, dir_name, '*.' + extension)
        file_list.extend(gfile.Glob(file_glob))
      if not file_list:
        print('No files found')
        continue
      if len(file_list) < 20:
        print('WARNING: Folder has less than 20 images, which may cause issues.')
      elif len(file_list) > MAX_NUM_IMAGES_PER_CLASS:
        print('WARNING: Folder {} has more than {} images. Some images will '
              'never be selected.'.format(dir_name, MAX_NUM_IMAGES_PER_CLASS))
      label_name = re.sub(r'[^a-z0-9]+', ' ', dir_name.lower())
      those_images = []
      for file_name in file_list:
        base_name = os.path.basename(file_name)
        those_images.append(base_name)

      if not(label_name in result):
        result[label_name] = {
          'dir': dir_name,
          'training': [],
          'testing': [],
          'validation': [],
        }
      result[label_name][cat] = those_images
  return result


def get_image_path(image_lists, label_name, index, image_dir, category):
  if label_name not in image_lists:
    tf.logging.fatal('Label does not exist %s.', label_name)
  label_lists = image_lists[label_name]
  if category not in label_lists:
    tf.logging.fatal('Category does not exist %s.', category)
  category_list = label_lists[category]
  if not category_list:
    tf.logging.fatal('Label %s has no images in the category %s.',
                     label_name, category)
  mod_index = index % len(category_list)
  base_name = category_list[mod_index]
  sub_dir = os.path.join(category, label_lists['dir']) # AS changed this line to
      support a directory per category
  full_path = os.path.join(image_dir, sub_dir, base_name)
  return full_path


def get_bottleneck_path(image_lists, label_name, index, bottleneck_dir,
                        category):
  return get_image_path(image_lists, label_name, index, bottleneck_dir,
                        category) + '.txt'


def create_inception_graph():
  with tf.Graph().as_default() as graph:
    model_filename = os.path.join(
        FLAGS.model_dir, 'classify_image_graph_def.pb')
    with gfile.FastGFile(model_filename, 'rb') as f:
      graph_def = tf.GraphDef()
      graph_def.ParseFromString(f.read())
      bottleneck_tensor, jpeg_data_tensor, resized_input_tensor = (
          tf.import_graph_def(graph_def, name='', return_elements=[
              BOTTLENECK_TENSOR_NAME, JPEG_DATA_TENSOR_NAME,
              RESIZED_INPUT_TENSOR_NAME]))
  return graph, bottleneck_tensor, jpeg_data_tensor, resized_input_tensor


def run_bottleneck_on_image(sess, image_data, image_data_tensor,
                            bottleneck_tensor):
  bottleneck_values = sess.run(
      bottleneck_tensor,
```

```
      {image_data_tensor: image_data})
  bottleneck_values = np.squeeze(bottleneck_values)
  return bottleneck_values


def maybe_download_and_extract():
  dest_directory = FLAGS.model_dir
  if not os.path.exists(dest_directory):
    os.makedirs(dest_directory)
  filename = DATA_URL.split('/')[-1]
  filepath = os.path.join(dest_directory, filename)
  if not os.path.exists(filepath):

    def _progress(count, block_size, total_size):
      sys.stdout.write('\r>> Downloading %s %.1f%%' %
                       (filename,
                        float(count * block_size) / float(total_size) * 100.0))
      sys.stdout.flush()

    filepath, _ = urllib.request.urlretrieve(DATA_URL,
                                             filepath,
                                             _progress)
    print()
    statinfo = os.stat(filepath)
    print('Successfully downloaded', filename, statinfo.st_size, 'bytes.')
  tarfile.open(filepath, 'r:gz').extractall(dest_directory)


def ensure_dir_exists(dir_name):
  if not os.path.exists(dir_name):
    os.makedirs(dir_name)


def write_list_of_floats_to_file(list_of_floats, file_path):
  s = struct.pack('d' * BOTTLENECK_TENSOR_SIZE, *list_of_floats)
  with open(file_path, 'wb') as f:
    f.write(s)


