Machine Learning and Deep Learning - 01TXFSM
Politecnico di Torino

Homework 2

Student: Riccardo Mereu Student ID: s277004

1 Introduction

The goal of this homework is to practice with PyTorch to train Convolutional Neural Networks for image classification. The architectures used are AlexNet, VGGNet and ResNet. The focus of the experiments is on the different techniques used to train the models. Beginning with a train from scratch approach and then using Transfer Learning an Data Augmentation to improve the results.

1.1 Dataset: Caltech101

Caltech101 is a dataset composed of pictures belonging to 101 different categories (plus a background category not used in this homework). There are 8677 images in total and each one is roughly 300x200 pixels. The dataset is relatively small to train a deep feedforward network from scratch.

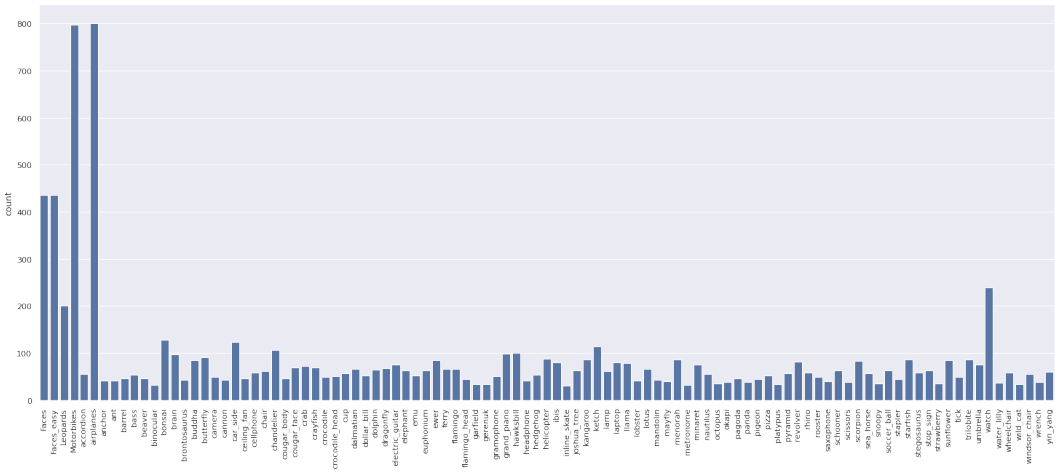

Figure 1: Caltech101 data distribution

The class distribution among the images shows that the dataset is highly imbalanced ranging from classes with 45 samples to others with 800 samples, like for instance the classes '`motorbikes`',

'airplanes' and the combination of 'faces' and 'faces_easy'. This means that after splitting the dataset into train set, validation set and test set, some categories end up with only 10 examples.

1.2 Data Preparation: Caltech class and DataLoaders

The images that compose the train and test set are specified in two distinct files. To handle the dataset PyTorch provides the abstract class `Dataset`. It is possible to define a custom class, called `Caltech` in the implementation, that inherits `Dataset` and overrides the `__len__` and `__getitem__` methods. `Caltech` takes as a parameter the file containing the images that compose a specific split and filters out the background category.
The training set provided in the file must be further divided into training set and validation set for hyperparameter tuning and model selection. In order to preserve the data distribution I used sklearn's `train_test_split()` function since it allows to sample the images in a stratified fashion from the original split. The original notebook used PyTorch `Subset` class, but using this solution one cannot specify different transformations to the two subsets as needed when using Data Augmentation. Then, I implemented a modified version of `Subset`, which directly applies the transformation to the images rather than depending on the underlying dataset class.
The other component of the pipeline needed to load the images to the network is `DataLoader` class. It is an iterator that can load the data batching and shuffling the samples and loads the data in parallel. The batch size which is one of the possible hyperparameters related to the training procedure is specified when the `DataLoader` is created.

2 Convolutional Neural Networks

The convolutional neural networks (CNN) used in this homework are AlexNet, VGG and ResNet. CNNs are a class of deep feed-forward neural networks built by repeated concatenation of five types of layers: convolutional (CONV), activation (ACT), pooling (POOL), fully-connected (FC) and classification (CLASS) [3].

The classification layer is usually implemented as a SoftMax function to obtain an output vector with each component in the range (0,1) and such that the componets sum up to 1. This function maps the non-normalized output to a probability distribution over the predicted output classes. However, this function does not appear explicitly in the models of `torchvision.model`. In fact, they trained these models using `nn.CrossEntropyLoss` which is a criterion that combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class. This means that PyTorch implementation applies to the outputs of the last layer the logarithm of the SoftMax function and then the Negative Log-Likelihood Loss (NLLLoss), to calculate the cross-entropy loss [1] [2].
AlexNet was proposed by Krizhevesky, Sutskever and Hinton in 2012 [7]. This architecture is a deeper and broader variant of the design introduced by Yann LeCun with the different LeNet architectures. It is composed of eight layers: five convolutional layers, two fully-connected hidden layers, and one fully connected layers, and one fully-connected output layer. After each convolutional layer, there is an activation layer which employs the ReLU function. Between some CONV layers are placed some pooling layers to perform a downsampling operation along the spatial dimensions.
Like AlexNet, the **VGG Network** [9] can be partitioned into two parts: the first composed of stacked VGG block modules and a second part composed of fully-connected layers. These VGG blocks are composed of a series of CONV, ACT and POOL layers. Based on the number of CONV and FC layers we have different types of VGG networks, the one used in this homework is VGG-16,

composed of thirteen CONV layers and three FC layers.
To keep increasing the depth of the network He et al. [6] came up with the idea of the residual layer, which allows every additional layer to contain the identity function as one of its elements. This design makes adding new layers to the network strictly more expressive and, as a result, **ResNet** is consistently deeper than AlexNet and VGGNet. The implementation used in this homework is ResNet50.
All these models were developed having in mind the ImageNet dataset that has 1000 classes. To use them in this homework, we have to modify the last layers of the networks to output a 101-dimensional vector corresponding to the 101 classes of our dataset. Another element that we need to define before training the networks is the loss function that will be used to measure the goodness of the performance. The loss function used is `CrossEntropyLoss`.

2.1 Optimizers and Learning Rate schedulers

To train a network is necessary to choose an optimizer and, optionally, a learning rate scheduler. PyTorch's `torch.optim` contains implementations of various optimization algorithms and learning rate schedulers. The optimizers used in this homework are SGD, Adam and AdamW, that are gradient-based optimization algorithms.

