Gabriele Tiboni
S276241

Report for Homework #2

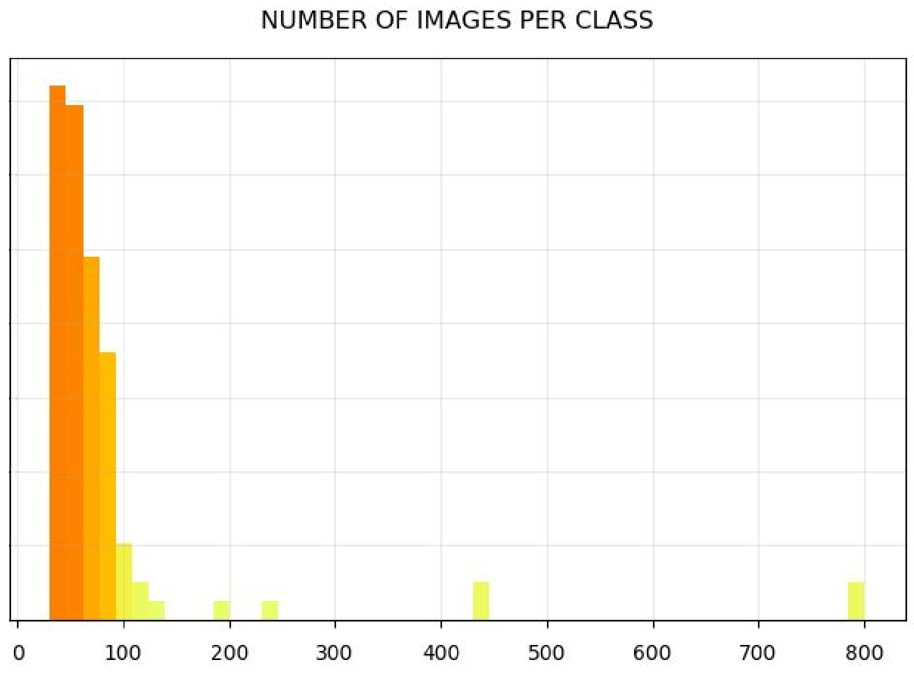
MACHINE LEARNING AND DEEP LEARNING COURSE

1. Introduction

In this work the Caltech-101 dataset will be used to perform a classification task on images. Specifically, 101 different object categories have to be recognized by using the AlexNet deep neural network structure.

The dataset is relatively small compared to other datasets used for computer vision and it is composed of a total of **8677 images** (after removing the BACKGROUND_Google class).

Each image is approx. 300x200 pixels (but sometimes smaller) and their distribution across the 101 categories is as follows:



Mean of about 86 images per class and Standard deviation of 118 images.

As shown, some classes are expected to be more easily recognizable as the dataset is not smoothly spread across the 101 labels, but it is safe to say that the majority of the object categories have about 50-60 images.

In particular, some of the outlier classes are:

- "*Airplanes*" have **800** images
- "*Motorbikes*" have **798** images
- "*Faces*" have **435** images
- "*Faces_easy*" have **435** images

The goal of the work will be to check how well a convolutional neural network can perform when trained from scratch and how much the results are affected if a transfer learning technique is instead adopted, by exploiting a pretrained network on a similar computer vision task.
Furthermore, data augmentation will be taken into account in the attempt of avoiding overfitting for such a small dataset.

2. Data preparation

The Caltech-101 dataset has firstly been preprocessed for the task by using an ad hoc dataset loader class. Images have been resized (short edge) to 256 pixels and later center-cropped for a final 224x224 square image, which is the expected input size by AlexNet.

When uploaded in memory, all channels of each image have been normalized to have mean around zero and a standard deviation of about 1, in order to reduce the vanishing gradient effect on the input data. Indeed, neural networks suffer from non-centered datasets being propagated through the network and batch normalization should also be considered to achieve the best convergence times and performances.
The *BACKGROUND_Google* class has been filtered out during the preprocessing phase, making the dataset of effectively 101 classes, as previously described.

The initial training set provided as in "train.txt" has been finally split into training and validation sets in a random stratified sampling fashion, in order to preserve enough training samples for each category and enough validation samples for each label to make the evaluations statistically matter.

Therefore, the final splits are as shown:

- Training set: **2892 images**
- Validation set: **2892 images**
- Test set: **2893 images**

3. Training from scratch

In the first case, AlexNet has been used as is (with no pretrained weights) in the attempt to train a good model from scratch, but still exploiting its important structure.

AlexNet is composed of a first series of **5 Convolutional layers** and a later series of **3 Fully connected layers**. ReLu activation functions are used and Max pooling technique is adopted after three out of the five conv. layers together with an adaptive average pooling at the end of the convolution part of the network. Also, Dropout with probability 0.5 is used on the first two FC layers in order to prevent co-adaptation of hidden neurons during training.

Cross Entropy is the main loss function used in deep learning classification tasks and will be the function of choice throughout this work.

When trained from scratch, PyTorch initializes the network with small random weights (however not exactly a Xavier's initialization [1]) such that we expect an output that does not diverge (for some input of finite variance across batches) and that shows no bias over certain classes randomly at start (output probability should be very similar for all classes).

Indeed, the network outputs an average loss function of about **4.615 +/- 0.005** at each start: this loss reflects how the random initialization performed by PyTorch together with the small variance in the input data make sure that the output probabilities of each class are all very close to **0.0099**, which is the probability of randomly choosing one label among 101. Note that **4.615 = $-\log_e$(1/101)** (theoretical cross entropy loss on one sample) which is ideally what we expect the network to output for a given sample when no training is performed[1].

Before doing any considerations on the hyperparameters, the network has been trained with the given values out of the box:

- Mini-batch Stochastic Gradient descent with Momentum **0.9**, Weight decay **5e-5** and Learning rate **1e-3**
- Batch size: **256**
- Number of epochs: **30**
- Step-down policy for LR after **20** epochs with Gamma **0.1**

This is how the model initially performed:

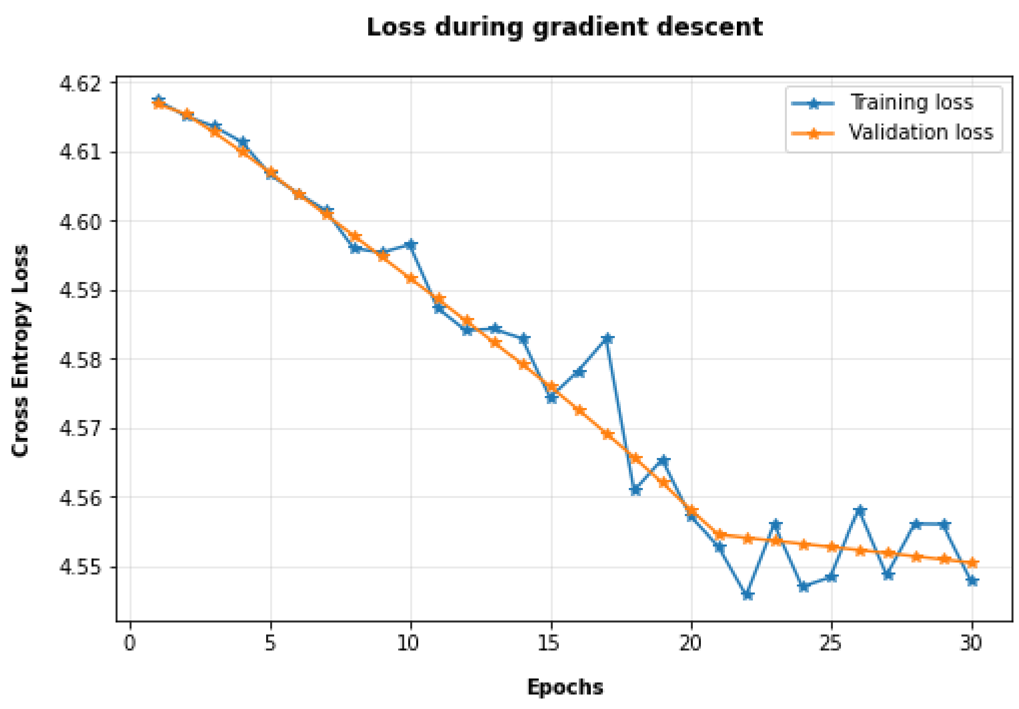


Figure 1: *Learning rate of 0.001*

It is important to note that the higher fluctuations in the training loss w.r.t. the validation loss are due to how the two measures are computed: the training loss is the cross entropy loss averaged on the samples of the current minibatch, while the validation loss has been calculated as the average cross entropy on the **whole** validation set. Indeed, in mini-batch SGD only an estimate of the gradient can be computed at each iteration and this can sometimes lead to a subsequent higher loss on the next mini-batches.