def read_list_of_floats_from_file(file_path):

  with open(file_path, 'rb') as f:
    s = struct.unpack('d' * BOTTLENECK_TENSOR_SIZE, f.read())
    return list(s)


bottleneck_path_2_bottleneck_values = {}


def create_bottleneck_file(bottleneck_path, image_lists, label_name, index,
                           image_dir, category, sess, jpeg_data_tensor,
                           bottleneck_tensor):
  """Create a single bottleneck file."""
  print('Creating bottleneck at ' + bottleneck_path)
  image_path = get_image_path(image_lists, label_name, index,
                              image_dir, category)
  if not gfile.Exists(image_path):
    tf.logging.fatal('File does not exist %s', image_path)
  image_data = gfile.FastGFile(image_path, 'rb').read()
  try:
    bottleneck_values = run_bottleneck_on_image(
        sess, image_data, jpeg_data_tensor, bottleneck_tensor)
  except:
    raise RuntimeError('Error during processing file %s' % image_path)

  bottleneck_string = ','.join(str(x) for x in bottleneck_values)
  with open(bottleneck_path, 'w') as bottleneck_file:
    bottleneck_file.write(bottleneck_string)


def get_or_create_bottleneck(sess, image_lists, label_name, index, image_dir,
                             category, bottleneck_dir, jpeg_data_tensor,
                             bottleneck_tensor):
  label_lists = image_lists[label_name]
  sub_dir = os.path.join(category,label_lists['dir']) # AS changed to support dir
      per cat
```

```
  sub_dir_path = os.path.join(bottleneck_dir, sub_dir)
  ensure_dir_exists(sub_dir_path)
  bottleneck_path = get_bottleneck_path(image_lists, label_name, index,
                                        bottleneck_dir, category)
  if not os.path.exists(bottleneck_path):
    create_bottleneck_file(bottleneck_path, image_lists, label_name, index,
                           image_dir, category, sess, jpeg_data_tensor,
                           bottleneck_tensor)
  with open(bottleneck_path, 'r') as bottleneck_file:
    bottleneck_string = bottleneck_file.read()
  did_hit_error = False
  try:
    bottleneck_values = [float(x) for x in bottleneck_string.split(',')]
  except ValueError:
    print('Invalid float found, recreating bottleneck')
    did_hit_error = True
  if did_hit_error:
    create_bottleneck_file(bottleneck_path, image_lists, label_name, index,
                           image_dir, category, sess, jpeg_data_tensor,
                           bottleneck_tensor)
    with open(bottleneck_path, 'r') as bottleneck_file:
      bottleneck_string = bottleneck_file.read()
    # Allow exceptions to propagate here, since they shouldn't happen after a
    # fresh creation
    bottleneck_values = [float(x) for x in bottleneck_string.split(',')]
  return bottleneck_values


def cache_bottlenecks(sess, image_lists, image_dir, bottleneck_dir,
                      jpeg_data_tensor, bottleneck_tensor):
  how_many_bottlenecks = 0
  ensure_dir_exists(bottleneck_dir)
  for label_name, label_lists in image_lists.items():
    for category in ['training', 'testing', 'validation']:
      category_list = label_lists[category]
      for index, unused_base_name in enumerate(category_list):
        get_or_create_bottleneck(sess, image_lists, label_name, index,
                                 image_dir, category, bottleneck_dir,
                                 jpeg_data_tensor, bottleneck_tensor)

        how_many_bottlenecks += 1
        if how_many_bottlenecks % 100 == 0:
          print(str(how_many_bottlenecks) + ' bottleneck files created.')