Stochastic Gradient Descent (SGD) is the most common optimization algorithm used in deep learning. It performs an update of the weight of the network for each mini-batch. SGD has had different improvements like momentum and Nesterov accelerated gradient. Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations that the standard algorithm has navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. Nesterov accelerated gradient (NAG) is a way to give our momentum term an approximation of the next position of the parameters, i.e. a rough idea where our parameters are going to be. It is used to look ahead by calculating the gradient not with respect to our current parameters, but with respect to the approximate future position of our parameters. Momentum and NAG are both available in the PyTorch implementation of the SGD optimizer (`torch.optim.SGD`).

SGD applies the same learning rate to all parameters updates. There are other optimization algorithms that try to address the idea of adapting the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. Adam and its variant AdamW are adaptive optimization algorithms. Adam stands for Adaptive Moment Estimation, which mixes the idea of momentum and the idea of storing exponentially decaying average of past squared gradients, like Adadelta and RMSprop. A paper from 2017 [8], pointed out that Adam's weight decay was implemented in most libraries was wrong and proposed a possible fix that they called AdamW. With more complex optimization algorithm the number of hyperparameters to tune increases and it is more difficult to tune them properly.

The learning rate of the optimization algorithm is often the single most important hyperparameter to tune [4]. However, choosing a proper learning rate can be difficult. If it is too small leads to slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge. To achieve even better results have been proposed many types of learning rate schedules that try to adjust the learning rate during training. Some of them rely on annealing, i.e. reducing the learning rate according to a predefined schedule or when the change in objective between epochs falls below a threshold. Like the initial learning rate, these schedulers have to be defined in advance and are thus unable to adapt to a dataset's characteristics. PyTorch's `torch.optim.lr_scheduler` offers many learning rates

schedulers' implementations, in this homework `StepLR` is used.

3 Experiments and results

This section contains a description of the different experiments conducted in this homework. The first part is about the training from-scratch of AlexNet by using different hyperparameters, optimizers and learning rate schedulers. The following two subsections are respectively about Transfer Learning and Data Augmentation using the pre-trained AlexNet model. The last section focuses on the use of VGGNet and ResNet.

3.1 Training from scratch

The first request consists of training AlexNet using the hyperparameters provided with the initial notebook template. The initial model uses the SGD optimizer (`optim.SGD`)with an initial learning rate of 10^{-3} and momentum of 0.9. The learning rate is adjusted using the StepLR scheduler, which after `step_size` iterations multiply the current learning rate `gamma`. The value used for the `step_size` is 20 iterations while the one used for `gamma` is 0.1.
The learning curve obtained with these hyperparameters is showed in Figure 2. In the first iterations, the loss function of the network is around 4.61. This value is equal to the negative logarithm of the probability of randomly picking a class out of the 101 possible ones. That means that the model is outputting the labels by chance as expected with the randomly initialized weights. Then, the network starts slowly decreasing its loss, but after 30 epochs it reaches only a minimum validation loss value of 4.54 and an accuracy of 0.09 on both validation and test set. Looking at the loss trend it is easy to spot that, after 20 epochs, the scheduler modified the learning rate because the decreasing rate of the loss changes drastically.

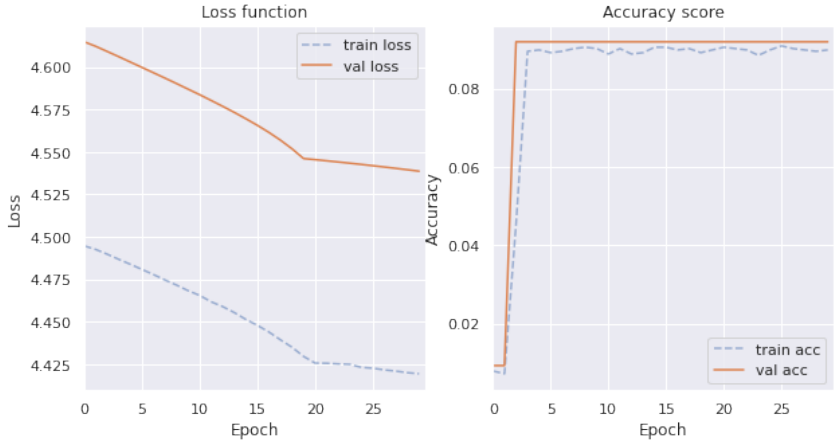

Figure 2: Loss function and accuracy scores obtained with default hyperparameters

3.1.1 Different Learning Rates

The easiest method to tune the hyperparameters is to try to obtain better results with a trial and error approach. This method is very time-consuming but it allows us to observe how the change in the hyperparameters affects the training procedure.

The first hyperparameter tuned was the learning rate. The initial learning rate is often the single most important hyperparameter and one should always make sure that is has been tuned. To identify the magnitude of its optimal value, the models were trained using different learning rates in the range $[1 \cdot 10^{-3}, 5 \cdot 10^{-1}]$ with SGD for 30 epochs, without a learning rate scheduler and momentum equal to 0.9.

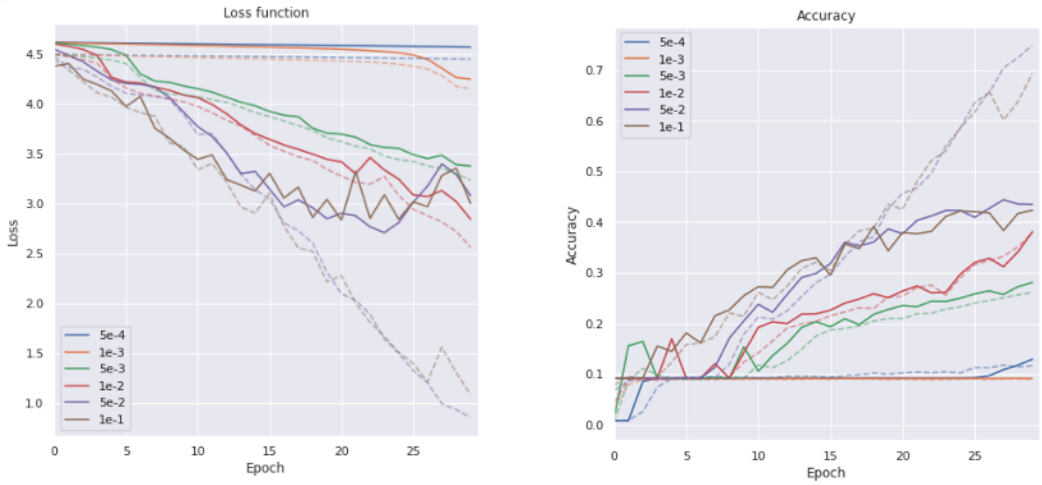

Figure 3: Loss and accuracy evolution using SGD with different learning rates

Figure 3 contains the losses and accuracy evolution of these models during training. It is clear that for lower learning rates values, like $1 \cdot 10^{-3}$, the model has a low convergence rate, while for high values, like $5 \cdot 10^{-3}$, the model diverges. With a learning rate just right, that in this case means around $5 \cdot 10^{-2}$, the model can achieve low loss function values and higher accuracy as a consequence.