With the given hyperparameters, our very first model was able to achieve better performances as the number of epochs increased. However, it could only learn so much: the final cost obtained is about **4.55** which indicates that the output average probability of the correct class is likely just slightly above **0.01**[2], which is still definitely too low. The model suffered from a low learning rate and wasn't able to converge fast enough. In addition, as long as the average loss function decreases on both sets it is safe to keep iterating with more epochs or a higher learning rate.

Finally, the decay policy for the learning rate can be greatly observed in the graph above where after 20 epochs the model was forced to learn even more slowly.

Thus, in the attempt to achieve better results and a lower cost function, a new training has been done with a 10 times higher learning rate (0.01), keeping the other hyperparameters the same:

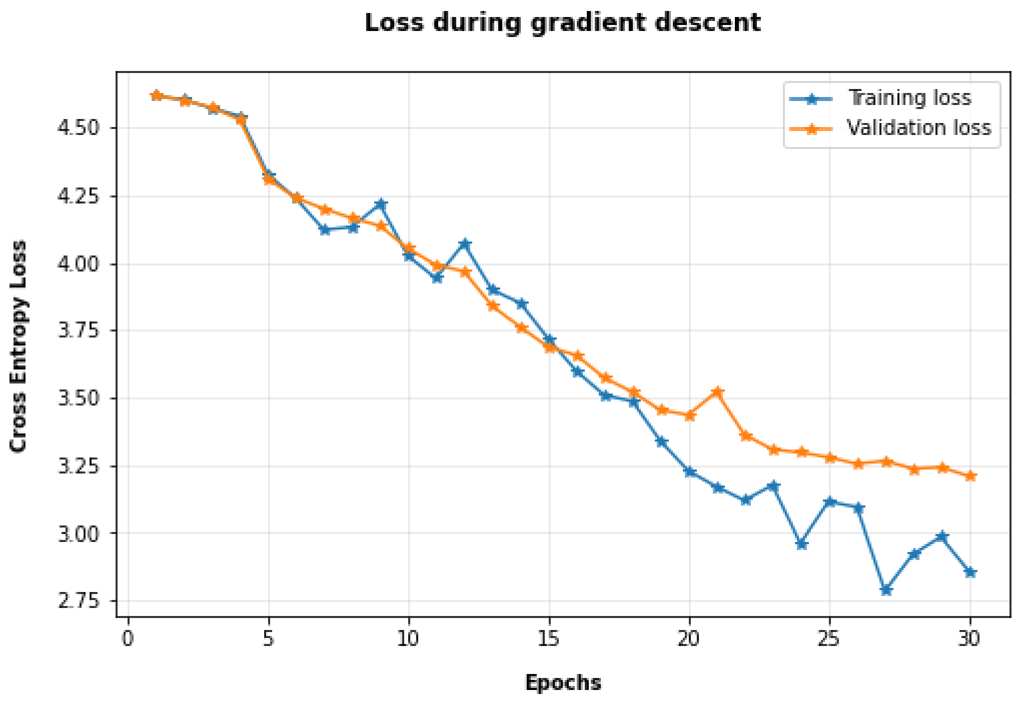


Figure 2: *Learning rate of 0.01*

This time, with a higher learning rate the model was able to reduce the cost function down to about **2.85** on the training set, without yet showing overfitting. The accuracy on the validation set was a solid **30%**, a lot higher than the previous almost-random classification model. Accuracy on training set was about **31%** and the average output probability of the correct class on the training set was **0.24**.

However, as the two losses on the graph above keep having a decreasing trend over 30 epochs, it is reasonable to keep tuning the learning rate together with the number of epochs until overfitting is shown and the training loss is very small.

Once an even larger learning rate was set (0.1) the model performed poorly and diverged after 10 epochs:

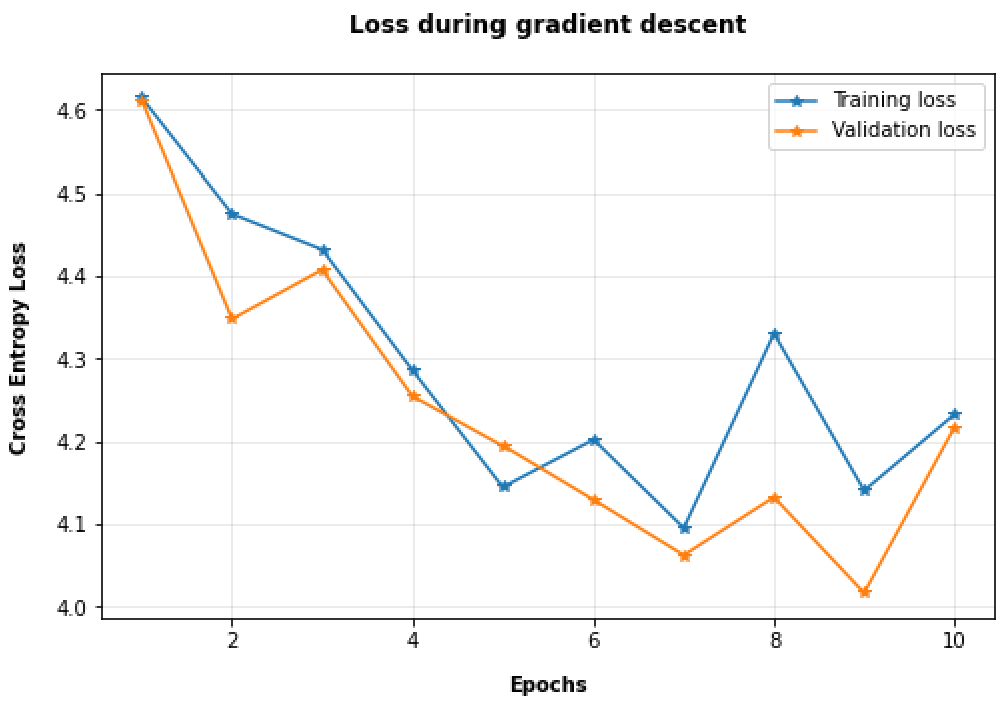


Figure 3: *With a learning rate of 0.1 the loss function diverges after 10 epochs.*

The network showed to be very sensitive to changes in the LR (as theory says) and finally an upper bound to the LR was found for this network. It is now safe to keep the LR in between **0.01** and **0.05** for the hypertuning of the network.

A small gridsearch on both the learning rate and the number of epochs has been performed, with the following values:

- LR: **[0.01, 0.05]**
- Number of epochs: **[30, 60]**

Showing the following results in terms of training loss, validation loss and accuracy on both sets at the end of the given number of epochs[3]:

	# Epochs = 30	*# Epochs = 60*
LR = 0.01	Train loss: **2.85** Val loss: **3.20** Train accuracy: **31.4%** Val accuracy: **30.2%**	Train loss: **0.31** Val loss: **3.19** Train accuracy: **94.9%** Val accuracy: **51.2%**
LR = 0.05	Train loss: **1.09** Val loss: **3.07** BEST Train accuracy: **78.7%** Val accuracy: **44.9%**	Train loss: **0.004** Val loss: **3.75** Train accuracy: **97.3%** Val accuracy: **53.7%** BEST

Table 1: *When training on 60 epochs, the decaying policy for the LR has been set to start after 45 epochs.*

By increasing the number of epochs to 60 we were finally able to overfit the network and reach convergence on the learning process for the training set. Though, this clearly showed overfitting and an upward trend in the validation loss after some time. The ideal hyperparameters would allow us to stop right when reaching the overfitting point and get both losses as low as possible.

