Project 1: Backpropagation and Gradient Descent

INF265 - University of Bergen

Vebjørn Sæten Skre Martin Flo Øfstaas

Spring 2024

Division of labour: We split the project into three parts. Martin did section 2 and Vebjørn did section 3, while both wrote the report.

1 Explanation of our approach and design choices

1.1 The Backpropagation algorithm

The backpropogation algorithm implemented in this project stems from *the equations of backpropogation* [1, p. 47][2, p. 106], which revolves around the four formulas BP1, BP2, BP3, BP4. Within the comments of our code , we have referenced these equations when they were calculated. Although the formulas from Nielson and Prince explain the same thing, we found it easy to reference the equations from Nielson's book, and thus we kept using his naming convention to reference the equations. To make it clear, we have written both Prince's and Nielson's equation below. Here we will first write the variable name we used in our code, then the equation of Prince and lastly the equation from Nielson:

$$\text{dL\_df\_L}: \frac{\partial \ell}{\partial \mathbf{f}_K} \odot \tanh'(z^L), \quad \delta_j^L = \frac{\partial L}{\partial z_j^L} \odot \tanh'(z^L) = \nabla_a C \odot \tanh'(z^L) \tag{BP1}$$

$$\text{dL\_df\_k}: \frac{\partial \ell_i}{\partial \mathbf{f}_{k-1}} = \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\mathbf{\Omega}_k^T \frac{\partial \ell_i}{\partial \mathbf{f}_k}\right), \quad \delta_j^l = \left((W^{l+1})^T \delta^{l+1}\right) \odot \sigma'(z^l) \tag{BP2}$$

$$\text{dL\_db}: \frac{\partial \ell_i}{\partial \beta_{\mathbf{k}}} = \frac{\partial \ell_i}{\partial \mathbf{f}_k}, \quad \frac{\partial L}{\partial b_j^l} = \delta^l \tag{BP3}$$

$$\text{dL\_dw}: \frac{\partial \ell_i}{\partial \mathbf{\Omega}_k} = \frac{\partial \ell_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T, \quad \frac{\partial L}{\partial w_{jk}^l} = a_k^{l-1} \delta^l \tag{BP4}$$

Then we implemented the equations $BP1, \dots, BP4$ in the backpropagation function, as seen on the next page. We first calculated the error for the outputs, weights and biases in layer L with respect to the Loss function, before we looped backwards from $L-1$ in our network and did the same for the hidden layers. We found this to be a clear way to do it as the formulas was a little different between the last layer L and the hidden layers l.

```
def backpropagation(model, y_true, y_pred):
    ...
    dL_df_L = dL_dy * tanh_prime_z_L #(BP1)
    ...
    dL_dw = dL_df_L.T @ h_k_T #(BP4)
    ...
    dL_db = dL_df_L #(BP3)

    for l in range(model.L-1, 0, -1):
        ...
        dL_df_k = (w_from_last.T @ dL_df_k.T).T * tanh_prime_z_l #(BP2)
        ...
        dL_dw = dL_df_k.T @ h_k_T #(BP4)
        ...
        dL_db = dL_df_k #(BP3)
```

1.2 Gradient Decent

In section 3 of the assignment paper, we were asked to define a machine learning (ML) model and train this model on the CIFAR10 dataset through gradient decent. The assignment included steps such as:

- Data splitting and preprocessing
- Defining of a deep neural network
- Creating two train function, one standard and one modified
- Testing the model with different hyperparameters
- Evaluating the best model

Testing hyperparameters and evaluation of the best model will be covered in section 2.6 and 3 respectively

1.2.1 Data splitting and preprocessing

We started by loading the CIFAR10 training dataset and splitting this data into a training and validation using our globally defined seed. The purpose of this was to be able to find the statistical values, such as mean and standard deviation, of the training set only, to later be able to normalize the data in our load_cifar function with the correct statistical values. Our model should however not be trained on the whole CIFAR10 dataset, but the objective was to train it to classify between birds and planes. We can see all the classes and their corresponding images in figure 1.

From figure 1 we could conclude that the classes to keep was class 0 and class 1. In the load_cifar function we added our preprocessor which transformed the images to tensors and normalized the images with the values retrieved earlier. We split the training dataset into a training and validation dataset with a 85% ratio, and reduced the dataset to two classes 0: planes, and 1: birds.