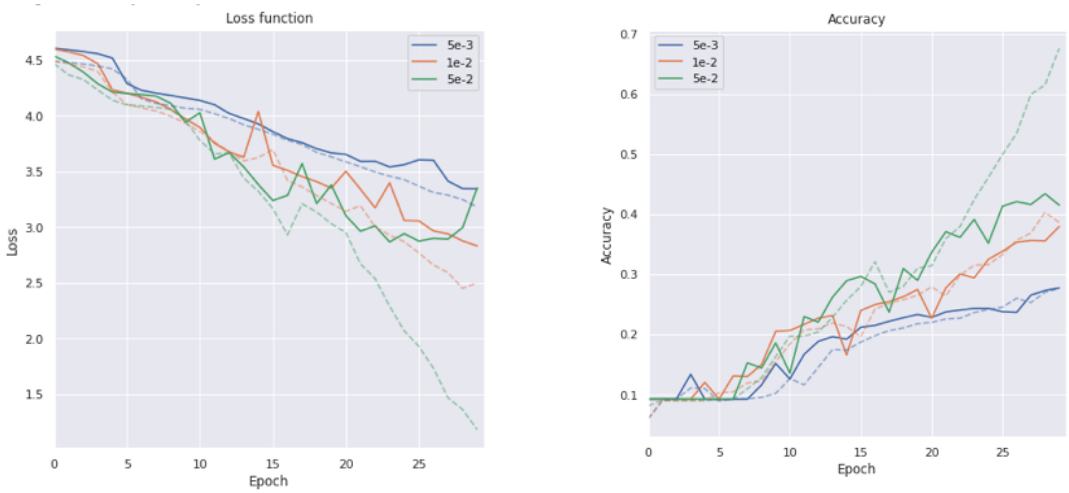

Figure 4: Loss and accuracy evolution using SGD with different learning rates

The same analysis on the learning rate was repeated using a different optimizer algorithm, SGD with Nesterov Accelerated Gradient (NAG). Three different models were trained using learning rates $\{5 \cdot 10^{-3}, 1 \cdot 10^{-2}, 5 \cdot 10^{-2}\}$ with NAG. We can see from table 1 the results of the experiments conducted at this point summarized. We can see that the addition of NAG does not help the model to reach a lower validation loss.

Optimizer	Learning rate	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
SGD	$1 \cdot 10^{-3}$	4,3462	0,0889	4,2457	0,0920	0,0919
	$5 \cdot 10^{-3}$	3,2350	0,2614	3,3757	0,2808	0,2786
	$1 \cdot 10^{-2}$	2,5627	0,3790	2,8440	0,3807	0,3757
	$5 \cdot 10^{-2}$	1,6409	0,5505	2,7077	0,4229	0,4227
	$1 \cdot 10^{-1}$	2,2763	0,4253	2,8344	0,3793	0,3951
NAG	$5 \cdot 10^{-3}$	3.1804	0.2763	3.3439	0.2773	0.2821
	$1 \cdot 10^{-2}$	2.4983	0.3862	2.8294	0.3793	0.3757
	$5 \cdot 10^{-2}$	2.2888	0.4243	2.8662	0.3911	0.3906

Table 1: SGD and NAG with different learning rates

The best network trained with these hyperparameters achieves train loss of 1.6409, validation loss of 2.7077 and accuracies of 55.05% and 42.29% respectively on train and validation set. Comparing these results with the one obtained with the default hyperparameters there is a big gap. However, after 30 epochs of training the model starts overfitting, it can be seen clearly from the increasing difference between the two losses.

As stated in Section 2.1, there exist other optimization algorithms with different characteristics than SGD, to understand that these algorithms can help to obtain better results training the model from scratch, Adam and AdamW were tried.
The first experiment was to test Adam with different learning rates in the range $[1 \cdot 10^{-4}, 5 \cdot 10^{-3}]$ and weight decay equal to $5 \cdot 10^{-5}$. Figure 5 contains the losses of these models. The networks after XX iterations start to overfit on the training set and this is particularly evident looking at the u-shaped validation loss. However, the use of Adam leads to a lower number of epochs to reach the same accuracy on the validation set compared to SGD.
The same experiment was conducted with AdamW. The use of this particular algorithm should provide some differences related to the weight decay implementation, but, in this case, these differences are not so evident. In table 2 are reported the results obtained with both algorithms varying the LR.

With these optimizers, the network after a few epochs tends to overfit the training set without any particular benefit for the validation accuracy. Adding an early stopping policy in the training function, it is possible to avoid to wait for many training epochs without getting any improvement. With this technique, two additional parameters should be provided to the training function, the number of epochs without a validation loss improvement, called patience, and the threshold that is the minimum change in the loss to qualify it an improvement.

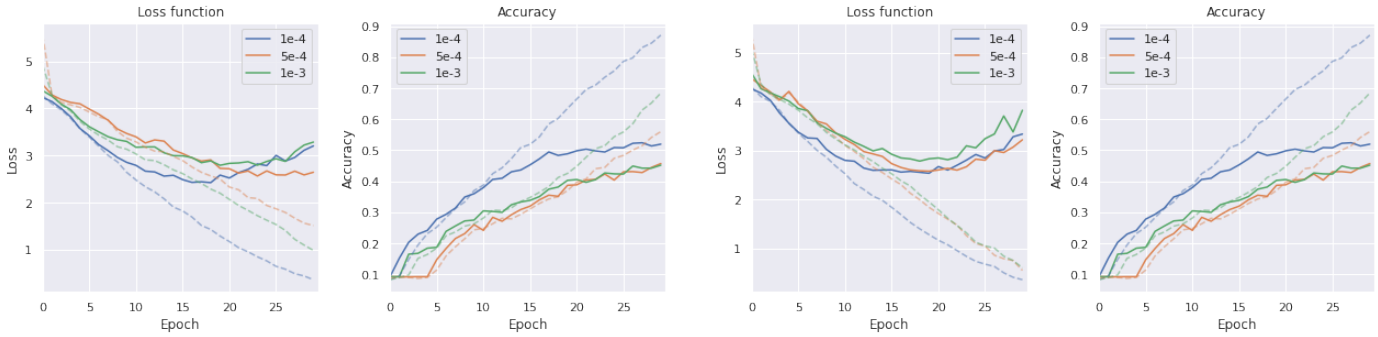

Figure 5: Loss and accuracy evolution using Adam (on the left) and AdamW (on the right) with different learning rates