The model with *LR*=0.05 and *number of Epochs*=60 from the GridSearch in the above table is described below by the following two figures:

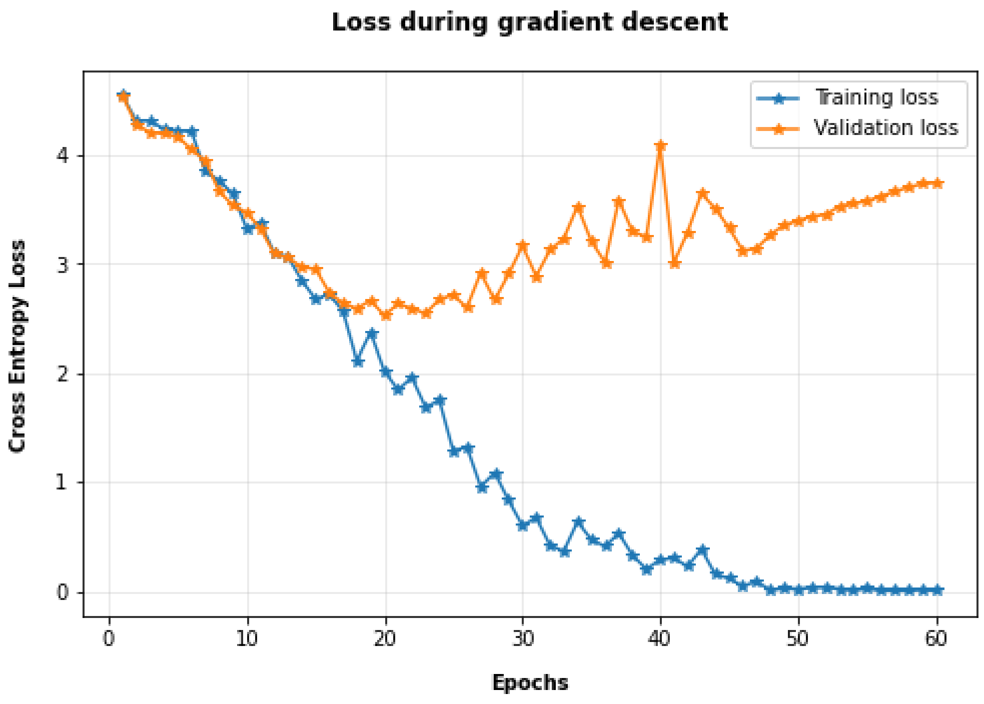


Figure 4: *Loss during training with the best LR found*

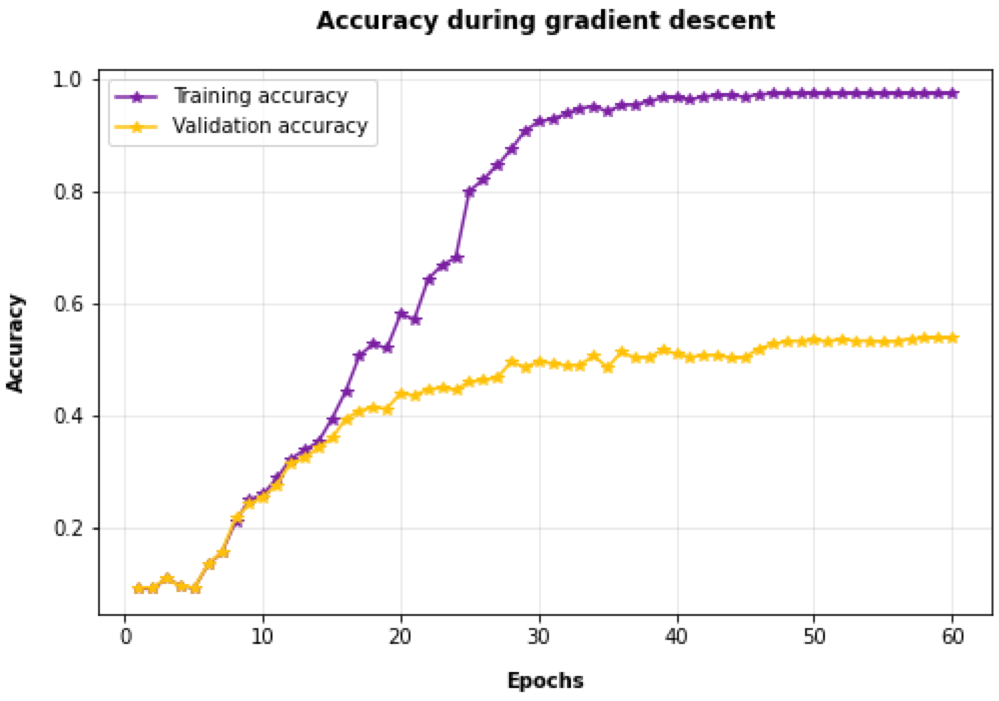


Figure 5: *Training and validation accuracies both calculated on the entire sets after each epoch.*

The model was able to converge fairly well already after 30 epochs and showed a steeper learning curve w.r.t. to the slower learner with *LR*=0.01.

It is interesting to note that while the validation loss was increasing due to overfitting, the accuracy on the validation set still remained quite the same or even increased. This is very well possible as the accuracy only takes into account whether the correct label had the highest probability among the softmax outputs, and not what this probability actually was. Also, the labels in the dataset are not balanced (e.g. airplanes have about 800 images whereas other labels have only 50 images on average) and it might result in affecting the loss and accuracy differently. Still, the upward trend in loss cannot be ignored and regularization has to take care of this in order to achieve the best results.

> The above model was chosen among the four for the **final evaluation** on the test set and achieved a **53.9% accuracy**.

As advised during the course, now it should be time to tune the regularization hyperparameters and make the model generalize as best as possible, avoiding some of the overfitting observed during training. Furthermore, the number of epochs should be kept under analysis in order to stop the model once the loss on the validation set noticeably starts increasing. Finally, a data augmentation task might also help the model generalize better and will be considered later on in this work with the pretrained AlexNet model.

The current chosen model has been then run with new sets of hyperparameters. In particular, a higher regularization in terms of *weight decay* was tried, as well as a different optimizer for the gradient descent (Adam):

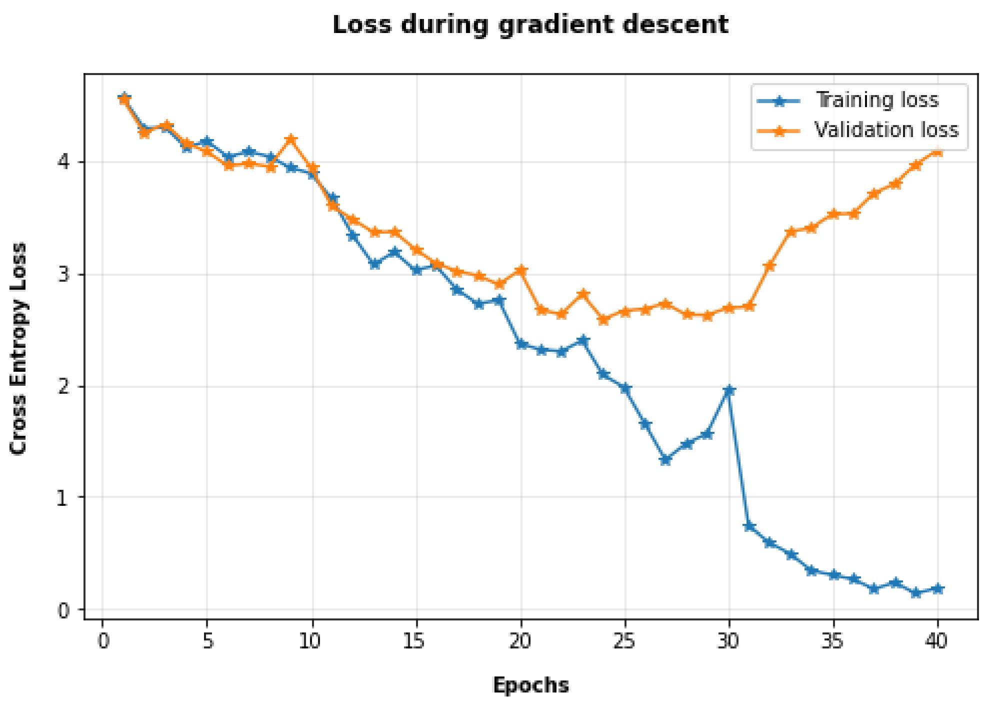


Figure 6: *L2 regularization with weight decay of 5e-3*

Unfortunately, increasing the strength of the regularization did not help much the model in closing the gap between the two losses, and the network still showed a high overfitting when converged.