def get_random_cached_bottlenecks(sess, image_lists, how_many, category,
                                  bottleneck_dir, image_dir, jpeg_data_tensor,
                                  bottleneck_tensor):
  class_count = len(image_lists.keys())
  bottlenecks = []
  ground_truths = []
  filenames = []
  if how_many >= 0:
    # Retrieve a random sample of bottlenecks.
    for unused_i in range(how_many):
      label_index = random.randrange(class_count)
      label_name = list(image_lists.keys())[label_index]
      image_index = random.randrange(MAX_NUM_IMAGES_PER_CLASS + 1)
      image_name = get_image_path(image_lists, label_name, image_index,
                                  image_dir, category)
      bottleneck = get_or_create_bottleneck(sess, image_lists, label_name,
                                            image_index, image_dir, category,
                                            bottleneck_dir, jpeg_data_tensor,
                                            bottleneck_tensor)
      ground_truth = np.zeros(class_count, dtype=np.float32)
      ground_truth[label_index] = 1.0
      bottlenecks.append(bottleneck)
      ground_truths.append(ground_truth)
      filenames.append(image_name)
  else:
    # Retrieve all bottlenecks.
    for label_index, label_name in enumerate(image_lists.keys()):
      for image_index, image_name in enumerate(
          image_lists[label_name][category]):
        image_name = get_image_path(image_lists, label_name, image_index,
                                    image_dir, category)
```

```
        bottleneck = get_or_create_bottleneck(sess, image_lists, label_name,
                                              image_index, image_dir, category,
                                              bottleneck_dir, jpeg_data_tensor,
                                              bottleneck_tensor)
        ground_truth = np.zeros(class_count, dtype=np.float32)
        ground_truth[label_index] = 1.0
        bottlenecks.append(bottleneck)
        ground_truths.append(ground_truth)
        filenames.append(image_name)
  return bottlenecks, ground_truths, filenames



def variable_summaries(var):
  """Attach a lot of summaries to a Tensor (for TensorBoard visualization)."""
  with tf.name_scope('summaries'):
    mean = tf.reduce_mean(var)
    tf.summary.scalar('mean', mean)
    with tf.name_scope('stddev'):
      stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
    tf.summary.scalar('stddev', stddev)
    tf.summary.scalar('max', tf.reduce_max(var))
    tf.summary.scalar('min', tf.reduce_min(var))
    tf.summary.histogram('histogram', var)


def add_final_training_ops(class_count, final_tensor_name, bottleneck_tensor,
    optimizer_name, my_learning_rate):
  with tf.name_scope('input'):
    bottleneck_input = tf.placeholder_with_default(
        bottleneck_tensor, shape=[None, BOTTLENECK_TENSOR_SIZE],
        name='BottleneckInputPlaceholder')

    ground_truth_input = tf.placeholder(tf.float32,
                                        [None, class_count],
                                        name='GroundTruthInput')

  # Organizing the following ops as `final_training_ops` so they're easier
  # to see in TensorBoard
  layer_name = 'final_training_ops'
  with tf.name_scope(layer_name):
    with tf.name_scope('weights'):
      initial_value = tf.truncated_normal([BOTTLENECK_TENSOR_SIZE, class_count],
                                          stddev=0.001)

      layer_weights = tf.Variable(initial_value, name='final_weights')

      variable_summaries(layer_weights)
    with tf.name_scope('biases'):
      layer_biases = tf.Variable(tf.zeros([class_count]), name='final_biases')
      variable_summaries(layer_biases)
    with tf.name_scope('Wx_plus_b'):
      logits = tf.matmul(bottleneck_input, layer_weights) + layer_biases
      tf.summary.histogram('pre_activations', logits)

  final_tensor = tf.nn.softmax(logits, name=final_tensor_name)
  tf.summary.histogram('activations', final_tensor)

  with tf.name_scope('cross_entropy'):
    cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
        labels=ground_truth_input, logits=logits)
    with tf.name_scope('total'):
      cross_entropy_mean = tf.reduce_mean(cross_entropy)
  tf.summary.scalar('cross_entropy', cross_entropy_mean)

  with tf.name_scope('train'):
    if(optimizer_name=='ALRGD'):
      optimizer = tf.train.GradientDescentOptimizer(learning_rate=my_learning_rate)
      train_step = optimizer.minimize(cross_entropy_mean)
    if(optimizer_name=='ALRGD_adam'):
      optimizer = tf.train.AdamOptimizer(learning_rate=my_learning_rate)
      train_step = optimizer.minimize(cross_entropy_mean)
    elif(optimizer_name=='gradientdescent'):
      optimizer = tf.train.GradientDescentOptimizer(FLAGS.learning_rate)
      train_step = optimizer.minimize(cross_entropy_mean)
```

