LEUPHANA
UNIVERSITÄT LÜNEBURG

Optimization for Deep Learning applied to X-Ray Image Classification

Author: Andrei Perov, 3041544

Course: Optimization techniques

Examinator: Prof. Dr.-Ing. Paolo Mercorelli

M.Sc. Management and Data Science

Summer semester 2021

1. Optimization in Deep Learning

Deep learning algorithms (artificial neural networks) involve optimization algorithms to learn how to fit the data the best way. For any deep learning task the process of optimization can be described as a search for the parameters of a neural network that significantly reduce a loss function (Goodfellow et al, 2016).

The parameters of a neural network is a set of weights for each connection in the network that characterize how neurons contribute to the final outcome. The value of the loss function reflects how the outcome of the model for the given input and weights deviates from the true outcome. For simplicity in this report we will assume that loss function minimum corresponds to the neural network state which fits the input data the best way. This assumption can be satisfied for the opposite case too, because the maximum of the function is the minimum of this function taken with the negative sign (Abu-Mostafa et al, 2012).

In order to get to the minimum of the loss function the set of parameters (weights) of a neural network should be adjusted by some optimization algorithm.

In particular machine learning models such as OLS (ordinary least squares) linear regression the minimum of the loss function can be found analytically. We only need to equate the derivative of the loss function to zero and solve the equation for the parameters in the vector form. But for complex non linear models, such as deep learning models with a big amount of parameters and input instances, this method is not suitable (Abu-Mostafa et al, 2012).

In deep learning, gradient based methods are the most common. These methods are based on the calculation of the derivative of the loss function (L). Derivative dL/dw gives the value of the slope of $L(w)$ at the particular point $w$and therefore the derivative tells us how to change w in order to make small improvement in L. We can make loss function L smaller by taking small steps in weights w with the opposite sign of the derivative. This method is called gradient descent. The point when the derivative of Lhas a value of zero is the minimum of the function and the algorithm will not change the values of the parameters. It is important to mention, that in the case of a neural network the critical point of the loss function can be a local minimum that is higher than the global minimum, and we are always interested in the global minimum.

The updates of the weights can be characterized by formula:

$$w' = w - \eta \cdot \nabla_w L(w)$$

where ∇_w - gradient is a vector which represents partial derivatives with respect to every weight, η - learning rate is the hyperparameter of the model that determines the size of the step. Learning rate is crucial for optimization. Too large learning rate can cause oscillations around minimum when the value of the loss function is not decreasing and can even increase with new steps. Too small learning rate can cause extremely slow learning when parameter updates are negligible, especially closer to the minimum (Abu-Mostafa et al, 2012).

The calculation of the loss function gradient can be done with the whole dataset and in this case we are talking about full batch gradient descent. The gradient calculated this way

reflects the true value for the dataset but calculation of the one step is computationally expensive and demands occupation of the large amount of computer memory at once and this method is not feasible for huge datasets.

The opposite case is stochastic gradient descent when gradient is calculated based on only one example. The loss function is being optimized with much less information per step. It can increase the runtime because updates will be done in arbitrary directions (Abu-Mostafa et al, 2012).

For deep learning models the most common is mini batch gradient descent that uses several examples to calculate one step but benefits from the computation speedup for multiple samples. The samples are drawn randomly without replacement. Randomly selecting examples will help avoid redundant examples or examples that are very similar that don't contribute much to the learning. Also the stochastic part of the mini batch gradient descent increases noise that can help to avoid overfitting (Goodfellow et al, 2016).

For this project I used two versions of optimizers based on mini batch gradient descent methods: stochastic gradient descent with momentum and ADAM optimizer. Both of these algorithms use "inertia" of the previous steps to steer the update direction smoother and reduce noisy movements (Kingma et al, 2015).