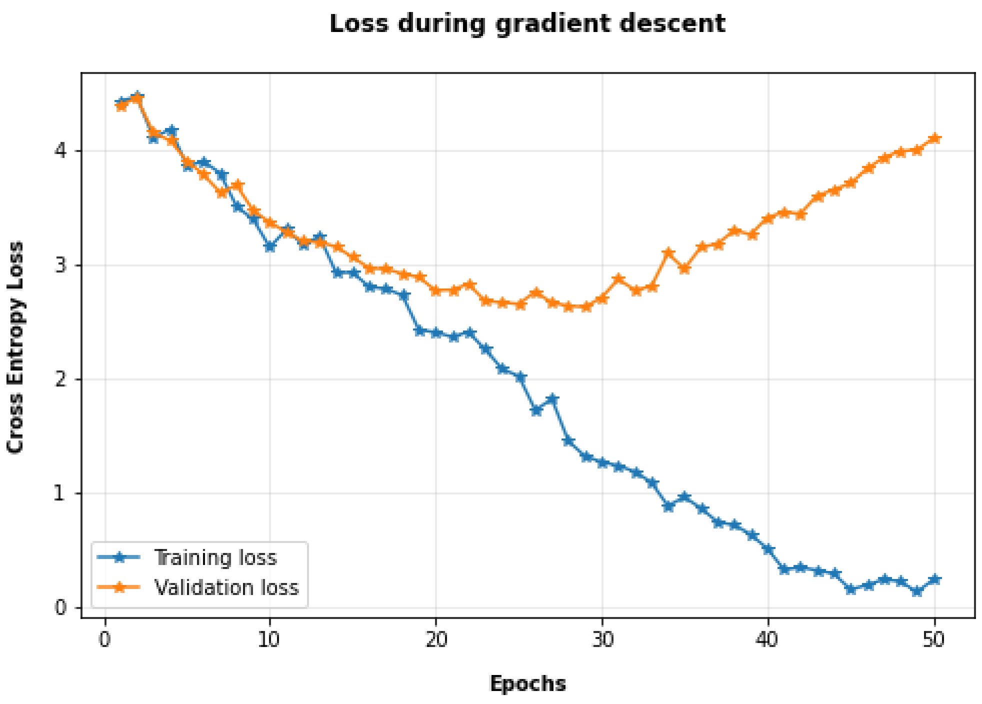


Figure 7: *Adam optimizer with LR=0.001*

Also, using a different optimizer for the gradient descent showed a similar convergence path and did not fix the overfitting issue.

For such a small dataset, overfitting is the main problem when dealing with the training of a new network from scratch and a **transfer learning** technique needs to be considered.

Though, before moving on to transfer learning, a couple final experiments on the current network have been done. How does the network behave if input features are not zero-mean or have a larger variance?

When input data is not centered, the network was not able to learn as fast, and gradient descent would perform very poorly. Indeed, the model could not converge and the training loss roughly reached only about 4.0 after 30 epochs:

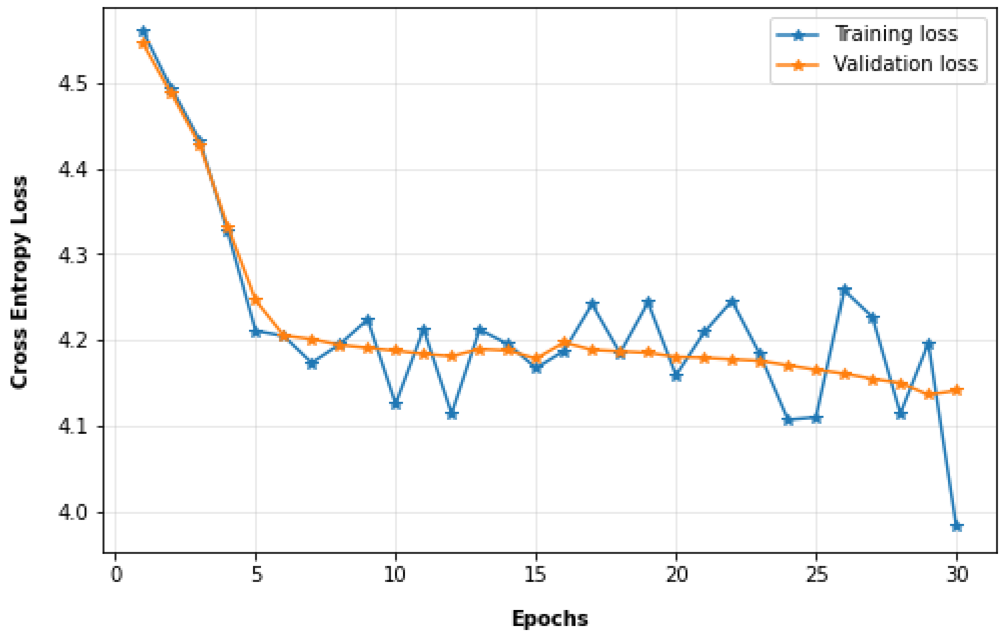


Figure 8: *Model with input features that are always positive.*

Instead, when input features have a very large variance, the starting outputs are more spread out in values and lead to a higher initial loss due to the non-linear log() function of the Cross Entropy (penalizes more as the output probability gets lower). Plus, the model loss tends to diverge after just a couple of iterations.

4. Transfer learning

Given the very small dataset, it was expected to obtain poor results when training a network from scratch. Transfer learning comes in handy in these situations and in practice really helps to boost up accuracies and build a better model. The idea is based on using a pre-existing neural network built from a very large dataset that is somewhat similar to the current one (i.e. other pictures of animals and people) to later finetune the network weights with our smaller dataset. Hopefully, the pretrained network learned generic patterns among images that will also be useful for our current task and will help our model generalize better, even with a limited amount of training samples.

Therefore, a pretrained model of AlexNet built on a much larger dataset on ImageNet was loaded and used it as a starting point for our classification task.

As stated in the Documentation, also the input standardization should change accordingly when using a pretrained network. Thus, the input features have been firstly normalized differently with the following means and standard deviations for each channel:

- means: (**0.485, 0.456, 0.406**)
- st. deviations: (**0.229, 0.224, 0.225**)

The learning rate has been reset to the lower value **0.001** and the other hyperparameters left as in the initial configuration.

The first run gave the following results:

