Deep Learning for Visual Computing

Marcin Konefal, Daria Lazic

January 3, 2019

1 Assignment 2

1.1 Part 1: Gradient Descent

The gradient descent algorithm in general in deep learning is used to minimize a loss function that is non-linear in its parameters, like the cross entropy loss (very popular in classification). In the stochastic gradient descent, by definition parameters of a loss function are updated using one sample. In practice however, the batch size is usually 64, 128 or 256. In the assignment three different functions were tested. In an iterative process function point coordinates (which correspond to the parameters of the function) were updated using the gradient at the current position (deduced from calculated Hessian) multiplied with the learning rate. To evaluate the function at the current location, bilinear interpolation was used for approximation of non-integer coordinates. The algorithm has the most interesting behaviour for the Schaffer function, which has multiple sharp local minima. For other functions, the algorithm monotonously converges to the global minimum (at least if it is possible from the passed starting points). For the schaffer function however, the algorithm struggles to escape the local minimum, it is also much more sensitive to parameter change. For all three functions a learning rate of 10 works best. However, increasing the learning rate speeds up the optimization but can lead to overshooting. The same is true for epsilon, which is used for computing the numeric gradient descent.B builds up momentum if successive gradients are similar and thus improves the speed of convergence. A β of 0.9 turned out to be the most appropriate trade-off between speed and convergence. Nesterov momentum, together with a momentum parameter β of 0.9 gave best results. The stopping condition defines the value of the gradient at which the algorithm is stopped and depending on the function should be close to zero.

1.2 Part 2: Convolutional Neural Networks

The following newtork architecture was used:

```python
class CNN(nn.Module):

    def __init__(self):
        super(CNN, self).__init__()

        # Layer 1, images are 32x32
        # Input channels = 3, output channels = 32
        # Padding for stride = 1 is calculated by (K-1)/2
        self._cn1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        # Layer 2, after pooling images are 16x16
        self._cn2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        # Layer 3, after pooling images are 8x8
        self._cn3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        # Layer 4, after pooling images are 4x4 - with padding 6x6
        self._cn4 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        # Layer 4, after pooling images are 2x2 - with padding 4x4
        self._cn5 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)

        self._pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self._global_avg_pool = nn.AdaptiveAvgPool2d(1) # 2 is the kernel size

        # 512*1*1 input features, 2 output features
        self._fc1 = nn.Linear(512, 2)

    def forward(self, data):
        data = F.relu(self._cn1(data))
        data = self._pool(data)
        data = F.relu(self._cn2(data))
        data = self._pool(data)
        data = F.relu(self._cn3(data))
        data = self._pool(data)
        data = F.relu(self._cn4(data))
        data = self._pool(data)
        data = F.relu(self._cn5(data))
        # global average
        data = self._global_avg_pool(data)
        data = data.view(-1, 512*1*1)
        data = self._fc1(data)

        return data
```

Listing 1: CNN

As can be seen in Listing 1, we used the basic design recipe presented in the lecture, which gave the best results. The first layer is a convolutional layer with a kernel size of 3 and stride 1. Padding for stride = 1 can be calculated by

$$padding = (kernelsize - 1)/2 \tag{1}$$

By using stride of 1 and padding of 1, the size of the image stays the same after convolution. Stride=2 would be too aggressive. Since our images have a size of WxH of 32x32 a kernel size of 3 makes sense (makes it possible to go deeper). After every conv layer, a ReLU activation function (sets all negative values to zero) and a pooling layer follow. Pooling is done by 2x2 max-pooling and stride of 2, which returns the maximum value in a window of 2x2 pixels. Width and height are reduced by a factor of 2 after each pooling layer. In the first convolutional layer we have three input channels (RGB) and 32 output channels, which correspond to 32 feature maps. In each subsequent block the number of feature maps is increased by a factor of 2. Since width and height of the image are reduced by a factor of 2 after each block, we used 4 subsequent blocks. After the last max-pooling layer, samples have a size of WxHxD of 2x2x256. After padding, WxH is 4x4 and thus a last conv layer with a kernel size of 3 and 512 output channels can be used (output

samples are of size 2x2x512). The last pooling layer applies a global average pooling, which returns an average value for each feature map. So samples that are fed into the linear layer are of size 1x1x512. The linear layer has as many neurons as classes (2 in our case).
As a loss function, cross entropy loss was used. The applied optimizer is an SGD with nesterov momentum and β of 0.9. Weight decay can be passed to the algorithm as an argument. The sequence of the training starts with a forward pass (Listing 1), followed by a backward pass to compute the gradients and ends with an optimization step using SGD to update model parameters (feature maps). A softmax layer is applied to the output of the linear layer for predicting probability class scores.