```
    elif (optimizer_name=='adagrad'):
      optimizer = tf.train.AdagradOptimizer(FLAGS.learning_rate)
      train_step = optimizer.minimize(cross_entropy_mean)
    elif (optimizer_name=='adam'):
      optimizer = tf.train.AdamOptimizer(FLAGS.learning_rate)
      train_step = optimizer.minimize(cross_entropy_mean)
    elif (optimizer_name=='ftrl'):
      optimizer = tf.train.FtrlOptimizer(FLAGS.learning_rate)
      train_step = optimizer.minimize(cross_entropy_mean)
    elif (optimizer_name=='adadelta'):
      optimizer = tf.train.AdadeltaOptimizer(FLAGS.learning_rate)
      train_step = optimizer.minimize(cross_entropy_mean)


  return (train_step, cross_entropy_mean, bottleneck_input, ground_truth_input,
          final_tensor)


def add_evaluation_step(result_tensor, ground_truth_tensor):
  with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
      prediction = tf.argmax(result_tensor, 1)
      correct_prediction = tf.equal(
          prediction, tf.argmax(ground_truth_tensor, 1))
    with tf.name_scope('accuracy'):
      evaluation_step = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
  tf.summary.scalar('accuracy', evaluation_step)
  return evaluation_step, prediction


def retrain(image_lists, optimizer_name=OPTIMIZERS[0], save_graph=True, adjustLR=
    True):

  graph, bottleneck_tensor, jpeg_data_tensor, resized_image_tensor = (
      create_inception_graph())


  with graph.as_default():
    my_learning_rate = tf.placeholder(tf.float32, shape=[],name='
        learning_rate_input') # AS

    with tf.Session(graph=graph) as sess:

      # Add the new layer that we'll be training.
      (train_step, cross_entropy, bottleneck_input, ground_truth_input,
       final_tensor) = add_final_training_ops(len(image_lists.keys()),
                                              FLAGS.final_tensor_name,
                                              bottleneck_tensor, optimizer_name,
                                                  my_learning_rate)

      # Create the operations we need to evaluate the accuracy of our new layer.
      evaluation_step, prediction = add_evaluation_step(
          final_tensor, ground_truth_input)

      # Merge all the summaries and write them out to the summaries_dir
      merged = tf.summary.merge_all()
      train_writer = tf.summary.FileWriter(FLAGS.summaries_dir + '/train',
                                           sess.graph)

      validation_writer = tf.summary.FileWriter(
          FLAGS.summaries_dir + '/validation')

      # Set up all our weights to their initial default values.
      init = tf.global_variables_initializer()
      sess.run(init)

      # AS - prepare saver
      saver = tf.train.Saver()
      randomID = random.randint(0,9999999)
      base_ckpt = '/tmp/'+str(randomID)+'_base.ckpt'
      best_ckpt = '/tmp/'+str(randomID)+'_best.ckpt'

      # AS - initial learning rate
      myLR = FLAGS.learning_rate

      # AS - keeping my own stats
```

```
mystat_steps = []
mystat_training = []
mystat_validation = []
mystat_learningrate = []
mystat_crossentropy = []

best_train_accuracy = 0
best_cross_entropy = 0 # well...

steps = FLAGS.how_many_training_steps
dynamicsteps = (FLAGS.how_many_training_steps == -1)
if dynamicsteps:
  steps = MAX_STEPS