The other family of methods for optimization that can be used for deep learning models is second order optimization methods. For these methods the update step of the parameters besides gradients incorporates an inverse Hessian matrix with the second partial derivatives as elements. To find a minimum of the function in one of the most common

second order methods - Newton method with adaptive step size, the following update rule is used:

$$w' = w - \eta \cdot \nabla_w L / \nabla^2_w L$$

where $\nabla^2_w L$ is a symmetrical Hessian matrix.

The second order derivative helps to estimate the curvature of the loss function and therefore find the shortest way to the minimum (Ngiam et al, 2011).

The calculation of the second derivatives takes a lot of time and requires occupation of significant computer memory. Because of that, these types of methods are not common for deep learning. We can witness it by looking at the major deep learning frameworks Tensorflow and Pytorch. Tensorflow does not have any second order optimization methods embedded and Pytorch has only one - LBFGS optimizer.

LBFGS (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) is quasi-Newton method where the Hessian matrix is not calculated but approximated. BFGS determines the descent direction by preconditioning the gradient with curvature information. Since the updates of the BFGS curvature matrix do not require matrix inversion, its computational complexity is only $O(n^2)$ in comparison to $O(n^3)$ for Newton's method, where n is the number of parameters. $O(n^2)$ means that the computational load is proportional to the squared number of parameters. A limited memory version of BFGS uses for approximation just a few vectors instead of full size approximation of the Hessian matrix (n by n, where n

is the number of parameters) and several (often less than 10) previous iterations that significantly decreases the amount of occupied memory (Byrd et al, 1996).

2. Objectives and Optimization problem

The main objectives of this project:

1) To solve optimization problem that requires deep learning classification algorithms
2) To provide a comparison of different optimization methods including gradient based methods and second order methods.

As an optimization problem I chose the classification of the chest x-ray images . The reasons for such a choice is to use full-fledged deep learning models and to evaluate performance of different optimization methods on hard classification task. Some images with different classes differ negligibly, which makes optimization task quite tricky. Also, it is important to mention that the size of the images (amount of parameters) and the dataset are big enough to show computational advantages and disadvantages of different optimization methods.

3. Data

The dataset contains 5,863 x-ray images (JPEG) of the patients of two classes: “normal” and “pneumonia”. The size of the pictures vary in the range 1000 - 2000 pixels per dimension.

Chest x-ray images (anterior-posterior) were selected from retrospective cohorts of pediatric patients of one to five years old from Guangzhou Women and Children’s Medical

Center, Guangzhou. All chest x-ray imaging was performed as part of patients' routine clinical care.

For the analysis of chest x-ray images, all chest radiographs were initially screened for quality control by removing all low quality or unreadable scans. The diagnoses for the images were then graded by two expert physicians before being cleared for training the AI system. In order to account for any grading errors, the evaluation set was also checked by a third expert (Kermany et al, 2018)

4. Models and Optimization settings

To approach the classification tasks I chose the Cross Entropy Loss function to evaluate performance of my models. Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. To provide the probability to the loss function I use softmax function which maps linear combination of the input-weight products (s) to the probability of the certain class (i). Softmax function can be applied to multi class problems and has following expression for K classes :

$$\sigma(s_i) = exp(s_i) / \sum_{k=1}^{K} exp(s_k)$$

For binary case loss function can be express by following formula (y- is 0 or 1 for particular class, p - probability that the example belongs to class):

$$L = -y \cdot log(p) + (1 - y) \cdot log(1 - p)$$

To deal with image data, the best practice is to use convolutional neural networks. Convolution is the tool to structure the information of the image and extract meaningful

features from it. Convolution applies the same linear transformation locally, everywhere, and preserves the signal structure. Using different filters to the image we can detect edges, symmetries, different shapes as low level features and faces, legs, wheels on the higher level (closer to the output layer).

Initialization of weights of the deep learning model is an important start for optimization. Good initialization helps to avoid local minimums and make the training process much shorter and efficient. By default neural networks have random initialization of the weights. But there are different techniques to come up with good initial weights and one of them is to use an existing model that was trained with similar data. This approach is called transfer learning.