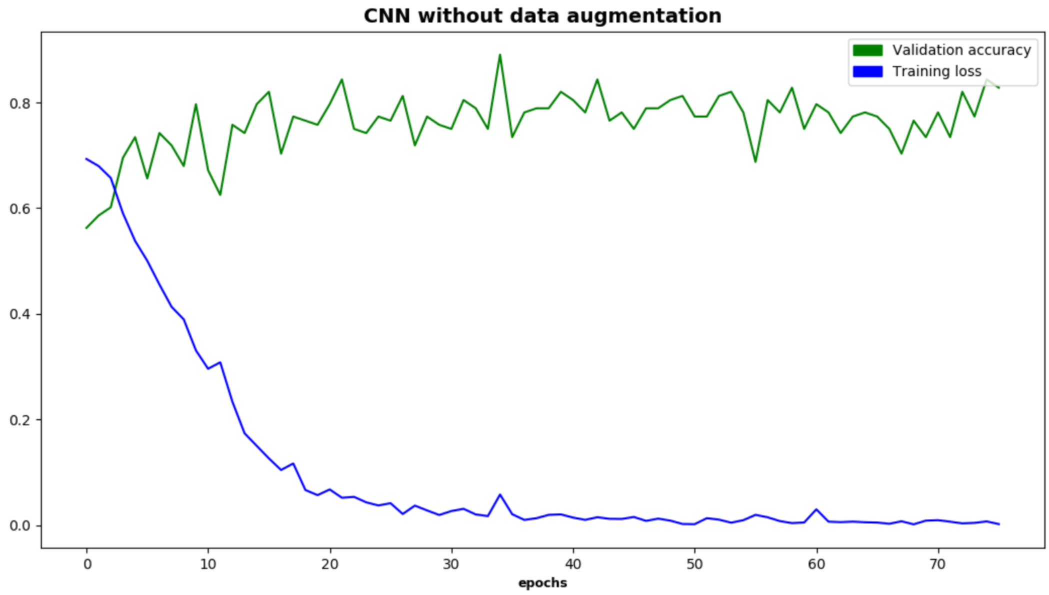

Figure 1: Training CNN without data augmentation.

Using a batch size of 128 samples and using input normalization, a learning rate of 0.1 turned out to be the best. The training loss goes to zero very quickly, while the validation accuracy reaches a maximum of 0.89 (Fig. 1).

1.3 Data Augmentation and Regularization

Data augmentation and regularization are used to reduce overfitting and 'explosion' of model parameters. While training loss decreases steadily with the number of epochs, validation loss never reaches the performance of the training loss and starts to increase at some point, because the model is overfitted to training data. This is because of the fact, that training data is limited and thus the model does not generalize well to unseen data. In addition to data augmentation and regularization, early stopping can be used, which saves the current state of the parameters if the highest validation performance is achieved (Listing 2).

```
if v_accuracy > best_accuracy:
    torch.save(cnn_net.state_dict(), os.path.join(os.getcwd(), 'best_model.pth'))
    best_accuracy = v_accuracy
```

Listing 2: Early stopping

In our assignment, we applied random cropping and flipping (Listing 3) for data augmentation, but also other transformations can be used like random scaling.

```
def hflip() -> Op:
    return np.fliplr


def rcrop(sz: int, pad: int, pad_mode: str) -> Op:
    def op(sample: np.ndarray) -> np.ndarray:
        origin_size = sample.shape
        cropped_size = (origin_size[0] - sz, origin_size[1] - sz)
        crop_point = (random.randrange(cropped_size[0]), random.randrange(
    cropped_size[1]))

        cropped_img = sample[crop_point[0]: crop_point[0]+sz, crop_point[1]:
    crop_point[1]+sz]

        if pad > 0:
            cropped_img = np.pad(cropped_img, ((pad, pad), (pad, pad), (0, 0)),
    pad_mode)

        if cropped_img.shape[0] > origin_size[0] or cropped_img.shape[1] >
    origin_size[1]:
            raise ValueError("Image ofter cropping and padding has size: " + str(
    cropped_img.shape) +
                             ", its size is bigger then size of origin picture: " +
     str(origin_size) + ".")

        return cropped_img
```

Listing 3: Flipping and Cropping

Regularization improves the validation and test performance with the expense of training performance, because of the possible decrease in model variance/ flexibility. In the assignment we used weight decay-which is incorporated in the optimizer-for regularization (penalizes large weights), but there are also other methods for regularization like dropout (output zero with probability p).