Optimizer	Learning rate	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
Adam	$1 \cdot 10^{-4}$	1.6913	0.5432	2.4225	0.4834	0.4898
	$5 \cdot 10^{-4}$	2.0869	0.4447	2.5637	0.4232	0.4245
	$1 \cdot 10^{-3}$	2.1990	0.4253	2.7917	0.4028	0.4089
	$5 \cdot 10^{-3}$	2.6257	0.3524	3.1493	0.3243	0.3336
AdamW	$1 \cdot 10^{-4}$	1.2864	0.6255	2.5409	0.4910	0.4943
	$5 \cdot 10^{-4}$	1.8450	0.4858	2.5840	0.4374	0.4431
	$1 \cdot 10^{-3}$	2.1018	0.4454	2.7900	0.4073	0.4048
	$5 \cdot 10^{-3}$	4.0846	0.0826	4.1869	0.0923	0.0919

Table 2: Adam and AdamW with different learning rates

3.1.2 Weight Decay

The last models trained after a small number of epochs start to overfit on the training set. To try to reduce this effect and understand how to tweak the optimization hyperparameters to regularize the model, the next hyperparameter consider is the weight decay of the optimizers. They implement one of the most common kinds of parameter norm penalty the L^2 parameter norm penalty. his regularization technique drives the weights of the model closer to the origin by adding a regularization term $\Omega(\theta) = \frac{1}{2}\|\mathbf{w}\|_2^2$ to the objective function. The new loss to optimize is the following function, note that the bias term is omitted,

$$\tilde{\mathcal{L}}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y}) + \frac{\alpha}{2}\|\mathbf{w}\|_2^2 \tag{1}$$

with correstponding parameter gradient

$$\nabla_{\mathbf{w}} \tilde{\mathcal{L}}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y}) + \alpha \mathbf{w} \tag{2}$$

Applyng our gradient descent algorithm for optimization using $\tilde{\mathcal{L}}$, a different update rule for the weights $\mathbf{w}$ is applied

$$\mathbf{w} \leftarrow (1 - \eta\alpha)\mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}; \mathbf{X}, \mathbf{y}) \tag{3}$$

The addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector on each step, before the usual gradient update.
The following experiments consist of trying different values for the weight decay hyperparameter with the various optimizers considered in the analysis. A higher weight decay will slow the convergence of the algorithm since the tuned hyperparameter α weights the importance of the regularization in the objective function. The values tried were $\{5 \cdot 10^{-1}, 5 \cdot 10^{-3}, 5 \cdot 10^{-5}\}$.

Optimizer	Learning rate	Weight Decay	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
SGD	$1 \cdot 10^{-2}$	$5 \cdot 10^{-1}$	4.4575	0.0902	4.5781	0.0923	0.0919
		$5 \cdot 10^{-3}$	2.5772	0.3728	2.9261	0.3731	0.3823
		$5 \cdot 10^{-5}$	2.3597	0.4232	2.8239	0.3721	0.3923
	$5 \cdot 10^{-2}$	$5 \cdot 10^{-1}$	4.4524	0.0878	4.5701	0.0920	0.0919
		$5 \cdot 10^{-3}$	2.3246	0.4229	3.0116	0.3925	0.4079
		$5 \cdot 10^{-5}$	1.7784	0.5311	2.6818	0.4232	0.4300
NAG	1e-2	$5 \cdot 10^{-1}$	4.4579	0.0899	4.5777	0.0923	0.0919
		$5 \cdot 10^{-3}$	2.7936	0.3347	3.0934	0.3285	0.3394
		$5 \cdot 10^{-5}$	2.4970	0.3938	2.8068	0.3683	0.3771
	5e-2	$5 \cdot 10^{-1}$	4.4512	0.0871	4.5666	0.0920	0.0919
		$5 \cdot 10^{-3}$	2.6166	0.3562	3.0226	0.3378	0.3391
		$5 \cdot 10^{-5}$	1.2202	0.6449	2.5832	0.4599	0.4608

Table 3: SGD + NAG Weight decay

Both best results obtained with SGD and SGD with NAG use a lower weight decay value of $5 \cdot 10^{-5}$ and learning rate of $5 \cdot 10^{-2}$. SGD reaches a validation loss of 2.6818 and 42.32% and 43.00% of accuracy on validation and test set. The addition of NAG leads to a validation loss of 2.5832 and accuracies 45.00% for the validation set and 46.08% on the test set.

Optimizer	Learning rate	Weight Decay	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
Adam	$1 \cdot 10^{-4}$	$5 \cdot 10^{-1}$	4.4835	0.0892	4.6043	0.0920	0.0919
		$5 \cdot 10^{-3}$	1.7593	0.5384	2.5061	0.4447	0.4615
		$5 \cdot 10^{-5}$	1.5277	0.5840	2.5042	0.4765	0.4895
AdamW	$1 \cdot 10^{-4}$	$5 \cdot 10^{-1}$	1.3889	0.6141	2.3641	0.4934	0.5095
		$5 \cdot 10^{-3}$	1.5826	0.5726	2.4441	0.4810	0.4967
		$5 \cdot 10^{-5}$	1.5251	0.5813	2.4338	0.4703	0.4825

Table 4: Adam + AdamW Weight decay

Adam like SGD obtains the best validation loss with weight decay equal to $5 \cdot 10^{-5}$, on the contrary AdamW prefers a higher weight decay of $5 \cdot 10^{-1}$. Note that with AdamW the model reaches the best validation loss using this value while the convergence of Adam for such value is very slow.

3.1.3 Learning Rate Scheduler

In this part of the homework the model was trained using the `pytorch.optimi.StepLR` scheduler with different hyperparameters. Learning rate schedulers seek to adjust the learning rate during training by reducing the learning rate according to a predefined schedule. `StepLR` takes as parameter `step_size` and `gamma`. It decays the learning rate of each parameter group by 'gamma' every `step_size` epochs.