Figure 1: Pictures of each class in CIFAR10

1.2.2 Defining the network

We chose to define our deep neural network (DNN) using the modular approach. In the init function we defined our layers and activation functions. The first layer took 3072 dimensions in and sent 512 out, the second layer took 512 in and 128 out, and the last layer took 32 in and 2 out, one for each of our two classes. The first three layers had the activation function ReLU but the last layer needed no function as cross-entropy loss already includes a softmax activation function. The code implementation can be shown below:

```
class MyMLP(nn.Module):
    def __init__(self):
        super(MyMLP, self).__init__()

        # We flatten the image to feed the first fully connected layer a 1d array
        self.flat = nn.Flatten()

        # First layer
        self.fc1 = nn.Linear(32*32*3, 512)
        self.act1 = nn.ReLU()

        # Second layer
        self.fc2 = nn.Linear(512, 128)
        self.act2 = nn.ReLU()

        # Third layer
        self.fc3 = nn.Linear(128, 32)
        self.act3 = nn.ReLU()

        # Output layer
        # No activation function on the last layer
        self.fc4 = nn.Linear(32, 2)
    def forward(self, x):
        out = self.flat(x)              # Turnes image into 1d array
        out = self.act1(self.fc1(out)) # The output of the 1. layer and activation
        out = self.act2(self.fc2(out)) # functions is the input of the next
        out = self.act3(self.fc3(out))
        out = self.fc4(out)
        return out
```

1.2.3 Creating the train functions

The assignment specified that they wanted two training functions. One standard train() where we were allowed to use an optimizer to effectively update/train the parameters in our model automatically, and one train_manual_update() where we had to update the parameters ourselves. So the only difference between the two methods was that we changed the optimizer.step() and optimizer.zero_grad() to our own implementation of parameter training. The train manual method took arguments such as number of training epochs, the model to be trained, loss function and train loader. The train loader is created through the use of torch.utils.data.DataLoader which neatly splits the data into batches our model could process. In train_manual_update() we also included the possibility to add hyperparameter values such as learning rate, momentum and weight decay. This was implemented using the logic from the documentation of torch.optim.SGD.

We instantiated momentum buffers, or velocities, before going into the training loop, and by updating the velocities of each parameter after every batch of data, we could choose to use this to regularize the model. The implementation of weight decay was simply a matter of subtracting a fraction of the weights from the gradients during each batch of data, where the fraction was determined by the weight decay hyperparameter. The learning rate was used to adjust how big of a step in the direction of the gradient the model should take. To be sure that our train_manual_update() worked we trained models with the same seed and compared the results. As the results form both models were identical we could conclude that our implementation was correct.

2 Answering the following questions

2.1 Which PyTorch method(s) correspond to the tasks described in section 2

Section 2 described the method of implementing backpropagation which in pytorch is done in two steps. After the forward pass, we can calculate the "Loss". Then you call the ".backward()" function on the error tensor [3].

```
loss = (prediction - labels).sum()
loss.backward() # backward pass
```

These gradients can then be accessed by the parameters ".grad" attribute, which Autagrad has calculated and stored [3].

2.2 Cite a method used to check whether the computed gradient of a function seems correct. Briefly explain how you would use this method to check your computed gradients in section 2

The method used to check whether the computed gradient of our backpropagation was correct was done by the file "tests_backpropagation". This file compares our gradients with PyTorch's autograd function, which is tested at two threshold levels. We used this method to ensure that we computed the correct biases and weights.

2.3 Which PyTorch method(s) correspond to the tasks described in section 3, question 4.?

The pytorch methods that corresponds to the task of manually updating the trainable parameters in the network is optim.SGD().step() and optim.SGD().zero_grad().

2.4 Briefly explain the purpose of adding momentum to the gradient descent algorithm

In stochastic gradient descent, the gradient step can often oscillate causing the optimization path to “bounce around“ rathar than converging smoothly. To address this issue, it is common to implement a “momentum“ term. The purpose of adding the momentum is to make the trajectory of the gradient steps more smooth. This is done by incorporating the velocity of the previous steps into the current one, making the trajectory of the optimization more smooth in reducing the loss function trajectory.

2.5 Briefly explain the purpose of adding regularization to the gradient descent algorithm

In a neural network, larger weights often correlate to more complex output functions which leads to overfitting. This can best be seen by looking at Figure 2, where it shows that by increasing the λ value, the model is encouraged to adopt smaller weights. This helps the model to generalize better to unseen data (plotted as the black line in the figure). The purpose of adding regularization in gradient descent is to encourage smaller weights, resulting in a less complex and smoother output functions [2, p.140].

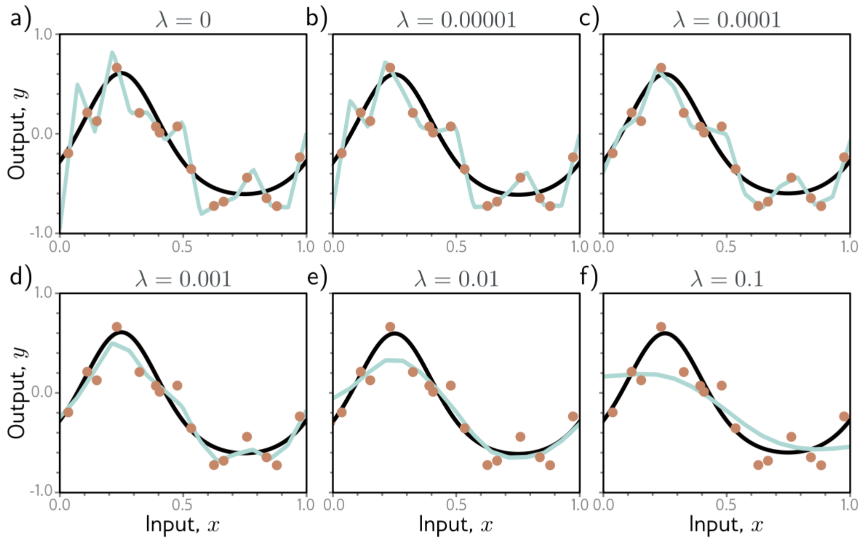


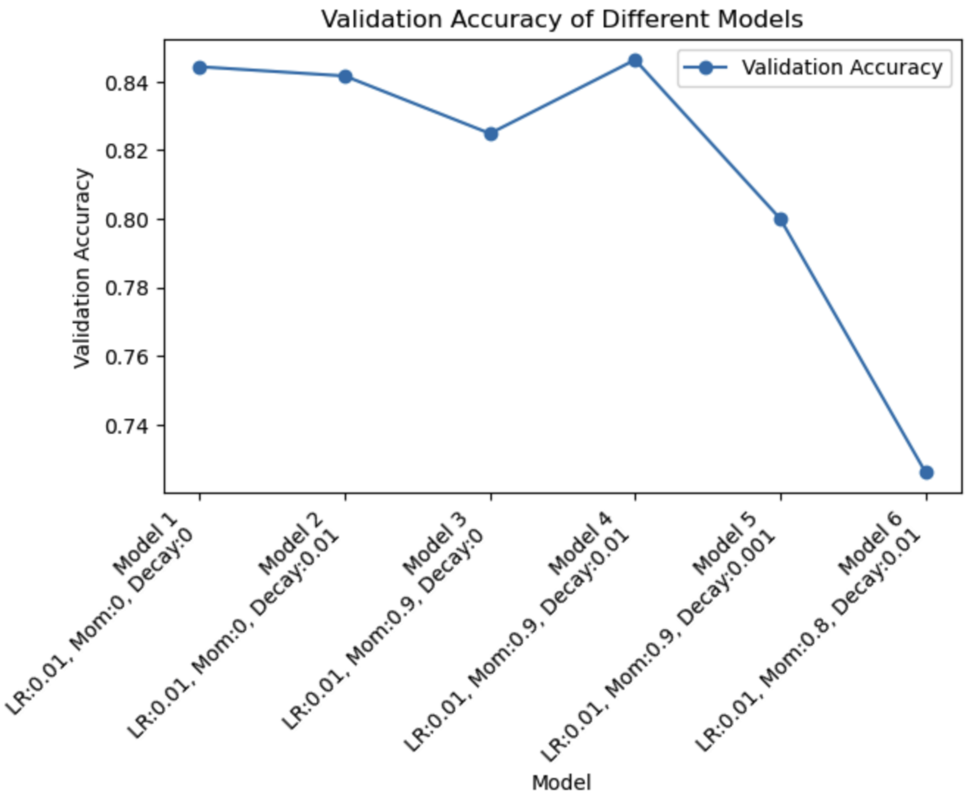
Figure 2: L2 regularization with different λ values [3, p. 141]

2.6 Report the different parameters used in section 3, question 8., the selected parameters in question 9. as well as the evaluation of your selected modem

We trained trained a total of 6 modes with the following hyperparameters:

1. Learning rate: 0.01, Momentum: 0 and weight decay: 0
2. Learning rate: 0.01, Momentum: 0 and weight decay: 0.01
3. Learning rate: 0.01, Momentum: 0.9 and weight decay: 0
4. Learning rate: 0.01, Momentum: 0.9 and weight decay: 0.01
5. Learning rate: 0.01, Momentum: 0.9 and weight decay: 0.001
6. Learning rate: 0.01, Momentum: 0.8 and weight decay: 0.01

The model was trained in one big loop and the information of each model was stored a a tuple. We then automatically selected the best model based on validation accuracy. We found that the best was model 4 and had learning rate = 0.01, momentum = 0.9 and velocity = 0.01. These results are also illustrated in the figure below:



2.7 Comment your results. In case you do not get expected results, try to give potential reasons that would explain why your code does not work and/or your results differ

When we finally evaluated the best model on the test dataset we found that it had an accuracy of 84%. On the validation set the model had an accuracy of 85% and on the training set the model had an accuracy of 96%. From these results we suspect that the model might be overfitting to the training data, and to find a better model we would like to test even more hyperparameters and weight initializations. We should also consider trying different training/validation splits as we throughout the project only used a 0.85 ratio. Overall our results seemed fine, and the train_manual_update method returned the same results as the SGD method. One thing however that was a bit unexpected was that out of the 6 models we trained the their accuracy seemed to lie between 80-84% validation accuracy, but model 6 however performed much worse. Model 6 had a validation accuracy of just 73% even though the only difference between model 6 and the best model was that model 6 had momentum = 0.8 whereas the best model had momentum = 0.9. One explanation for this could be that model 6 was simply unlucky and the difference in momentum made it not converge quick enough, which sent it to a suboptimal region.

References

[1] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[2] Simon J.D. Prince. *Understanding Deep Learning.* MIT Press, 2023.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.