# Run the training for as many cycles as requested on the command line.
for i in range(steps):
  print(i)

  # AS - check if stopping criterion is reached
  if dynamicsteps:
    LB = DYNAMIC_STEPS_LOOKBACK
    if i>LB:
      #avg = np.mean(mystat_validation[-LB:-1])
      #if mystat_validation[-1]<avg:
      compare = np.min(mystat_validation[-LB:-1])
      if mystat_validation[-1]<=compare:
        saver.restore(sess, base_ckpt) # previous was better
        break



  def dotraining(learning_rate_to_use_this_step):
    STEPS = 1 # TODO increase when using smaller batch sizes
    for stp in range(STEPS):

      # Get a batch of input bottleneck values from the cache stored on disk.
      (train_bottlenecks,
       train_ground_truth, _) = get_random_cached_bottlenecks(
           sess, image_lists, FLAGS.train_batch_size, 'training',
           FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor,
           bottleneck_tensor)
      # Feed the bottlenecks and ground truth into the graph, and run a
          training
      # step. Capture training summaries for TensorBoard with the `merged` op
          .

      train_summary, _ = sess.run(
          [merged, train_step],
          feed_dict={bottleneck_input: train_bottlenecks,
                     ground_truth_input: train_ground_truth,
                     my_learning_rate: learning_rate_to_use_this_step})
      #train_writer.add_summary(train_summary, i)

  def dotrainingeval():
    (train_bottlenecks,
     train_ground_truth, _) = get_random_cached_bottlenecks(
         sess, image_lists, -1, 'training',
         FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor,
         bottleneck_tensor)
    return sess.run(
        [evaluation_step, cross_entropy],
        feed_dict={bottleneck_input: train_bottlenecks,
                   ground_truth_input: train_ground_truth})

  # save state
  saver.save(sess, base_ckpt)

  # AS - adjusting learning rates
  if(adjustLR):
    previous_best_train_accuracy = best_train_accuracy
    previous_best_cross_entropy = best_cross_entropy

    # same LR
    dotraining(myLR)
    sameLR_train_accuracy, sameLR_cross_entropy = dotrainingeval()
    saver.save(sess, best_ckpt)
```

```
best_train_accuracy = sameLR_train_accuracy
best_cross_entropy = sameLR_cross_entropy
print('%s: Same LR Train accuracy = %.1f%%, cross entropy = %f' % (
    datetime.now(), sameLR_train_accuracy * 100, sameLR_cross_entropy))

# AS - check improvement
if (sameLR_cross_entropy>previous_best_cross_entropy):
  # recalculate learning rate
  initialstep = (previous_best_cross_entropy==0.0)
  if not initialstep:
    print('%s: Warning! cross entropy increased from %f to %f' % (
        datetime.now(), previous_best_cross_entropy, sameLR_cross_entropy
        ))
  currentLR = myLR
  previous_cross_entropy = sameLR_cross_entropy
  while (best_cross_entropy>previous_best_cross_entropy):
    currentLR = currentLR / 2 # decrease lr exponentially
    if(currentLR<MIN_LEARNING_RATE): break
    saver.restore(sess, base_ckpt)
    dotraining(currentLR)
    current_train_accuracy, current_cross_entropy = dotrainingeval()
    print('%s: LR = %f, Train accuracy = %.1f%%, cross entropy = %f' % (
        datetime.now(), currentLR, current_train_accuracy * 100,
        current_cross_entropy))
    if(current_cross_entropy<=previous_best_cross_entropy):
      myLR = currentLR
      best_train_accuracy = current_train_accuracy
      best_cross_entropy = current_cross_entropy
      saver.save(sess, best_ckpt)
    elif(initialstep):
      if(current_cross_entropy>previous_cross_entropy):
        saver.restore(sess, best_ckpt)
        break
      else:
        previous_cross_entropy = current_cross_entropy
        myLR = currentLR
        best_train_accuracy = current_train_accuracy
        best_cross_entropy = current_cross_entropy
        saver.save(sess, best_ckpt)

else: # go on as normal (cross entropy did not increase)