Researchers have already created plenty of well tuned architectures of convolutional neural networks and trained them on millions of different labeled images. Therefore we can use architecture and the weights of the pretrained model to start our optimization.

First model that I decided to train was a pretrained VGG-16 model trained on Imagenet dataset. Pretrained model has 30 convolution layers and 6 fully connected layers with up to 25088 neurons. I left the weights of the initial model preserved, except fully connected layers and trained. Fully connected layers I trained with a training set of x-ray images.

For training of this model and all models of this project I used a validation set to track the performance and avoid overfitting that can easily happen given complexity of the model and relatively small number of training examples. Validation is intermediary evaluation of the model on the hold off examples. Overfitting is one of the biggest problems of machine learning especially for such complex models as neural networks that can just adjust all

parameters to a particular dataset so the performance on the training set will be perfect but out of sample performance will be terrible.

Before training I also accounted for loss function calculation for the unbalanced training set, because “pneumonia” class is three times more common than “normal”. To deal with it I assigned weights to the classes for calculating the loss function so that “normal” images got two times as much loss points. I assigned two and not three because for medical purposes we have a priority to be able to detect “pneumonia” cases , in other words to avoid false negative classifications.

Next I found the optimal learning rates for optimization algorithms which happened to be 0.001 for SGD and 0.0005 for ADAM optimizer.

I trained pretrained VGG models with SGD and ADAM optimizers with 5003 training examples and evaluated trained models on a test set with 855 examples. The accuracy reached 0.9 for both algorithms. For LBFGS optimizer Pytorch provides only a batch version and I was not able to render the pre-trained model due to memory issues and even decreasing the training set could not help to resolve this problem.

Next I created a simple neural network with 3 convolution layers, 3 max pooling layers and 3 fully connected layers up to 360 (in comparison to 25000) neurons each. I managed to decrease the dimensionality of the fully connected layers for faster computation with a cost of information lost after max pooling layers. This model achieved accuracy 0.85 with 50 times shorter training time than the pretrained model. On the full training set LBFGS optimizer caused memory conflicts during the training (error: “not enough memory: you tried to allocate 5811484800 bytes”).

Next I shrunk the training set to 1400 examples which was the maximum size of the dataset that could be used with LBFGS optimizer without getting errors. To get the same amount of samples for each class I used oversampling of images with class “normal” . Also I set low values to hyperparameters of the optimizer to avoid computational overload : four for history size and four for maximum iterations. Given all limitations after this project only preliminary conclusions about second order optimization methods can be made.

5. Results and Interpretations

I used accuracy (ratio of predicted right to whole test set) and AUC (measure of separability) as main evaluation metrics . I also used *recall* (the ratio of true positives to all positives in the test set) and precision (the ratio of true positives to all predicted positives). It is crucial to get high recall because the aim of chest x-ray classification is to detect pneumonia to prevent consequences of the disease by treatment. Therefore the model should be trained to catch positive cases (recall is close to 1) and simultaneously control the number of false positives (precision as close to 1 as possible) to avoid unnecessary interventions for patients who do not need it.

1. Results for pretrained VGG-16 model

Initial settings:

- architecture: pretrained VGG-16
- model is trained on 5003 train examples
- Model is evaluated on 855 test examples

- mini batch size: 5
- number of epochs (epoch - the algorithm covered whole dataset one time): 4
- rendered in Google Colab (GPU 11.5 GB)

Table 1. Evaluation of the model based on VGG-16		
Metric	SGD optimizer	ADAM optimizer
Train accuracy	0.984	0.969
Test accuracy	0.905	0.903
AUC	0.905	0.909
Recall	0.992	0.982
Precision	0.832	0.834
Training time, s	3497	3315

For the pretrained model SGD with momentum and ADAM optimizer both got a high AUC of 0.9 that shows high ability to separate two classes even given an imbalanced training set. The accuracies are also equally high given arguable ground truth based on the expert opinions. For both optimizers the model shows reasonable differences between test and train accuracies that reflects low level of overfitting, most likely because of the applied validation technique.