We tested different methods and combination of methods to reduce overfitting (Fig. 2, Fig. 3).

The training loss goes to zero fast for all methods. With weight decay of 10^{-5} and 10^{-6} it stays at zero and there are less fluctuations. Cropping only has the worst performance concerning training loss. Validation accuracy reaches the highest value using data augmentation (flipping, cropping) and a weight decay of 10^{-5}. This maximum value of 0.921875 is reached already after 12 epochs. This model was thus considered the best model and model parameters were saved (Listing 2). The worse performance of using the model without any kind of augmentation or regularization is very prominent for validation accuracy.

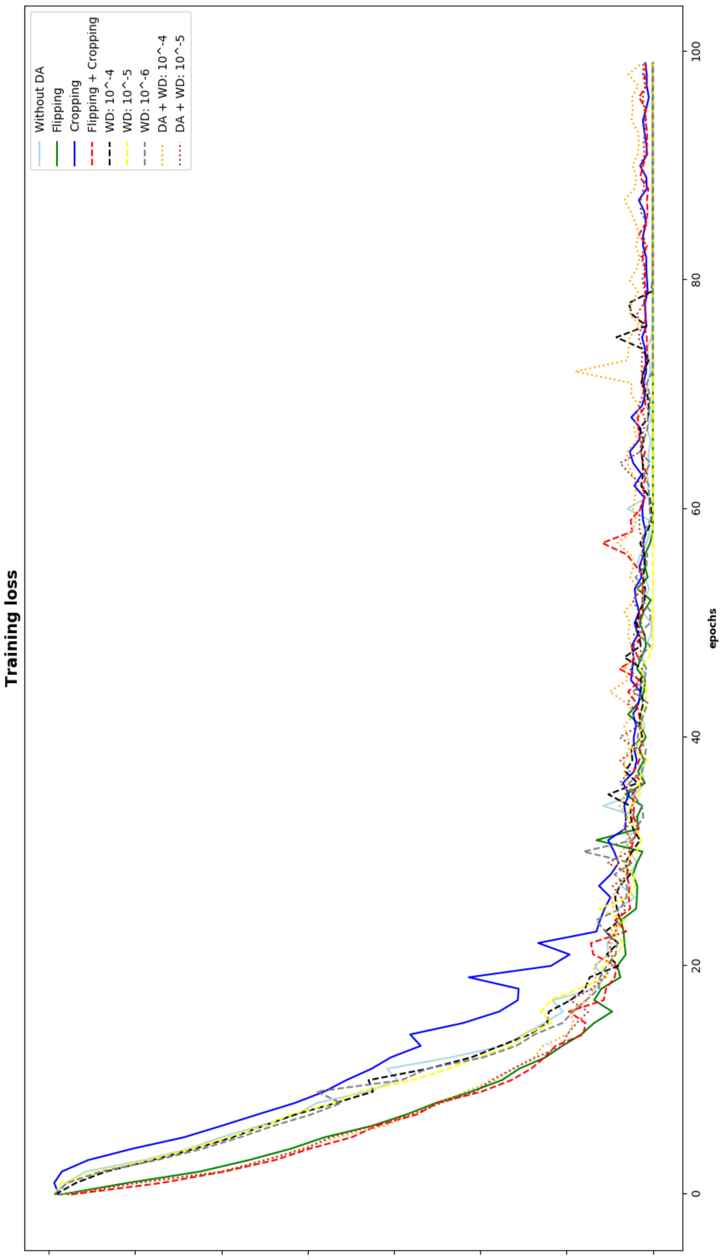

Figure 2: Training loss using different data augmentation and regularization methods. Without DA - without data augmentation and regularization- corresponds to part 2; Flipping - flipping only; Cropping - cropping only; Flipping + Cropping - flipping and cropping only; WD 10^{-4} - weight decay = 10^{-4}; WD 10^{-5} - weight decay = 10^{-5}; WD 10^{-6} - weight decay = 10^{-6}; DA + WD: 10^{-4} - weight decay = 10^{-4} with flipping and cropping; DA + WD: 10^{-5} - weight decay = 10^{-5} with flipping and cropping.