  # AS - calculate modifyer
  c = CHANGING_LEARNING_RATE_ADJUST_PERCENTAGE_MODIFYER
  cross_entropy_diff = previous_best_cross_entropy-sameLR_cross_entropy
  cross_entropy_diff_perc = abs(cross_entropy_diff/sameLR_cross_entropy)
  cmin = CHANGING_LEARNING_RATE_MIN

  lrmodify = cmin+(c*cross_entropy_diff_perc)

  highLR = myLR*lrmodify
  lowLR = myLR/lrmodify
  #hugeLR = myLR*20
  #tinyLR = myLR/20

  print('%s: LR modifyer %.3f: %3f and %3f' % (datetime.now(), lrmodify,
      lowLR, highLR))


  # low
  saver.restore(sess, base_ckpt)
  dotraining(lowLR)
  lowLR_train_accuracy, lowLR_cross_entropy = dotrainingeval()
  print('%s: Low LR Train accuracy = %.1f%%, cross entropy = %f' % (
      datetime.now(), lowLR_train_accuracy * 100, lowLR_cross_entropy))
  #if (lowLR_train_accuracy>compare_train_accuracy):
  if(lowLR_cross_entropy<best_cross_entropy):
    myLR = lowLR
    best_train_accuracy = lowLR_train_accuracy
    compare_train_accuracy = best_train_accuracy
    best_cross_entropy = lowLR_cross_entropy
    saver.save(sess, best_ckpt)


  # high
```

```
      saver.restore(sess, base_ckpt)
      dotraining(highLR)
      highLR_train_accuracy, highLR_cross_entropy = dotrainingeval()
      print('%s: High LR Train accuracy = %.1f%%, cross entropy = %f' % (
          datetime.now(), highLR_train_accuracy * 100, highLR_cross_entropy))
      #if (highLR_train_accuracy>compare_train_accuracy):
      if(highLR_cross_entropy<best_cross_entropy):
        myLR = highLR
        best_train_accuracy = highLR_train_accuracy
        compare_train_accuracy = best_train_accuracy
        best_cross_entropy = highLR_cross_entropy
        #saver.save(sess, best_ckpt)
      else:
        saver.restore(sess, best_ckpt)


    print ("%s: proceeding with LR = %f"%(datetime.now(),myLR))

  else:
    dotraining(myLR)

  # Every so often, print out how well the graph is training.
  is_last_step = (i + 1 == FLAGS.how_many_training_steps)
  if (i % FLAGS.eval_step_interval) == 0 or is_last_step:
    train_accuracy, cross_entropy_value = dotrainingeval()
    print('%s: Step %d: Train accuracy = %.1f%%' % (datetime.now(), i,
                                                    train_accuracy * 100))
    print('%s: Step %d: Cross entropy = %f' % (datetime.now(), i,
                                               cross_entropy_value))
    validation_bottlenecks, validation_ground_truth, _ = (
        get_random_cached_bottlenecks(
            sess, image_lists, FLAGS.validation_batch_size, 'validation',
            FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor,
            bottleneck_tensor))
    # Run a validation step and capture training summaries for TensorBoard
    # with the `merged` op.
    validation_summary, validation_accuracy = sess.run(
        [merged, evaluation_step],
        feed_dict={bottleneck_input: validation_bottlenecks,
                   ground_truth_input: validation_ground_truth})
    validation_writer.add_summary(validation_summary, i)
    print('%s: Step %d: Validation accuracy = %.1f%% (N=%d)' %
          (datetime.now(), i, validation_accuracy * 100,
           len(validation_bottlenecks)))

    mystat_steps.append(i)
    mystat_training.append(train_accuracy)
    mystat_validation.append(validation_accuracy)
    mystat_learningrate.append(myLR)
    mystat_crossentropy.append(cross_entropy_value)

