Assignment 2

183.663 Deep Learning for Visual Computing (2022S)

Group 9
Yu Kinoshita (01623806), Michael Köpf (01451815)

Part 1: Experimenting with Optimizers

Experimental Setup & Findings
In total, we experimented with *three different optimizer settings* (all without weight decay):

- SGD – learning rate 150, w/o momentum
- SGD – learning rate 150, Nesterov Momentum 0.9
- Adam – learning rate 2, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-8$

Our *stopping criterion* is based on the loss. In case the loss does not improve (i.e., decreases) for 50 consecutive epochs, the optimization stops. Additionally, we also stop the optimization after 3000 epochs. For `--eps` (ϵ for the calculation of the numerical gradient), we use a value of 2.0. The function evaluation at given points x_1, x_2 (`Fn.__call__`) does not use interpolation but simply rounds and maps to integers.

We evaluate every function with two different starting points. One starting point is chosen near to a local minimum or far away from a global minimum (and vice versa for the other starting point).

The experimental results are shown in Table 1.

Optimizer | Function	`beale` (10, 10)	`beale` (700, 500)	`camel3` (100, 800)	`camel3` (400, 400)	`camel6` (900, 500)	`camel6` (400, 400)	`eggholder` (500, 200)	`eggholder` (400, 800)	`madsen` (700, 200)	`madsen` (200, 600)	`schaffern4` (650, 400)	`schaffern4` (100, 100)	`tang` (900, 900)	`tang` (500, 125)
SGD	3000*	1	1330	1827	3000*	3000*	670	1514	3000*	1791	34	479	1299	3000*
SGD w/ Nest. Mom.	1223	142	580	595	460	475	113	181	2404	1641	72	80	239	667
Adam	1370	7	74	76	243	152	70	202	610	150	5	24	154	137

* max. number of epochs reached
... global minimum reached

Table 1: Number of optimization epochs for each optimizer–function–starting-point combination. Starting points are given as (row, column) in the OpenCV coordinate system. The starting point in the left column of each function is a point close to a local minimum/far away from a global minimum, the starting point in the right column is a point close to a global minimum/far away from a local minimum.

With our chosen parameters, none of the optimizers is able to find a global minimum when starting close to a local minimum/far away from a global minimum. `schaffern4` is the only function where none of the optimizers finds a global minimum. This can be explained due to the fact that this function has many local minima where the optimization easily can get stuck.

Overall, SGD without Nesterov Momentum shows the worse performance:

(1) It is the only optimizer that reaches the maximum number of epochs several times. On average, it has a slower convergence rate than the other two optimizers but yields worse results. This could be avoided by increasing the learning rate with the downside of possibly 'missing' a global minimum.

(2) It *never* finds a global minimum but gets stuck as soon as the gradient levels out.

Adam shows the best performance and finds a global minimum five out of seven times – given a starting point near a global minimum – in a very low number of epochs (compared to SGD and SGD with Nesterov Momentum). SGD with Nesterov Momentum shows a better performance than SGD but a worse one than Adam. However, SGD with Nesterov Momentum finds a global minimum for `eggholder` while Adam does not. Figure 1 shows an example of the optimization on `camel6` using our optimizer settings listed above. We can observe that Adam converges more 'directly' towards the global minimum than SGD and SGD with Nesterov Momentum.

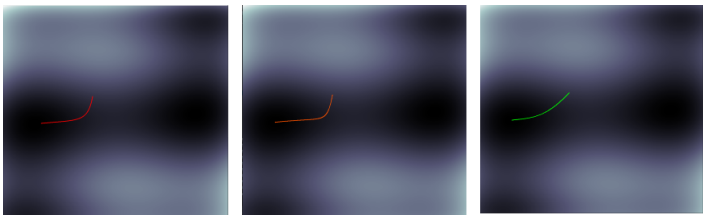

Figure 1: Optimization on `camel6` starting from (400, 400): SGD (red line), SGD with Nesterov Momentum (orange line) and Adam (green line).