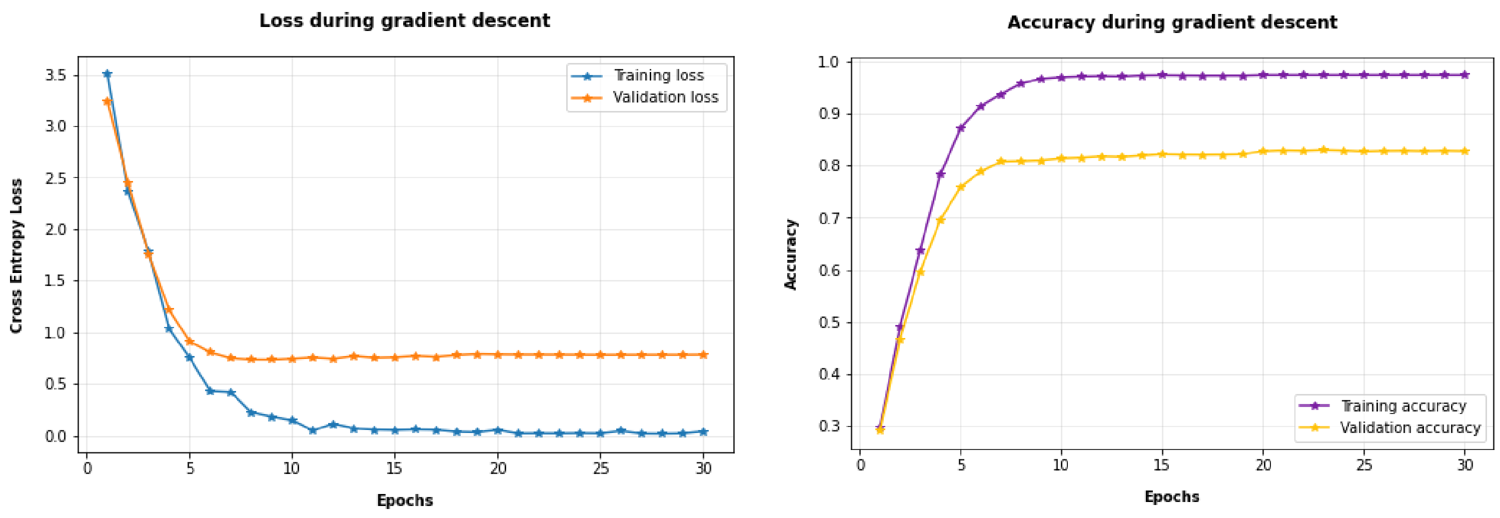


Figure 9: *Training on the pretrained model of AlexNet with LR=0.001, BatchSize=256, Weight decay=5e-5, # of Epochs=30 and LR-Step-Down every 20 epochs with gamma=0.1.*

Huge improvements are noticeable already from the very first run. The model is not showing overfitting on the validation set as the gap in between the two losses is not increasing over time, and the network is now able to converge much faster to low loss values. The loss on the validation set was this time brought all the way down to about **0.78**.

The final accuracies on the training and validation sets were respectively of **97.3%** and **82.8%**.

For the second run the learning rate was increased to 0.01, to monitor how sensitive the current model was to changes in the LR:

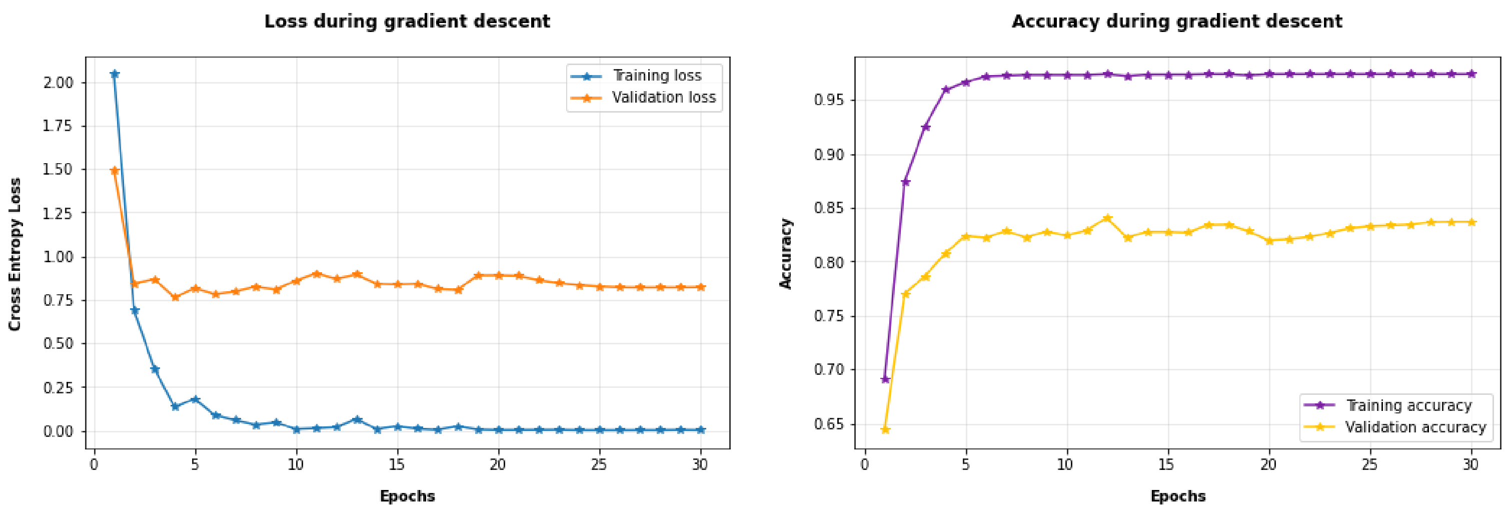


Figure 10

With a higher learning rate the model converged even faster, after only about **5-7 epochs**. The final loss on the validation set was **0.82**, but could have been lower if the model had been stopped earlier. The final validation accuracy recorded was **83.7%**, slightly higher than the previous model with a lower LR.

Later, the learning rate was once again increased to **0.05**, but the model started **diverging**.

Given these initial observations, a small grid search has been set up looking for the best hyperparameters values among the learning rate, the number of epochs and the LR-decay step:

	LR=1e-2	*LR=5e-3*	*LR=1e-3*
Num of epochs: 30 *Decay after: 20 epochs*	TL: **0.0016** +/- **0.0006** VL: **0.81** +/- **0.03** TA: **97.37%** +/- **0.01%** VA: **84.15%** +/- **0.41%**	TL: **0.005** +/- **0.003** VL: **0.76** +/- **0.04** TA: **97.32%** +/- **0.02%** VA: **84.46%** +/- **0.58%**	TL: **0.035** +/- **0.004** VL: **0.75** +/- **0.02** TA: **97.33%** +/- **0.02%** VA: **83.01%** +/- **0.26%**
Num of epochs: 15 *Decay after: 10 epochs*	TL: **0.0021** +/- **0.0007** VL: **0.78** +/- **0.04** TA: **97.35%** +/- **0.03%** VA: **84.05%** +/- **0.45%**	TL: **0.011** +/- **0.008** VL: **0.719** +/- **0.005** TA.: **97.34%** VA: **84.34%** +/- **0.25%**	TL: **0.087** +/- **0.009** VL: **0.70** +/- **0.02** TA: **97.10%** +/- **0.09%** VA: **81.96%** +/- **0.42%**
Num of epochs: 10 *Decay after: 6 epochs*	TL: **0.009** +/- **0.002** VL: **0.69** +/- **0.02** TA: **97.34%** +/- **0.03%** VA: **83.87%** +/- **0.52%**	TL: **0.029** +/- **0.009** VL: **0.65** +/- **0.02** BEST TA: **97.27%** +/- **0.03%** VA: **84.29%** +/- **0.67%**	TL: **0.35** +/- **0.05** VL: **0.75** +/- **0.02** TA: **92.27%** +/- **0.47%** VA: **80.17%** +/- **0.26%**

Table 2: *TL=Training loss, VL=Validation loss, TA=Training accuracy, VA=Validation accuracy. Means and standard deviations are computed over three runs.*

The best model has been taken in terms of best loss obtained on the validation set. As previously discussed, accuracy is likely not the best metric to use in this case also given the unbalanced classes in the dataset.

From the table above, it can be observed that with a reasonably high learning rate the convergence on the training set was reached very fastly, and stepping down the learning rate sooner consistently improved the results.