Recall of the SGD optimizer is almost 10 percent higher than for ADAM, which is a strong advantage for a particular task when it is important to not miss the positive case. Training time is similar.

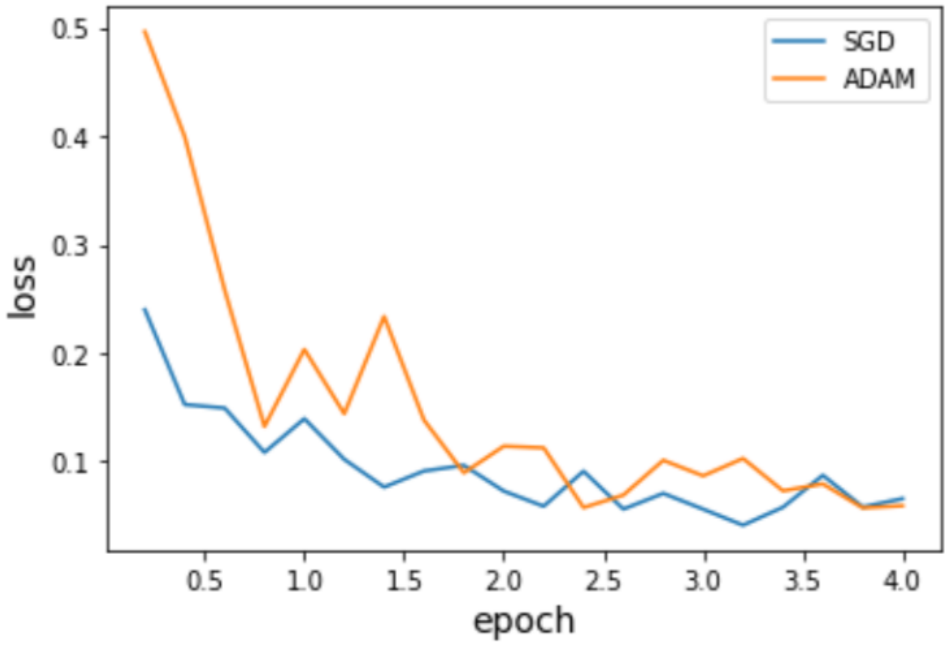

Fig 1 - Loss dynamic during training of VGG-16 for SGD and ADAM

On the Fig 1 we can see that SGD had possibly better initialization and it converged faster, also ADAM optimizer shows oscillations during the second epoch that can be based on peculiarities of the loss function surfaces and differences between batches (loss is calculated every 200 batches given 1000 batches in one epoch).

2. Results for custom neural net (5003 examples)

Initial settings:

- architecture : custom convolutional neural network
- model is trained on 5003 train examples
- model is evaluated on 855 test examples
- mini batch size: 5
- number of epochs: 4

- rendered on PC (GPU 2.5 GB)

Table 2. Evaluation of the custom neural network (5003 training examples)		
Metric	SGD optimizer	ADAM optimizer
Train accuracy	0.965	0.931
Test accuracy	0.855	0.903
AUC	0.865	0.905
Recall	0.982	0.936
Precision	0.766	0.862
Training time, s	577.27	577.27

We can see that even with a quite shallow neural network with just few convolutional and short fully connected layers the results of the classification are pretty good. It means that for production purposes it does not make sense to use heavy pretrained models.

ADAM optimizer classified dataset with the same accuracy and AUC as the pretrained model, for SGD accuracy decreased by 5 percent for this model in comparison to the pretrained one.

Train and test accuracies for SGD optimizer is significant and reflects occurance of overfitting. ADAM optimizer performs better in this sense too showing pretty low for such classification problem difference between train and test accuracies.