How does gradient descent work?
Gradient descent is an iterative optimization algorithm used to minimize the loss function in order to maximize the classification performance on the training set. In every iteration the gradient in the direction of the greatest decrease of the loss function is computed and the parameters are updated according to the step size consisting of learning rate times the magnitude of the gradient. Possible limitations of gradient descent are its slow convergence rate and that it may show a bad performance on many functions (may only find a local minimum instead of a global one).

Part 2: Choosing an Architecture

Which network architecture did you choose for part 2 and why?
We evaluated two different network architectures:

(1) An adapted version of the network architecture for CIFAR-10 in the PyTorch tutorial[1] which we refer to as `BaselineCNN`. The adopted network has 28,666 trainable parameters

(2) Our own architecture which we refer to as `PerrosYGatosNet` (Engl. CatsAndDogsNet) which as 98,722 trainable parameters

In detail, the *adaptions in (1)* are as follows. We use two instead of three fully-connected layers in the rear part of the network. The first fully-connected layer has 64 units and uses ReLU as activation function. The second linear layer is the ouptut layer, and hence has two units. With this adaption we achieve a marginally better validation accuracy than with the proposed architecture in the tutorial.

The architecture of `PerrosYGatosNet` is shown in Figure 2. Our architecture re-uses the idea of the architecture presented in the lecture and makes use of two blocks in the front part of the network. Each block consists of two convolutional layers with identical layout followed by a pooling layer that reduces the data dimension. In the rear part, two fully-connected layers are used. A `ReLU` activation function is used after every convolutional layer and the fully-connected hidden layer. We chose this shallow architecture since it should be sufficient to achieve a reasonable classification accuracy for a classification problem that involves tiny images of only two classes.

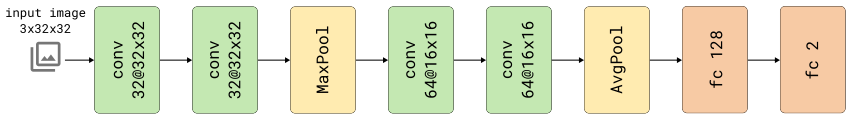

Figure 2: Conceptual network architecture of `PerrosYGatosNet`.

Additionally, we also implemented *per-channel normalization*. The code for calculating the statistics of the training set (mean and standard deviation) can be found in `calc_training_set_stats.py`. The corresponding function in `ops.py` is called `normalizePerChannel()` and takes two arguments – `mean` and `std`. The statistics of the training set are stored in a dictionary in `cnn_cats_and_dogs.py` and `cnn_cats_and_dogs_pt3.py`.

For the training of the neural networks we used SGD with a Nesterov Momentum of 0.9, a learning rate of $1e-2$, no weight decay and a batch size of 128. We trained for 100 epochs and shuffled the samples of the training set. The results for both network architectures with and without per-channel normalization, respectively, are shown in Table 2.

[1] `https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#define-a-convolutional-neural-network`, Accessed: 2022-05-19

Network Architecture	Generic Normalization			Per-Channel Normalization		
	Train Loss	Acc. Validation Set (Best)	Acc. Training Set	Train Loss	Acc. Validation Set (Best)	Acc. Training Set
`BaselineCNN`	0.001 ± 0.000	0.712 (epoch 19)	0.714	0.000 ± 0.000	0.713 (epoch 22)	0.708
`PerrosYGatosNet`	0.115 ± 0.033	0.804 (epoch 98)	0.788	0.033 ± 0.017	**0.810** (epoch 66)	**0.801**

Table 2: Evaluation of our two network architectures – with and without per-channel normalization. The column 'Train Loss' lists the loss of the last epoch.

We can observe that our `PerrosYGatosNet` is clearly superior to the `BaselineCNN`; the highest achieved accuracy on the validation set is 81.0% – which is higher than expected. Furthermore, the per-channel normalization increases the accuracy on the validation set of the `PerrosYGatosNet` by 0.6%. Additionally, we can see that for three out of four configurations shown in Table 2 the training loss is already very close to 0. Furthermore, the performance on the validation set and test set is very similar.