# We've completed all our training, so run a final test evaluation on
# some new images we haven't used before.
test_bottlenecks, test_ground_truth, test_filenames = (
    get_random_cached_bottlenecks(sess, image_lists, FLAGS.test_batch_size,
                                  'testing', FLAGS.bottleneck_dir,
                                  FLAGS.image_dir, jpeg_data_tensor,
                                  bottleneck_tensor))
test_accuracy, predictions = sess.run(
    [evaluation_step, prediction],
    feed_dict={bottleneck_input: test_bottlenecks,
               ground_truth_input: test_ground_truth})
print('Final test accuracy = %.1f%% (N=%d)' % (
    test_accuracy * 100, len(test_bottlenecks)))

# AS - prepare for confusion matrix
labels = []
for i, test_filename in enumerate(test_filenames):
  labels.append(test_ground_truth[i].argmax())

if FLAGS.print_misclassified_test_images:
  print('=== MISCLASSIFIED TEST IMAGES ===')
  for i, test_filename in enumerate(test_filenames):
    if predictions[i] != test_ground_truth[i].argmax():
      print('%70s  %s' % (test_filename,
```

```
                                    list(image_lists.keys())[predictions[i]]))

      if (save_graph):
        # Write out the trained graph and labels with the weights stored as
        # constants.
        output_graph_def = graph_util.convert_variables_to_constants(
            sess, graph.as_graph_def(), [FLAGS.final_tensor_name])
        with gfile.FastGFile(FLAGS.output_graph, 'wb') as f:
          f.write(output_graph_def.SerializeToString())
        with gfile.FastGFile(FLAGS.output_labels, 'w') as f:
          f.write('\n'.join(image_lists.keys()) + '\n')

      # AS - return statistics
      mystat = {
          "steps": mystat_steps,
          "training": mystat_training,
          "validation": mystat_validation,
          "learning_rate": mystat_learningrate,
          "cross_entropy": mystat_crossentropy,
          "evaluation": test_accuracy,
          "predictions": predictions,
          "truths": labels,
          "labels": list(image_lists.keys())
          }
      return mystat

def main(_):
  # Setup the directory we'll write summaries to for TensorBoard
  if tf.gfile.Exists(FLAGS.summaries_dir):
    tf.gfile.DeleteRecursively(FLAGS.summaries_dir)
  tf.gfile.MakeDirs(FLAGS.summaries_dir)

  # Look at the folder structure, and create lists of all the images.
  image_lists = create_image_lists(FLAGS.image_dir)
  class_count = len(image_lists.keys())
  if class_count == 0:
    print('No valid folders of images found at ' + FLAGS.image_dir)
    return -1
  if class_count == 1:
    print('Only one valid folder of images found at ' + FLAGS.image_dir +
          ' - multiple classes are needed for classification.')
    return -1


  # Set up the pre-trained graph.
  maybe_download_and_extract()
  graph, bottleneck_tensor, jpeg_data_tensor, resized_image_tensor = (
      create_inception_graph())


   # AS - if user is sure bottlenecks exist this part can be skipped
  bottlenecks_exist = FLAGS.bottlenecks_exist
  if not bottlenecks_exist:
    with tf.Session(graph=graph) as sess:
      # We'll make sure we've calculated the 'bottleneck' image summaries and
      # cached them on disk.
      cache_bottlenecks(sess, image_lists, FLAGS.image_dir,
                        FLAGS.bottleneck_dir, jpeg_data_tensor,
                        bottleneck_tensor)

  # AS - run retrainings for each optimizer
  optims = OPTIMIZERS
  lr = FLAGS.learning_rate
  if FLAGS.optimizer:
    optims = [FLAGS.optimizer]
  for optim in optims:
    adjustLR = optim.startswith('ALRGD')
    lrs = str(lr).replace('.', 'p')
    vers='vT' # version string for output file
    stepsstr = str(FLAGS.how_many_training_steps)
    if(FLAGS.how_many_training_steps == -1):
      stepsstr = 'dynamic'
    filename = optim+'_'+vers+'_'+lrs+'_'+stepsstr+'.data'
    path = os.path.join(FLAGS.output_dir, filename)
    if os.path.exists(path):
      print(filename + " already exists. - skip")
```