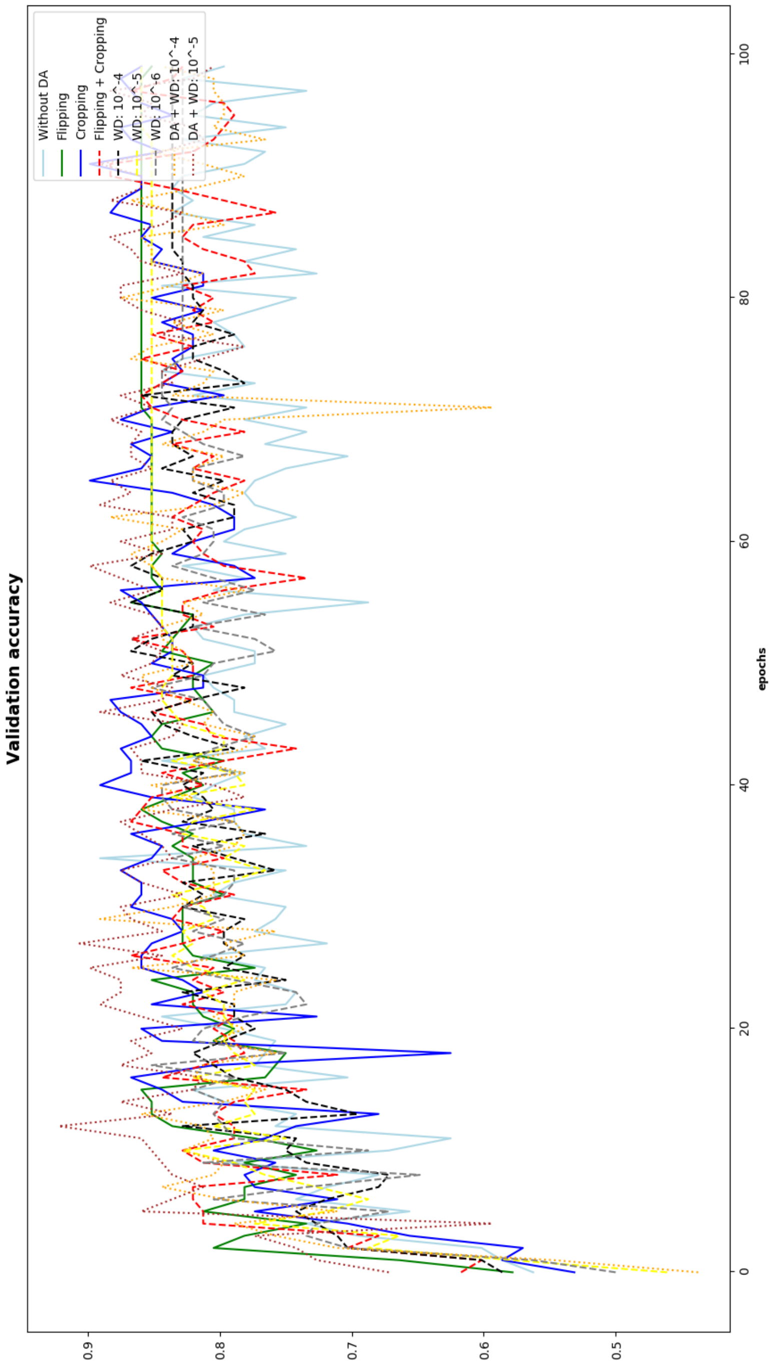

Figure 3: Validation accuracy using different data augmentation and regularization methods. Without DA - without data augmentation and regularization- corresponds to part 2; Flipping - flipping only; Cropping - cropping only; Flipping + Cropping - flipping and cropping only; WD 10^{-4} - weight decay = 10^{-4}; WD 10^{-5} - weight decay = 10^{-5}; WD 10^{-6} - weight decay = 10^{-6}; DA + WD: 10^{-4} - weight decay = 10^{-4} with flipping and cropping; DA + WD: 10^{-5} - weight decay = 10^{-5} with flipping and cropping.