Optimizer	Step size	Gamma	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
SGD (LR=5e-2, WD=5e-5)	1	0.5	4.0630	0.0972	4.1714	0.1048	0.1054
	1	0.9	3.2085	0.2718	3.4901	0.2642	0.2589
	10	0.1	2.9956	0.3126	3.3410	0.3033	0.3021
	10	0.5	2.1529	0.4530	2.8212	0.3776	0.3858
	15	0.1	2.4086	0.4035	2.8152	0.3938	0.3913
	15	0.5	1.3737	0.6086	2.4722	0.4761	0.4808
	15	0.7	2.1619	0.4481	2.9132	0.3731	0.3844
AdamW (LR=1e-4, WD=5e-3)	1	0.5	4.0344	0.1407	4.1456	0.1463	0.1441
	1	0.9	2.3125	0.4208	2.7160	0.3959	0.4013
	10	0.1	2.5785	0.3734	2.8828	0.3634	0.3643
	10	0.5	1.4763	0.6027	2.4375	0.4848	0.4888
	15	0.1	1.8218	0.5270	2.4940	0.4537	0.4566
	15	0.5	1.4175	0.6082	2.4084	0.4945	0.4946
	15	0.7	1.2982	0.6366	2.4105	0.4896	0.5054

Table 5: Results obtained traning the model with different hyperparameters for StepLR Scheduler

Table 5 contains the results of the various experiments, inspecting the results it is evident that a rapid decay obtained with short step size and a low value of gamma leads to higher loss. To achieve a slower decay it is possible to use longer step sizes and keep a low value for `gamma`, or to increase the value of `gamma` using shorter step sizes.

The scheduler plays an important role for the SGD optimizer, since this algorithm otherwise will keep the learning rate fixed for the entire training. Instead, AdamW computes internally its adaptive learning rates for different groups of parameters. Looking at the results in Table 5, we can clearly see that AdamW is able to achieve better results than SGD using `StepLR`. However, a lower validation loss was achieved without a learning rate scheduler in Section 3.1.2 (see Table 4).

3.1.4 Batch Size

The batch size (or better mini-batch size) is an hyperparameter typically chosen between 1 and few hundreds that defines the number of each mini-batch that will be used by the optimizer to approximate the gradient of the model and then update its weights. The impact of this hyperparameter is mostly computational, using a larger batch size yield faster computation but requires a larger number of epochs to reach the same loss, since the batch size is strictly related to the number of weights update per epoch. Training with a small batch size might require a smaller learning rate to maintain stability because of the high variance in the estimate of the gradient. Using a small batch size leads to noisier gradient estimates. Table 6 and Figure 6 contain the results obtained varying

the batch size and training the network with AdamW using the hyperparameters found in Section 3.1.3 ($lr = 1 \cdot 10^{-4}$, $wd = 5 \cdot 10^{-3}$, `step_size` = 15, `gamma` = 0.5).

Batch Size	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
32	1.3194	0.6463	2.3209	0.5028	0.4981
64	1.3262	0.6518	2.2465	0.5207	0.5216
128	1.1810	0.6580	2.2929	0.5225	0.5171
256	1.3737	0.6086	2.4722	0.4761	0.4808
512	1.9676	0.4201	2.6037	0.4170	0.4124

Table 6: Experiments with SGD ($lr = 1 \cdot 10^{-4}$, $wd = 5 \cdot 10^{-3}$, `step_size` = 15, `gamma` = 0.5) using different batch sizes

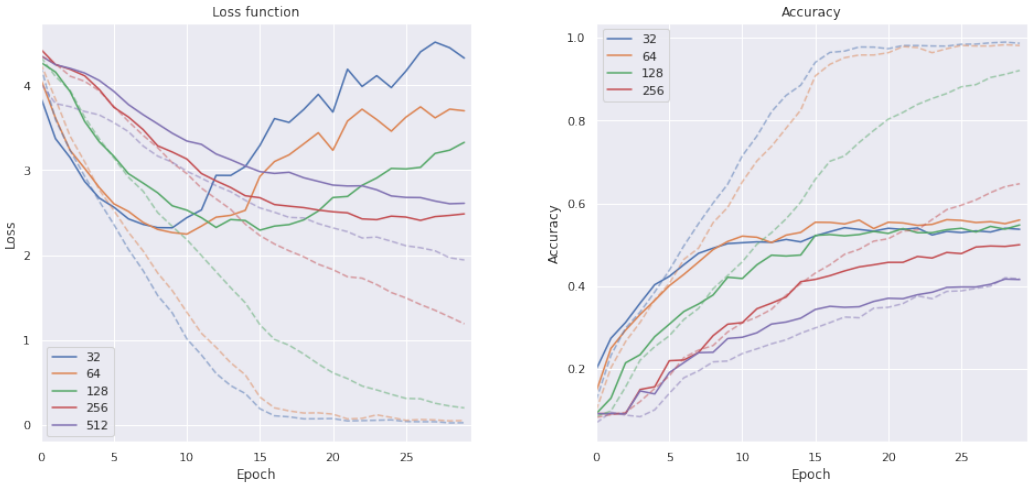

Figure 6: Loss and accuracy evolution using AdamW with different mini-batch sizes

A batch size B means that B samples from the training set will be used to estimate the loss gradient with respect to the weights of the model. Before updating the model weights, during one epoch the algorithm makes one pass through the whole training dataset. With a lower B the train loss decrease faster, but after few epochs the validation loss increases until the model starts overfitting. Increasing the batch size means having a slower decrease in the train loss and a more precise estimates of the gradient, leading to less overfitting and a convergence to a more stable model.

3.2 Transfer Learning

Training a feedforward neural network from scratch without a large amount of data or with data with imbalanced class distribution is a difficult task and often leads to unsatisfactory results, as happened with the experiments described in the previous section. Conventional machine learning and deep learning algorithms have been traditionally designed to work in isolation, i.e. in a scenario

where training and test data are drawn from the same feature space and the same distribution. When the distribution changes, like in this case where models designed for ImageNet dataset are used on Caltech101 dataset, it would be nice to overcome this isolated training paradigm and to use the knowledge acquired for one task to solve related ones. This learning paradigm is called Transfer Learning and a possible formal and general definition from [10] is the following: given a source domain $\mathcal{D}_S$ and a learning task T_S a target domain $\mathcal{D}_T$ and learning task $\mathcal{T}_T$, transfer learning aims to improve the learning of the target predictive function $f_T(\cdot)$ in $\mathcal{D}_T$ using the knowledge in $\mathcal{D}_S$ and $\mathcal{T}_S$, where $\mathcal{D}_S \neq \mathcal{D}_T$, or $\mathcal{T}_S \neq \mathcal{T}_T$.
It is possible to categorize transfer learning in different sub-settings, but the focus of this homework is only on *inductive transfer learning* because the aim is to help improve the learning of the target predictive function $f_T(\cdot)$ in $\mathcal{D}_T$ when $\mathcal{T}_S \neq \mathcal{T}_T$. The objective of inductive transfer learning is to use inductive bias. It means to use the set of assumptions of the source training data, i.e. the weights learned by the model on ImageNet in this case, to help the learning on the target domain.