The best configuration found was:

- *Learning rate:* **5e-3**
- *LR-StepDown every* **6 epochs** *of a factor of 0.1*

The current best model was rerun for 20 epochs to see if even better results could be obtained by giving it more time:

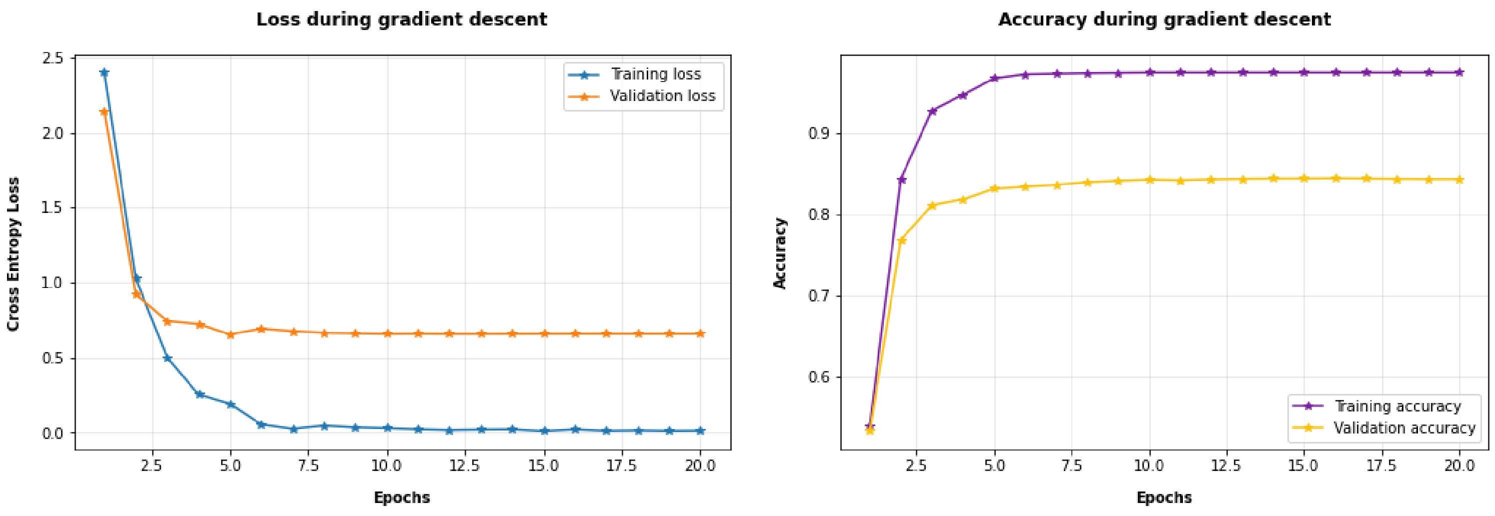


Figure 11: *LR=5e-3, 20 epochs with LR-StepDown policy of a factor of 0.1 every 6 epochs*

The model did not improve after 10 epochs and remained very much the same.

The current best model was then used for the **final evaluation** on the test set, and obtained a good **85.24% accuracy**.

Thus, transfer learning was already being able to increase the accuracy of our classifier from about **54%** (when learning from scratch) to **85%**.

Later, a couple more hyperparameters have been tried to further check whether other improvements can be obtained:

1. Stronger regularization with *weight decay* **0.001**:

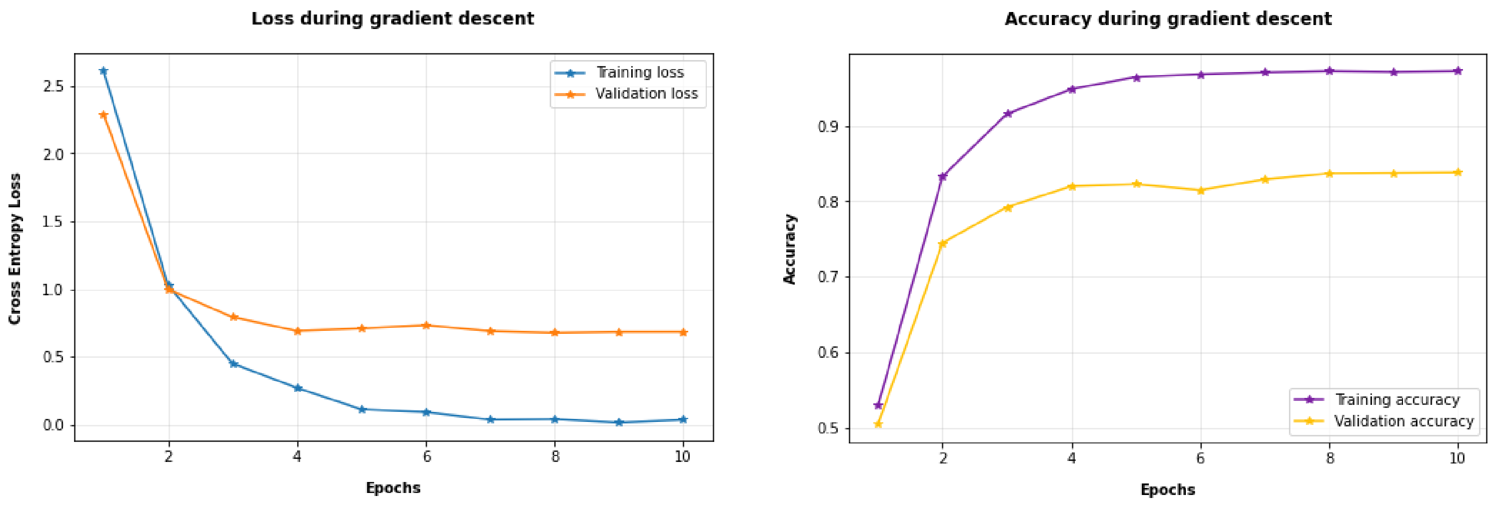


Figure 12

The model gave very similar results with a stronger L2 regularization. Remind that Dropout is already being used as a regularization effect inside the network.

2. Adam optimizer for gradient descent with *LR*=**1e-3** and default *Beta's* **(0.9, 0.999)**:

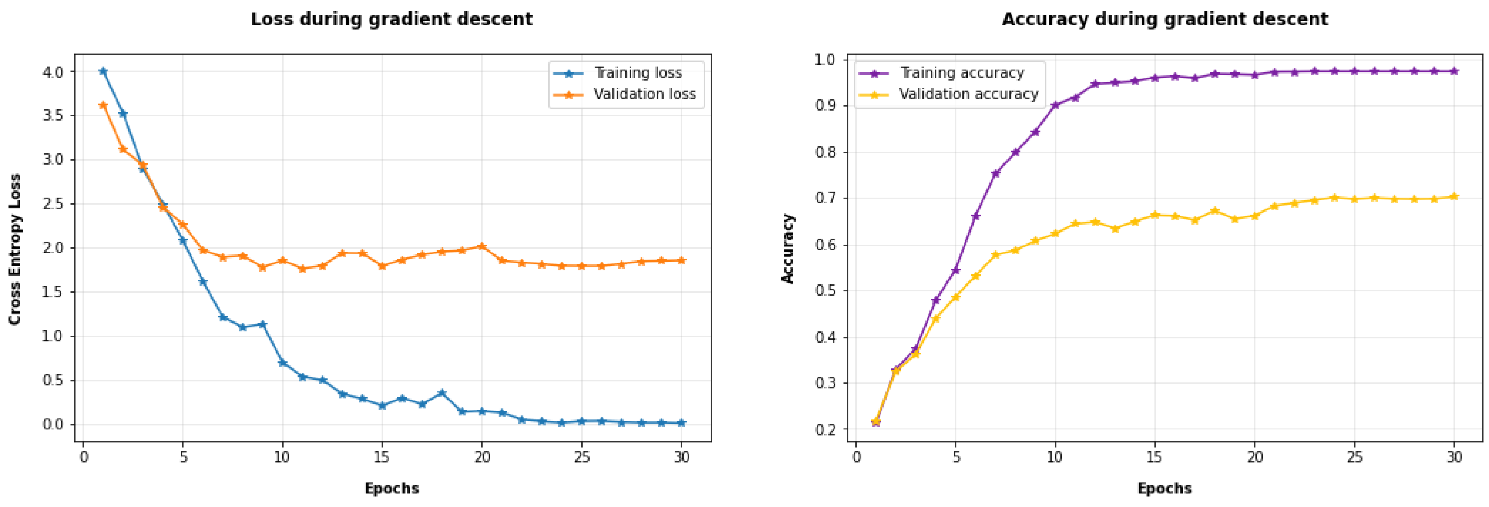


Figure 13

Adam optimizer performed very poorly with respect to SGD and, while the model converged, the validation loss was only able to go down to about **1.8**.

The current best model was obtained by finetuning the whole pretrained network. Though, other approaches to transfer learning suggest to freeze some layers of the network entirely and train the new model by only tuning the remaining layers. This is particularly effective when very few samples in our training set are available.