Did you have any problems reaching a low training error?
Yes, because we forgot to set the momentum in the optimizer in `CnnClassifier` (took as more than half an hour of debugging to realize what the problem is 🥲).

Part 3: Addressing Over-fitting

What are the goals of regularization, data augmentation, and early stopping?
The purpose of regularization is to combat over-fitting of a model and thus aims at reducing the generalization error of a model (or, equivalently, increase the validation and test peformance). This is possibly achieved at the expense of an increased training set error. An example for a commonly used regularization technique is L_2 weight regularization which penalizes large weights to prevent certain inputs to dominate the output. Another regularization method is dropout. Neurons of a layer that uses dropout output 0 with a probability of p during training, i.e., neurons are not considered during a particular forward or backward pass with a probability of p.

Data augmentation is a technique that 'artificially' increases the size of a (limited) training set by applying transformations to the input images during training such as random crops, shifts and rotations but also changes to the brightness and contrast of the images. The aim is, again, to combat over-fitting.

Similarly, early stopping is another technique used to avoid over-fitting. In general, the error on the training set decreases progressively with the number of elapsed training epochs. So does the validation error, however, in most cases the performance of the model on the validation set increases after a certain amount of epochs. Therefore, the model with the best performance (i.e., highest accuracy or lowest loss) on the validation set is stored and the training stops if the performance on the validation set does not show an improvent for a pre-defined number of epochs e.

Both, data augmentation and early stopping can also be classified as types of regularizations.

How exactly did you use these techniques (hyperparameters, combinations) and what were your results (train and val performance)?
For the experiments we use the `PerrosYGatosNet` with per-channel normalization since it was clearly superior to the `BaselineCNN` in Part 2. Similar to Part 2, we use SGD with a Nesterov Momentum of 0.9, a learning rate $1e-2$, a batch size of 128 and again shuffle the samples during training. The maxmimum number of training epochs is 500.

We make use of the following data augmentation/regularization techniques:

- Data augmentation: random crop, random horizontal flip, random vertical flip (implemented in addition to the mandatory operations, see `vflip()` in `ops.py`)
- Regularization: weight decay, channel dropout after every activation function of a convolutional layer and dropout after the activation function of the hidden fully-connected layer
- Early stopping: We stop the training in case the validation accuracy has not improved for 25 consecutive epochs

In total, we experimted with 11 different settings (see Table 3). Figure 3 exemplarily visualizes the training loss and the validation accuracy of selected configurations.

Data Augmentation rcrop	hflip	vflip	Channel Dropout*	Weight Decay	# Training Epochs	Train Loss	Acc. Validation Set (Best)	Acc. Test Set
–	✓	–	–	–	135	0.020 ± 0.015	0.745	0.729
✓	✓	–	–	–	99	0.121 ± 0.048	0.744	0.739
✓	✓	–	0.25	–	123	0.496 ± 0.042	0.753	0.725
✓	✓	–	–	$1e-3$	91	0.293 ± 0.044	0.741	0.731
✓	✓	–	0.25	$1e-3$	215	0.443 ± 0.035	0.762	0.760
–	–	✓	0.25	$1e-3$	234	0.314 ± 0.050	0.815	0.803
–	–	–	0.25	–	159	0.327 ± 0.047	0.809	0.814
–	–	–	0.25	$1e-3$	251	0.287 ± 0.048	**0.821**	**0.827**
–	–	–	0.35	$1e-3$	257	0.381 ± 0.040	0.811	0.817
–	–	–	–	$1e-2$	34	0.693 ± 0.001	0.559	0.564
–	–	–	–	$1e-3$	96	0.123 ± 0.042	0.791	0.774

* Dropout of hidden fully-connected linear layer (with probability p=0.5) is used whenever channel dropout is used
rcrop ... random crop (`sz=32`, `pad=1`, `pad_mode="constant"`)
hflip ... random horizontal flip
vflip ... random vertical flip

Table 3: Impact of different data augmentation/regularization techniques. The column 'Train Loss' lists the loss of the last epoch.