There are different ways to apply transfer learning to ConvNets, but the most common scenarios are two: fine-tuning the entire pre-trained model and using the frozen CONV layers as a fixed feature extractor. Moreover, the assignment also requires freezing the FC layers and training only the CONV layers. In this homework, to use the pre-trained AlexNet we need to modify the last FC layer of the original model replacing them with a new FC layer that outputs a 101-dimensional vector. Since the pre-trained model was trained on ImageNet, it expects to receive as input images appropriately preprocessed as during training on the original dataset. For this reason, the mean and the standard deviation of ImageNet were used for the normalization function of the `DataLoader`.

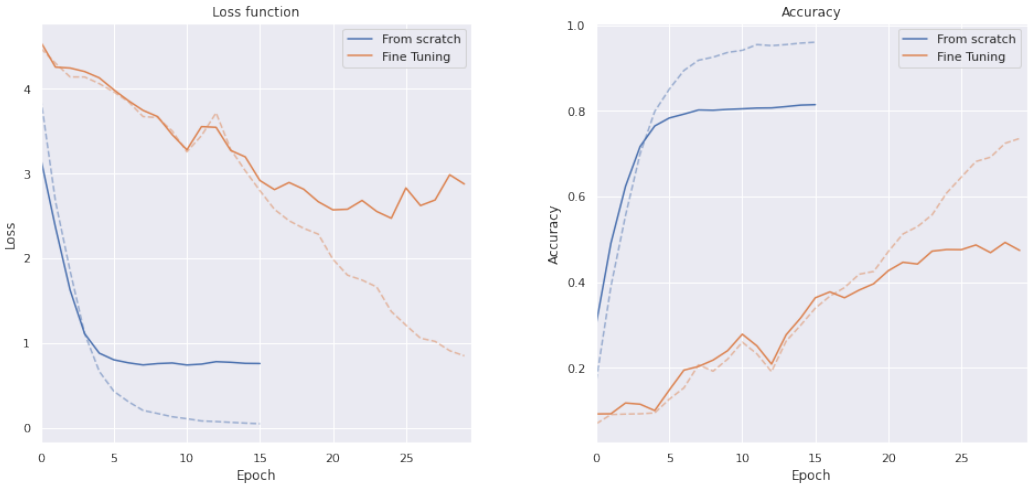

Figure 7: Comparison of the learning curves and accuracies of two models trained from scratch and using Fine Tuning (both using AdamW, $lr = 1 \cdot 10^{-4}$ and $wd = 5 \cdot 10^{-5}$)

The following sections contain the results obtained using transfer learning with an Alexnet pre-trained model. Each subsection delves into the details related to the three techniques applied. Then for each of them, the various experiments were conducted using as optimizers SGD, SGD with NAG and AdamW. For each optimizer, different values of learning rate and weight decay were used.

3.2.1 Fine Tuning

The fine-tuning approach is to update the pre-trained weights by backpropagation using the new dataset. Since the weights of the network should be a good starting point compared to the random-initialized weights of the from-scratch training, it could be a good idea to use a smaller learning rate.

Optimizer	Learning rate	Weight Decay	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
SGD	1e-3	0.005	0.1219	0.9391	0.7063	0.8167	0.8220
		0.0005	0.1219	0.9391	0.7063	0.8167	0.8220
		0.00005	0.1399	0.9336	0.6822	0.8129	0.8161
	5e-3	0.005	0.2350	0.9011	0.6699	0.8223	0.8313
		0.0005	0.1441	0.9326	0.6397	0.8330	0.8272
		0.00005	0.1142	0.9409	0.6567	0.8281	0.8317
	1e-2	0.005	0.0228	0.9678	0.7319	0.8292	0.8244
		0.0005	0.3661	0.8707	0.6968	0.8095	0.8165
		0.00005	0.0982	0.9429	0.7181	0.8205	0.8258
NAG	1e-3	0.005	0.1105	0.9416	0.7434	0.8050	0.8168
		0.0005	0.2086	0.9118	0.7209	0.8105	0.8102
		0.00005	0.1598	0.9308	0.7218	0.8185	0.8178
	5e-3	0.005	0.1645	0.9253	0.6629	0.8302	0.8216
		0.0005	0.1547	0.9319	0.6785	0.8192	0.8292
		0.00005	0.3266	0.8748	0.6633	0.8247	0.8168
	1e-2	0.005	0.2222	0.9073	0.6811	0.8202	0.8220
		0.0005	0.2455	0.9021	0.6836	0.8178	0.8178
		0.00005	0.2350	0.9049	0.6703	0.8223	0.8178
AdamW	1e-5	0.005	0.1537	0.9340	0.8618	0.7763	0.7798
		0.0005	0.1453	0.9398	0.8495	0.7728	0.7746
		0.00005	0.1678	0.9288	0.8468	0.7839	0.7788
	5e-5	0.005	0.0360	0.9647	0.7498	0.8164	0.8102
		0.0005	0.0357	0.9661	0.7452	0.8247	0.8137
		0.00005	0.0986	0.9499	0.7694	0.8112	0.8095
	1e-4	0.005	0.2997	0.8817	0.8129	0.8015	0.8016
		0.0005	0.0755	0.9506	0.7795	0.8185	0.8161
		0.00005	0.0872	0.9464	0.7680	0.8115	0.7988

Table 7: Fine Tuning

Table 7 contains the results obtained with different optimizers and their specific hyperparameters. As we can see the Fine Tuning requires a lower learning rate to achieve the best results. The best result overall was achieved with SGD with learning rate equals to $5 \cdot 10^{-3}$ and weight decay equal to $5 \cdot 10^{-4}$.

3.2.2 Convolutional Layers Frozen

The second experiment is with CONV layers frozen. In this setting, the filters learned on ImageNet are applied on Caltech101, avoiding the learning procedure on their parameters. On the other hand, the FC layers of the pre-trained model should be fine-tuned on Caltech101.