Therefore, the current best configuration of hyperparameters was used to train a new model with **all convolutional layers frozen** (weight updates will be computed only for the fully connected layers parameters):

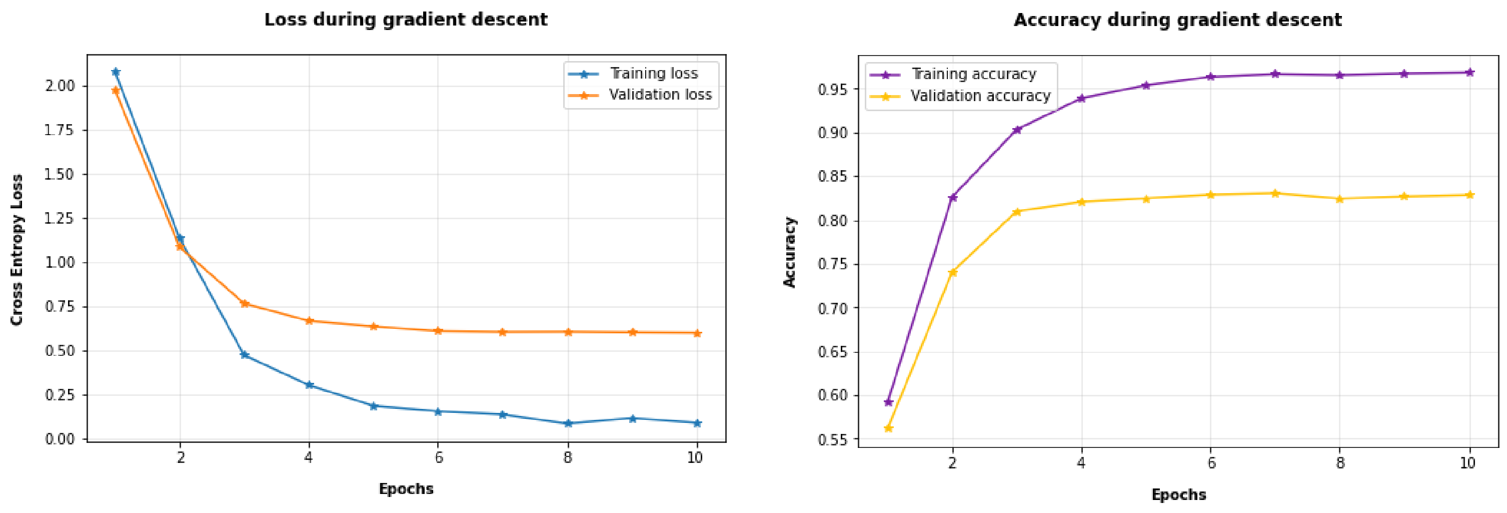


Figure 14

Freezing the first part of the network related to the low level features extracted automatically by the convolutional layers resulted in a slightly lower validation loss with the same configuration of hyperparameters. In particular, the recorded values were:

- Final validation loss: **0.59**
- Final accuracy on validation: **83.7%**

Later, the same experiment was conducted in reverse order, by **freezing all the fully connected layers** (training only the convolutional layers). The results are shown in fig. 15:

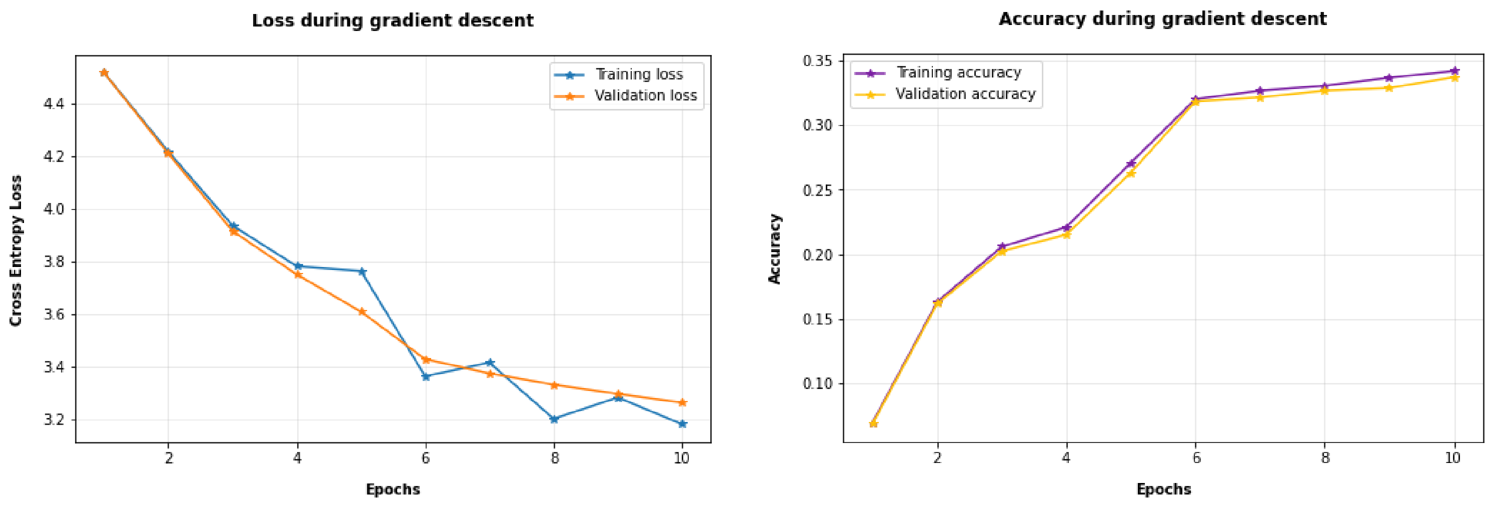


Figure 15: *The model could not converge when the FC layers were frozen.*

This time, the network wasn't able to converge in 10 epochs and learning happened very slowly compared to the previous model. This result shows that it is harder to adjust the convolutional layers on our given dataset based on the frozen fully connected layers. Indeed, even when run for 30 epochs, the model did not converge by only learning the convolutional layers, and both the validation and training loss remained at about **2.80**.

Overall, doing transfer learning from a pretrained network allowed to noticeably increase performances both in terms of accuracy and convergence speed. The loss on the validation was decreased to a best value of **0.59** (from the **3.07** obtained when learning from scratch) and accuracy on the validation set was increased to a maximum of **85.24%**. Furthermore, training the network by freezing the first convolutional layers led to a lower final validation loss w.r.t. finetuning the entire network, while freezing the last FC layers showed convergence issues.

5. Data augmentation

In addition to transfer learning, a data augmentation task can also be performed to further prevent overfitting on the given dataset. The strategy consists in randomly applying transformations on images at runtime, so that data is augmented during training and the model is forced to generalize more. Transformations need to preserve class label and to result in images that are still realistic and useful for the task. For example, horizontal flips and random crops of the images are often used (See all transformations available in PyTorch [5]).

The previous best model with convolutional layers frozen was rerun with dataset augmentation on the training set only, by adding different sets of image transformations:

	Evaluation metrics
No data augmentation	TL: **0.10** +/- **0.01** VL: **0.59** +/- **0.01** VA: **83.66%** +/- **0.21%**
Horizontal flip (p=0.5)	TL: **0.13** +/- **0.01** VL: **0.59** +/- **0.01** VA: **83.37%** +/- **0.59%**
Random Crop + Horizontal flip (p=0.5)	TL: **0.20** +/- **0.04** VL: **0.57** +/- **0.01** BEST VA: **83.39%** +/- **0.32%**

Random Crop + Horizontal flip (p=0.5) + ColorJitter(0.2,0.2,0.2,0)	TL: **0.23** +/- **0.02** VL: **0.59** +/- **0.01** VA: **83.43%** +/- **0.31%**

Table 3: *TL=Training loss, VL=Validation loss, VA=Validation accuracy. Means and standard deviations are computed over three runs.*

Although data augmentation has not led to significant improvements, when combining a random square crop of size 224 to the initial image resized to 256 and a random horizontal flip the model slightly decreased the loss function on the validation set, on average. Also note that performing data augmentation increased the computation time taken to compute the final model by about **+33%**. At the same time, the model found learning the training set a little harder, resulting in higher losses on the training set with the same number of epochs.

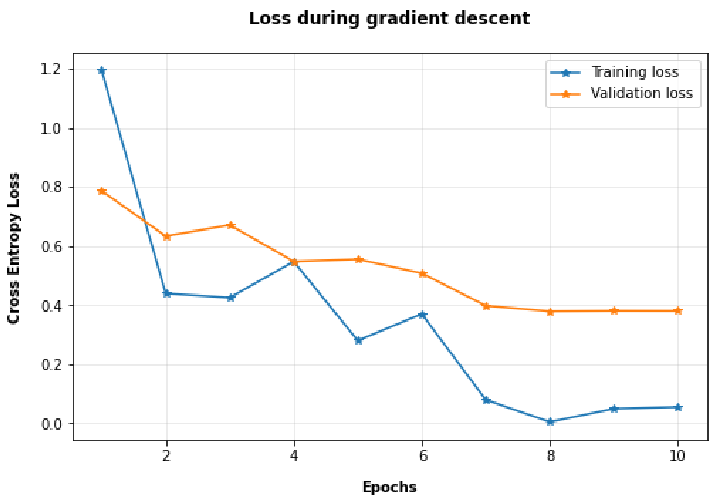
Thus, the best model in terms of lowest validation loss in *table 3* was rerun for 30 epochs, giving it more time to learn the augmented dataset, and reached an overall lowest validation loss of **0.56**.

> This new best model has been chosen for a **final evaluation** on the test set and obtained **84.32% accuracy.**

6. (Extra) Beyond AlexNet

More recent models than AlexNet are also available on the PyTorch documentation ([7]). These could be more complex networks like VGG-11, ResNet-18 and their variants. Previous model have also been pretrained on ImageNet, and reached higher top-1 and top-5 scores w.r.t. AlexNet.

In particular, with the previous best configuration of hyperparameters and data augmentation, the **VGG-11** pretrained model has been tried out, but required too much memory at runtime. Though, finetuning the layers with a smaller batch size of 64 gave the following results:



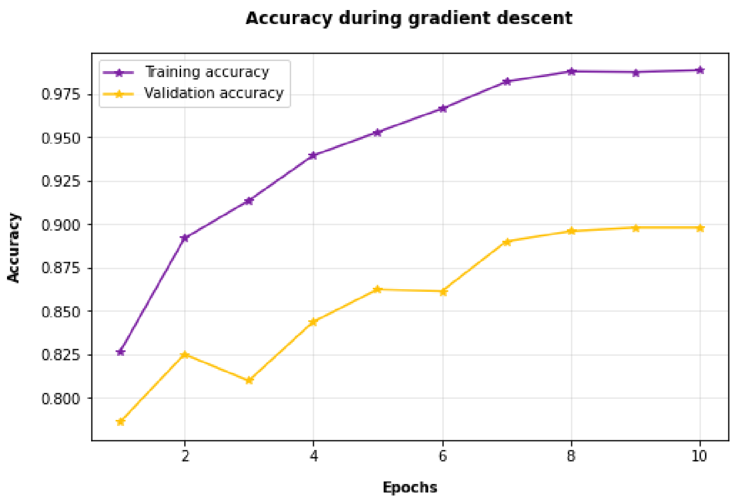


Figure 16

With a new best validation loss of **0.38** and validation accuracy of **89.80%**, the VGG-11 model obtained a final evaluation of **89.77%** on the test set.

Later, also the **ResNet-18** has been tried, with the same previous hyperparameters and batch size of 64:

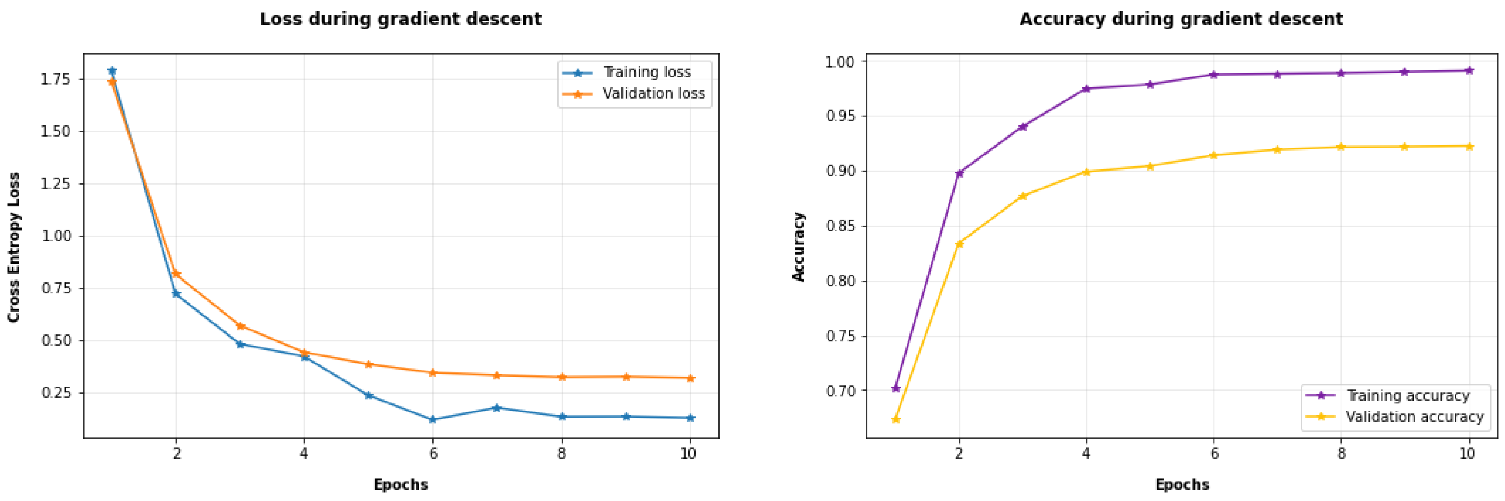


Figure 17: *ResNet-18 on Caltech-101*

As shown in *figure 17*, the pretrained ResNet-18 obtained the best overall results among all models tried in this work:

- Validation loss: **0.32**
- Validation accuracy: **92.25%**
- Final evaluation on the test set: **92.15%**

In conclusion, more recent models were able to significantly improve performances on all sides, reaching faster and better convergence on the training set and higher accuracies on the test set.

References

[1] Default weight initialization in PyTorch
https://discuss.pytorch.org/t/default-weight-initialization-vs-xavier-initialization/

[2] Cross Entropy Loss
https://en.wikipedia.org/wiki/Cross_entropy

[3] Optimizations of mini-batch SGD
https://ruder.io/optimizing-gradient-descent/

[4] Caltech-101 dataset
http://www.vision.caltech.edu/Image_Datasets/Caltech101/

[5] PyTorch transformations on images
https://pytorch.org/docs/stable/torchvision/transforms.html

[6] PyTorch source code of AlexNet
https://github.com/pytorch/vision/blob/master/torchvision/models/alexnet.py

[7] EXTRA - PyTorch pretrained models
https://pytorch.org/docs/stable/torchvision/models.html

Notes

[1] In general, it's mathematically wrong to infer the average output probability of the correct class from the given average loss function. In fact, the cross entropy loss is averaged among the samples in the current minibatch and, given that log(·) is a non-linear function, the average logarithms of output probabilities is in general different than the logarithm of the average probabilities. Thus, exponentiating the negative loss value obtained does not give back the true output average probability of the correct class for the given minibatch. However, the function ***$-log(p_i)$*** has an always-negative first-order derivative and an always-positive second-order derivative and this can be exploited to at least get a **lower bound** of the true average probability obtained during training, by computing ***$e^{-1*Loss}$***.
Furthermore, at the start of training when output probabilities are all really close to each other, the lower bound is approximately equal to the actual average output probability.

[2] **$e^{-4.55}$=0.01**. As stated in the previous note, this is in general just a lower bound for the true average output probability of the correct class in the current minibatch, but it is a good approximation at the early stages of the learning process. The true average was indeed **0.0107**.

[3] In order to get more precise evaluations, given the random batches and initialization, more runs should be performed and values should be reported with their corresponding averages and standard deviations.