Recall of the SGD optimizer is much higher than for ADAM optimizer, even more significantly than for the previous model. It means that using SGD optimizer we will have better chances to detect pneumonia among new images. But the situation with precision

is the opposite, it means that with SGD optimizer we would classify normal images as images with pneumonia more often than for ADAM optimizer.

Training time is identical.

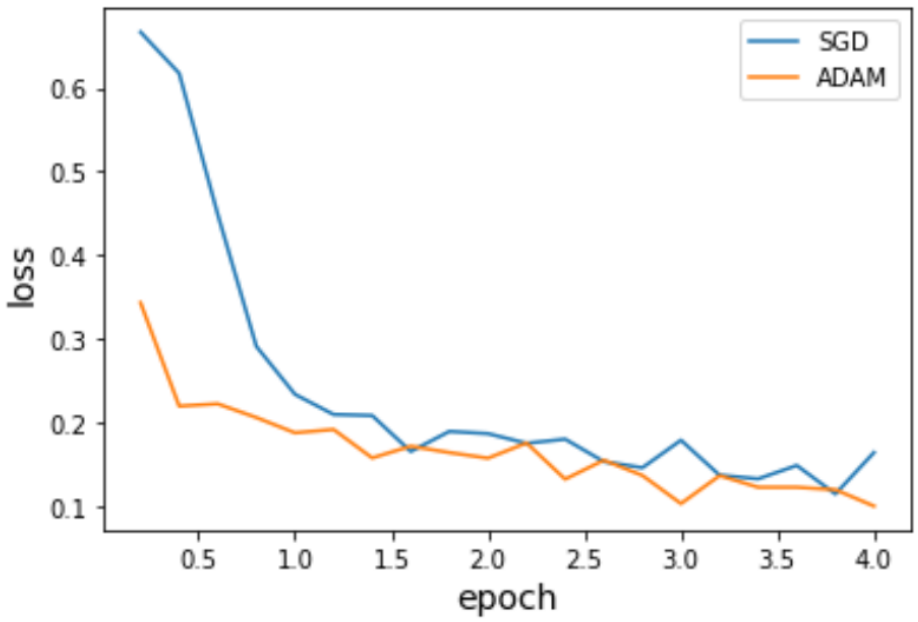

Fig 2 - Loss dynamic during training of custom neural net for SGD and ADAM

According to Fig 2, ADAM optimizer converges much faster than SGD, probably because of better initialization.

3. Comparison first and second order optimizers (1400 examples)

Initial settings:

- architecture : custom convolutional neural network
- model is trained on 1400 train examples
- model is evaluated on 855 test examples
- mini batch size: 5 (SGD and ADAM) and full dataset (LBFGS)
- number of epochs: 8

- SGD and ADAM were rendered on PC (GPU 2.5 GB)
- LBFGS was rendered in Google Colab (CPU 12 GB RAM)

Table 3. Evaluation of the custom neural network (1400 training examples)			
Metric	SGD optimizer	ADAM optimizer	LBFGS optimizer
Train accuracy	0.960	0.909	0.917
Test accuracy	0.869	0.880	0.822
AUC	0.876	0.876	0.830
Recall	0.956	0.897	0.923
Precision	0.797	0.841	0.746
Training time, s	327.69(was trained on 2.5 GB GPU)	327.69 (was trained on 2.5 GB GPU)	1089.55 (was trained on 12 GB CPU)

Looking at these results we need to consider that LBFGS optimizer's hyperparameters were set to lowest levels and an estimation of the inverse Hessian can be poorly done because of it. The performance of the model with LBFGS optimizer can give just a glimpse to potential capacities of this method. Also it is important to take into account that gradient based optimizers represented here use mini-batches of data for training and LBFGS has only full dataset mode.

Test accuracy and AUC of the model trained with LBFGS optimizer is significantly lower than gradient based methods, but for such a small portion of the dataset it is already good performance.

Overfitting for SGD with momentum and LBFGS optimizers is much higher than for ADAM optimizer, looking at recall scores we can conclude that overfitting is connected with “pneumonia” class. This can happen because of initial imbalance in classes.