Optimizer	Learning rate	Weight Decay	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
SGD	1e-3	0.005	0.1521	0.9550	0.6255	0.8295	0.8272
		0.0005	0.1427	0.9589	0.6229	0.8302	0.8348
		0.00005	0.1530	0.9550	0.6277	0.8302	0.8337
	5e-3	0.005	0.0338	0.9689	0.5756	0.8368	0.8458
		0.0005	0.0447	0.9661	0.5792	0.8402	0.8469
		0.00005	0.0583	0.9668	0.5848	0.8375	0.8455
	1e-2	0.005	0.0305	0.9665	0.5741	0.8427	0.8431
		0.0005	0.0291	0.9685	0.5930	0.8406	0.8420
		0.00005	0.0214	0.9692	0.5841	0.8434	0.8441
NAG	1e-3	0.005	0.1450	0.9575	0.6131	0.8385	0.8368
		0.0005	0.1497	0.9568	0.6173	0.8313	0.8362
		0.00005	0.1483	0.9582	0.6232	0.8302	0.8355
	5e-3	0.005	0.0299	0.9710	0.5655	0.8413	0.8438
		0.0005	0.0403	0.9682	0.5740	0.8413	0.8493
		0.00005	0.0307	0.9692	0.5684	0.8375	0.8441
	1e-2	0.005	0.0428	0.9654	0.5917	0.8347	0.8424
		0.0005	0.0319	0.9672	0.5936	0.8385	0.8448
		0.00005	0.1211	0.9385	0.7192	0.8192	0.8216
AdamW	5e-5	0.005	0.0171	0.9710	0.5943	0.8409	0.8410
		0.0005	0.0169	0.9716	0.6102	0.8399	0.8372
		0.00005	0.0156	0.9713	0.6175	0.8406	0.8417
	1e-4	0.005	0.0278	0.9678	0.6080	0.8375	0.8358
		0.0005	0.0244	0.9699	0.6204	0.8320	0.8386
		0.00005	0.0355	0.9685	0.6019	0.8382	0.8344
	5e-4	0.005	0.0226	0.9692	0.6442	0.8430	0.8438
		0.0005	0.0064	0.9716	0.6186	0.8593	0.8617
		0.00005	0.1112	0.9423	0.6213	0.8275	0.8320

Table 8: Results obtained with frozen CONV layers

The results obtained applying the three optimization algorithms are in Table 8. The lower number of parameters to train leads to a lower training time. The model uses as a feature extractor the lower layers of the network trained on ImageNet. Inspecting the results, we can see that the model trained with SGD with NAG reached the lowest validation loss of 0.5655 and accuracies of 84.13% on validation set and 84.38% on test set.

3.2.3 Fully Connected Layers Frozen

The last experiment is with the FC layers frozen. The modification of the last FC layer is highly important when considering this scenario. This new FC layer is randomly initialized and since is a new instance of `torch.nn.Linear` has `requires_grad = True`. Two possible choices can be evaluated, to freeze this new random-initialized layer or to train it with the CONV layers.
If the last FC layer is frozen, then the network is not able to learn anything and the loss is stuck, see results in Table 9. Without freezing the last layer the random weights can be updated and the network is able to learn compared to the previous case reaching even good results.

Optimizer	LR	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy	Test Accuracy	Model
SGD	$1 \cdot 10^{-3}$	3.0436	0.3226	3.1123	0.3530	0.3543	All FC frozen
	$5 \cdot 10^{-3}$	1.8079	0.5422	2.3716	0.4903	0.4877	
	$1 \cdot 10^{-2}$	1.4010	0.6373	2.0914	0.5481	0.5451	
AdamW	$5 \cdot 10^{-5}$	2.2659	0.4703	2.5669	0.4582	0.4587	
	$1 \cdot 10^{-4}$	1.5245	0.6155	2.3080	0.5138	0.5071	
	$5 \cdot 10^{-4}$	0.5100	0.8472	1.8777	0.6010	0.6108	
SGD	$1 \cdot 10^{-3}$	0.1708	0.9267	0.7721	0.7963	0.8061	All FC frozen except FC[6]
	$5 \cdot 10^{-3}$	0.2565	0.8994	0.7301	0.8088	0.8199	
	$1 \cdot 10^{-2}$	0.3887	0.8582	0.7696	0.8046	0.8082	
AdamW	$5 \cdot 10^{-5}$	0.1141	0.9429	0.9262	0.7728	0.7625	
	$1 \cdot 10^{-4}$	0.2722	0.8932	0.8890	0.7652	0.7636	
	$5 \cdot 10^{-4}$	0.2503	0.8959	0.8629	0.7884	0.7857	

Table 9: Results obtained with frozen FC layers

By freezing the FC layers the training process is updating the weights that the model learnt on ImageNet. They are adapted to the new input data but keeping the weights of the classifier fixed (at least up to FC[6]). This is not the optimal choice to exploit the previously acquired knowledge. Even with these limitations however the model trained in this way are able to achieve better results than training the network from scratch.

3.3 Data Augmentation

Machine learning models generalize better when trained on more data. However, in practice, the amount of data to train the model is limited. It is common to artificially increase the dataset size by applying label-preserving transformations to images. Images are high dimensional and include an enormous variety of factors of variation, that can be easily simulated. PyTorch offers a set of functions grouped under `torchvision.transforms` to perform image transformations. Geometric transformations, flipping, colour modification, cropping, rotation, noise injection and random erasing are used to augment image in deep learning. During dataset augmentation, one must be careful not to apply transformations that would change the correct class of the image.

In this section of the homework, three different setups of PyTorch's transformations are used. The transforms used are `RandomHorizontalFlip`, `ColorJitter` and `RandomRotation`. The first experiment exploits the combination of `RandomHorizontalFlip` and `ColorJitter`. The first one horizontally flips the given image randomly with a given probability p, while the second one randomly changes the brightness, contrast and saturation of an image.

Optimizer	**LR**	**WD**	**Train Loss**	**Train Accuracy**	**Valid. Loss**	**Valid. Accuracy**	**Test Accuracy**	**Transfer Learning**
SGD	$5 \cdot 10^{-3}$	$5 \cdot 10^{-4}$	0.2239	0.9039	0.6197	0.8413	0.8424	Fine Tuning
NAG	$5 \cdot 10^{-3}$	$5 \cdot 10^{-5}$	0.1825	0.9212	0.6343	0.8413	0.8355	
AdamW	$1 \cdot 10^{-4}$	$5 \cdot 10^{-5}$	0.0938	0.9457	0.6739	0.8389	0.8292	
SGD	$5 \cdot 10^{-3}$	$5 \cdot 10^{-3}$	0.0774	0.9613	0.5123	0.8593	0.8593	Feature Extractor
NAG	$1 \cdot 10^{-2}$	$5 \cdot 10^{-3}$	0.0736	0.9578	0.5157	0.8589	0.8593	
AdamW	$5 \cdot 10^{-5}$	$5 \cdot 10^{-3}$	0.0386	0.9685	0.5692	0.8479	0.8514	

Table 10: Results of Dataset Augmentation: Experiment 1

The second experiment makes use of `RandomHorizontalFlip` together with `RandomRotation`. This time the image can be flipped horizontally, but also rotate by an angle between zero degrees and `angle`, i.e. the parameter of the transform (in the experiment the `angle` = 30°).

Optimizer	**LR**	**WD**	**Train Loss**	**Train Accuracy**	**Valid. Loss**	**Valid. Accuracy**	**Test Accuracy**	**Transfer Learning**
SGD	$5 \cdot 10^{-3}$	$5 \cdot 10^{-4}$	0.2172	0.9080	0.6959	0.8278	0.8130	Fine Tuning
NAG	$5 \cdot 10^{-3}$	$5 \cdot 10^{-5}$	0.2170	0.9094	0.6836	0.8261	0.8234	
AdamW	$1 \cdot 10^{-4}$	$5 \cdot 10^{-5}$	0.1887	0.9160	0.7836	0.8136	0.8088	
SGD	$5 \cdot 10^{-3}$	$5 \cdot 10^{-3}$	0.0603	0.9627	0.5708	0.8458	0.8441	Feature Extractor
NAG	$1 \cdot 10^{-2}$	$5 \cdot 10^{-3}$	0.0997	0.9495	0.5784	0.8465	0.8355	
AdamW	$5 \cdot 10^{-5}$	$5 \cdot 10^{-3}$	0.1047	0.9533	0.6372	0.8299	0.8330	

Table 11: Results of Dataset Augmentation: Experiment 2

Optimizer	LR	WD	Train Loss	Train Accuracy	Valid. Loss	Valid. Accuracy	Test Accuracy	Transfer Learning
SGD	$5 \cdot 10^{-3}$	$5 \cdot 10^{-4}$	0.2611	0.8997	0.9875	0.7479	0.7532	Fine Tuning
NAG	$5 \cdot 10^{-3}$	$5 \cdot 10^{-5}$	0.2237	0.9077	0.9342	0.7628	0.7757	
AdamW	$1 \cdot 10^{-4}$	$5 \cdot 10^{-5}$	0.1274	0.9405	1.1845	0.7414	0.7535	
SGD	$5 \cdot 10^{-3}$	$5 \cdot 10^{-3}$	0.0909	0.9585	0.7186	0.8057	0.8158	Feature Extractor
NAG	$1 \cdot 10^{-2}$	$5 \cdot 10^{-3}$	0.1116	0.9506	0.7385	0.7984	0.8057	
AdamW	$5 \cdot 10^{-5}$	$5 \cdot 10^{-3}$	0.1105	0.9540	0.8509	0.7832	0.7819	

Table 12: Results of Dataset Augmentation: Experiment 3

The last experiment consists of using together at the same time all the previous transforms. For each experiment, the best models obtained using Fine Tuning and keeping the CONV layers fixed (Feature Extractor in the Tables), were trained again using dataset augmentation (DA). Increasing the amount of data, dataset augmentations should act as a regularizer and help reduce overfitting. The results of these experiments are reported in Tables 10, 11 and 12. There is only a slight change in the difference between training loss and validation loss. However, the test accuracy, i.e. the performance of the model on unseen data, is higher than the one obtained training the models with the same hyperparameters without DA.

3.4 Beyond AlexNet: VGG and ResNet

After having tried different techniques to train AlexNet, in the last section of this homework, we train two other architectures VGG16 and ResNet50. This section aims to investigate how the use of deeper and more expressive models impacts the results. The training of these models was conducted using a library for HPO (HyperParameter Optimization) called `hpbandster`, in particular, the BOHB [5] algorithm was used. BOHB combines Bayesian optimization (BO) and Hyperband (HB) optimization to explore the hyperparameter space and train the model. As in the previous sections, the maximum number of epochs is fixed to 30 and this time this value is also used as the budget for the HPO algorithm.

Tables 13 and 14 contain the results obtained training these models with Fine Tuning and keeping the CONV layers frozen. The training process makes use of the third DA configuration. The complete output of the HPO process can be found on GitHub [1]. As expected these networks perform better than AlexNet on Caltech101. VGG16 is able to reach a validation loss equal to 0.3025 with validation and test accuracy of 92.74% and 94.16% respectively, using Fine Tuning. While the deeper ResNet50 reaches a validation loss of 0.1864 and accuracies of 95.23% on the validation set and 95.85% on the test set.

[1] `https://github.com/rm-wu/homework_MLDL/tree/master/HW2`

Transfer Learning	Optimizer	LR	WD	Moment.	Train Loss	Train Acc.	Valid. Loss	Valid. Acc.	Test Acc.
Fine Tuning	SGD	1.33E-02	3.02E-05	8.659E-01	0.0042	0.9955	0.3025	0.9274	0.9416
	AdamW	4.50E-05	1.15E-06	-	0.0021	0.9959	0.3384	0.9163	0.9402
	SGD	1.55E-02	3.15E-05	8.665E-01	0.0444	0.9872	0.3686	0.9039	0.9202
Feature Extractor	AdamW	1.08E-04	1.37E-04	-	0.0131	0.9934	0.3798	0.8952	0.9336
	SGD	3.61E-03	3.45E-04	8.56E-01	0.0280	0.9910	0.3897	0.8900	0.9202
	SGD	1.15E-02	6.18E-05	6.32E-01	0.0302	0.9900	0.3936	0.8921	0.9212

Table 13: VGG16

Transfer Learning	Optimizer	LR	WD	Moment.	Train Loss	Train Acc.	Valid. Loss	Valid. Acc.	Test Acc.
Fine Tuning	SGD	4.61E-02	2.36E-04	1.40E-01	0.0038	0.9959	0.1864	0.9523	0.9585
	SGD	5.12E-02	1.61E-04	5.00E-02	0.0036	0.9959	0.1891	0.9526	0.9575
	AdamW	1.03E-04	1.60E-03	-	0.0063	0.9959	0.2372	0.9443	0.9520
Feature Extractor	AdamW	1.07E-03	8.63E-02	-	0.0180	0.9948	0.2347	0.9346	0.9329
	AdamW	1.21E-03	8.69E-02	-	0.0115	0.9955	0.2379	0.9326	0.9326
	AdamW	3.29E-03	2.77E-05	-	0.0074	0.9945	0.2504	0.9315	0.9305

Table 14: ResNet50

References

[1] URL: `https://pytorch.org/docs/master/generated/torch.nn.CrossEntropyLoss.html`.

[2] URL: `https://discuss.pytorch.org/t/torchvision-models-dont-have-softmax-layer/18071`.

[3] Andrej Karpathy, et. al. CS231n Convolutional Neural Networks for Visual Recognition notes. URL: `https://cs231n.github.io`.

[4] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. 2012. URL: `http://arxiv.org/abs/1206.5533`.

[5] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, 2018.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. 2012.

[8] Ilya Loshchilov and Frank Hutter. Fixing Weight Decay Regularization in Adam. 2017. URL: `http://arxiv.org/abs/1711.05101`.

[9] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*, 2015.

[10] Sinno Jialin Pan, Qiang Yang. A Survey on Transfer Learning. 2009. URL: `http://www.cse.ust.hk/~qyang/Docs/2009/tkde_transfer_learning.pdf`.