Similar to Part 2, we can observe that the accuracy on the test set is always close to the respective best validation accuracy. Using a weight decay of $1e-2$ does not work – at all; it leads to a non-decreasing training loss. Using only data augmentation leads to a decrease in validation accuracy compared to the results achieved Part 2. One reason for this could be that the used augmentations do not reflect 'good' transformations in our image domain. The setting that yields the best performance is a combination of dropout and weight decay ($1e-3$) (saved as `best\model.pt`). It achieves a validation accuracy of 82.1% (+1.1% compared to Part 2) and a test accuracy of 82.7% (+1.6% compared to Part 2). We tried to further improve the accuracy of this setting by increasing the channel dropout to 0.35 and by using data augmentation (vertical flip), however, this lead – again – to worse results. Actually, we would have expected more improvement with data augmentation/regularization. One reason that results only improved marginally could be that the training set is too small in general to train from scratch.

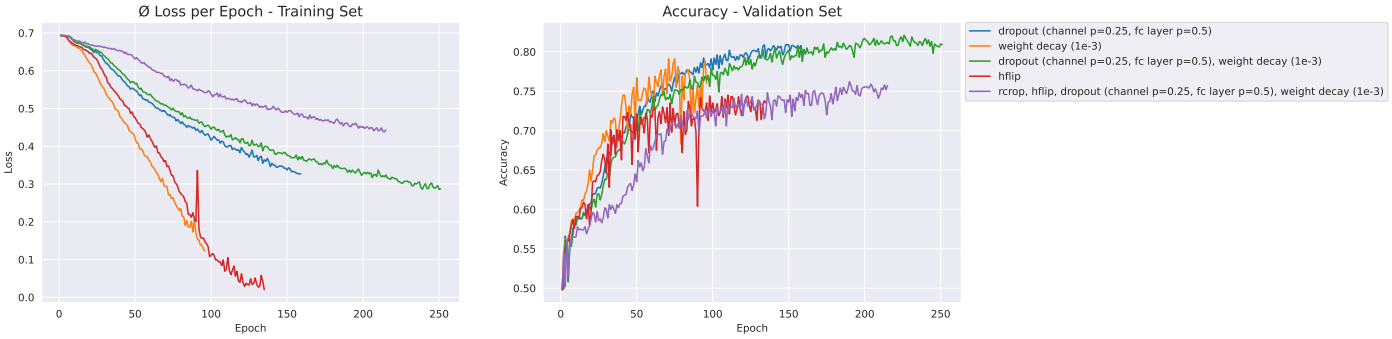

Figure 3: Average loss per epoch on the training set (left) and accuracy on the validation set (right) for selected configurations of Table 3.

Transfer Learning

For transfer learning, we use a pre-trained `ResNet18` and re-train the *whole* model (see `transfer_learning_cats_and_dogs_pt3.py`). The classification layer was replaced by a 2-unit linear layer. Images are re-shaped to the expected input dimension (224×224) with bilinear interpolation (see `resize()` in `ops.py`) and normalized per channel using the statistics of the ImageNet training set[2]. We shuffle the training set, use a batch size of 128, early stopping (after 5 epochs) and train for max. 30 epochs. No further regularization/data augmentation is used. Again, SGD with a Nesterov Momentum of 0.9 is used. Since our approach is computationally expensive we only varied the learning rate (see Table 4 for results). Compared to Part 2 we improved by 8.6% (validation set) and 6.4% (test set). The best model is saved as `best_model_transfer_learning.pt`

Learning Rate	# Train Epochs	Train Loss	Acc. Validation Set (Best)	Acc. Training Set
$1e-3$	16	0.005 ± 0.003	**0.907**	**0.891**
$1e-4$	30	0.138 ± 0.028	0.879	0.876

Table 4: Fine-tuning of a pre-trained `Resnet18`.

[2] `https://pytorch.org/vision/stable/models.html`, Accessed: 2022-05-20