```
        continue
      else:
        print("working on "+optim+ " with lr="+str(lr)+" "+stepsstr+" steps.")

      with open(path, "wb") as picklefile:
        mystats = retrain(image_lists,optim,True,adjustLR)
        pickle.dump(mystats, picklefile)


if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument(
      '--image_dir',
      type=str,
      default='',
      help='Path to folders of labeled images.'
  )
  parser.add_argument(
      '--output_graph',
      type=str,
      default='/tmp/output_graph.pb',
      help='Where to save the trained graph.'
  )
  parser.add_argument(
      '--output_labels',
      type=str,
      default='/tmp/output_labels.txt',
      help='Where to save the trained graph\'s labels.'
  )
  parser.add_argument(
      '--summaries_dir',
      type=str,
      default='/tmp/retrain_logs',
      help='Where to save summary logs for TensorBoard.'
  )
  parser.add_argument(
      '--how_many_training_steps',
      type=int,
      default=4000,
      help='How many training steps to run before ending.'
  )
  parser.add_argument(
      '--learning_rate',
      type=float,
      default=0.001,#0.01,
      help='How large a learning rate to use when training.'
  )
  parser.add_argument(
      '--eval_step_interval',
      type=int,
      default=10,
      help='How often to evaluate the training results.'
  )
  parser.add_argument(
      '--train_batch_size',
      type=int,
      default=-1,#100,
      help='How many images to train on at a time.'
  )
  parser.add_argument(
      '--test_batch_size',
      type=int,
      default=-1,
      help="""\
      How many images to test on. This test set is only used once, to evaluate
      the final accuracy of the model after training completes.
      A value of -1 causes the entire test set to be used, which leads to more
      stable results across runs.\
      """
  )
  parser.add_argument(
      '--validation_batch_size',
      type=int,
      default=-1,#100,
      help="""\
```

```
        How many images to use in an evaluation batch. This validation set is
        used much more often than the test set, and is an early indicator of how
        accurate the model is during training.
        A value of -1 causes the entire validation set to be used, which leads to
        more stable results across training iterations, but may be slower on large
        training sets.\
        """
    )
    parser.add_argument(
        '--print_misclassified_test_images',
        default=False,
        help="""\
        Whether to print out a list of all misclassified test images.\
        """,
        action='store_true'
    )
    parser.add_argument(
        '--model_dir',
        type=str,
        default='/tmp/imagenet',
        help="""\
        Path to classify_image_graph_def.pb,
        imagenet_synset_to_human_label_map.txt, and
        imagenet_2012_challenge_label_map_proto.pbtxt.\
        """
    )
    parser.add_argument(
        '--bottleneck_dir',
        type=str,
        default='/tmp/bottleneck',
        help='Path to cache bottleneck layer values as files.'
    )
    parser.add_argument(
        '--bottlenecks_exist',
        default=False,
        help="""\
        Whether to skip the bottleneck creation step.\
        """,
        action='store_true'
    )
    parser.add_argument(
        '--final_tensor_name',
        type=str,
        default='final_result',
        help="""\
        The name of the output classification layer in the retrained graph.\
        """
    )
    parser.add_argument(
        '--output_dir',
        type=str,
        default='.',
        help='Path to the directory to store training statistics.'
    )
    parser.add_argument(
        '--optimizer',
        type=str,
        help="Select the optimizer to use. Possible values are ['ALRGD', 'ALRGD_adam
            ', 'adadelta', 'gradientdescent', 'adagrad', 'adam', 'ftrl'] "
    )
    FLAGS, unparsed = parser.parse_known_args()
    tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```