Training time for LBFGS optimizer is roughly 10 times higher given the computational power difference between devices the models were rendered on.

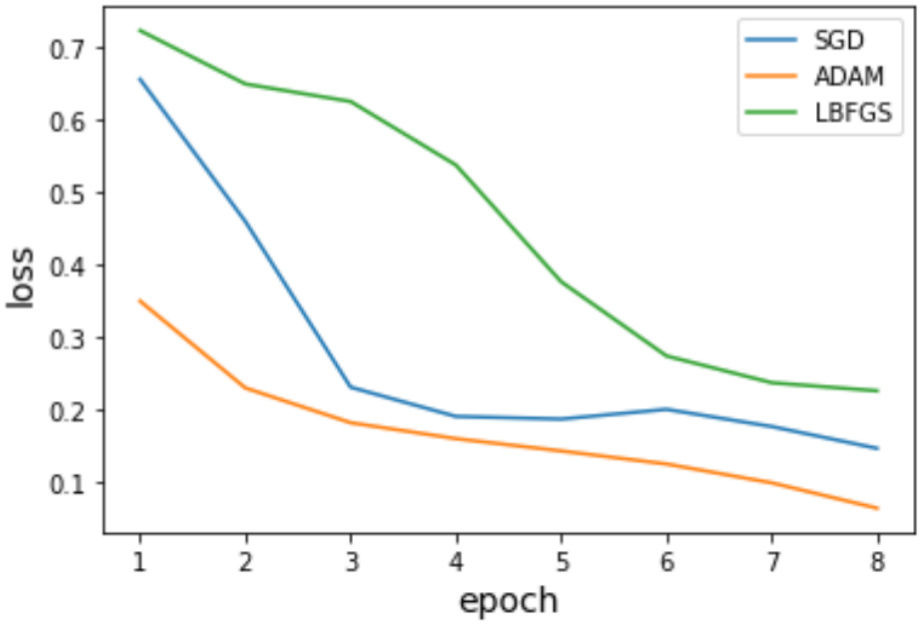

Fig 3 - Loss dynamic during training for SGD, ADAM and LBFGS

According to Fig 3, the learning process with the LBFGS optimizer is much more linear than for gradient based methods. However, all three algorithms need the same amount of epochs to converge (in this case convergence is more about validation accuracy).

6. Conclusion

In this project I successfully implemented two different architectures of the deep learning models. The analysis of the performance showed that complexity of the pretrained model is not worth the performance improvement in comparison to a simple neural network “from scratch”. At least it was the case for this particular task.

The accuracies of the models have not exceeded 0.91 and this limit can be connected with quality of labeling (experts can make mistakes) and loss of information after preprocessing of data. Some images with different classes are too similar to separate.

ADAM optimizer showed better performance with a simple model, without significant overfitting. SGD performed well too, but overfitting took place even with L2 regularization. For this particular dataset this overfitting played well, because it provided high values of recall that aligned with requirements for medical image classifiers. On the pretrained model SGD showed better results.

Computational expenses of the second order optimization method LBFGS has not allowed us to make strong statements about performance of this optimizer because I used only a limited version of it with a small dataset. Overall impression about this optimization method is promising but requires further studies with more computational power.

References

Ian Goodfellow, Yoshua Bengio, Aaron Courville. (2016). Deep Learning. MIT Press

Yaser S. Abu-Mostafa, Malik Magdon-Ismail, Hsuan-Tien Lin. (2012). Learning From Data.

Jiquan Ngiam, Quoc V. Le, Adam Coates.(2011). On optimization methods for deep learning

Richard H. Byrd, Peihuang Lu , Jorge Nocedal. (1996). A Limited Memory Algorithm for Bound-Constrained Optimization.

Diederik P. Kingma, Jimmy Lei Ba. (2015). Adam: a Method for Stochastic Optimization

Daniel Kermany, Kang Zhang, Michael Goldbaum. (2018). Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification