Methods for Deep Learning

ODS Final Project

Shehroz Ali, Alexandra Pilipyuk, Adriano Rasetta

Contents

1. Introduction

In this report, we describe three algorithms: Adam, AdaGrad and Deep Frank-Wolfe. We explore the work of the algorithms, main theoretical results and the performances of these algorithms applied on two real world datasets in a multiclass classification task. We use a feed-forward neural network with two hidden layers of the same size. The model and the three algorithms are implemented on GoogleColab notebook using PyTorch, an open source machine learning library based on the Torch library. The programming language used in our notebook is Python.

1.1 The problem

We consider a multiclass classification problem. We have a data set $(x_i, y_i)_{i\in 1\dots N}$with feature vectors $x_i \in \mathbb{R}^d$ and corresponding labels $y_i \in Y$, $|Y| = m$. The model is represented by a function $f_i(w) = f(x_i, w): \mathbb{R}^d \times \mathbb{R}^p \to \mathbb{R}^m$, where $w \in \mathbb{R}^p$ is a parameter vector. For a sample x_i and a vector w, the model gives a score per element in the output space Y.

We denote a loss function by $\mathcal{L}_i(s) = \mathcal{L}(s, y_i): \mathbb{R}^m \times Y \to \mathbb{R}$ that computes the mistake on the vector of predicted scores and the true answer, where s is a vector of scores for each label $y \in Y$. The learning problem is to find such parameters w that will solve

$$\min_{\mathbf{w}\in\mathbb{R}^p} \frac{1}{N} \sum_{i\in[N]} \mathcal{L}_i(\mathbf{f}_i(\mathbf{w}))$$

In our work we use the Cross-Entropy loss function, that is:

$$\mathcal{L} : (s, y) \mapsto log\left(\sum_{k\in Y} \exp(s_k)\right) - s_y$$

The optimization algorithms considered in this work are based on the idea of Stochastic Gradient Descent (SGD). Given the initial parameters vector w_0, on every step t of SGD a sample j is randomly chosen and the update of parameters w_t is performed by

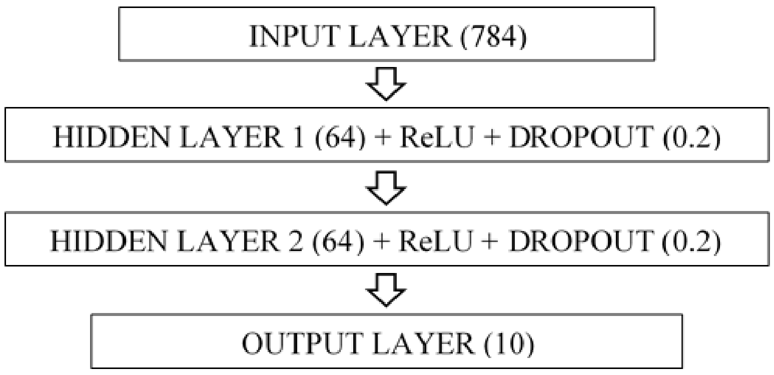
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \left(\partial\mathcal{L}_j(\mathbf{f}_j(\mathbf{w}))\big|_{\mathbf{w}_t} \right)$$

where η_t is a learning rate. In the descriptions of optimizers we denote the vector parameters either by θ or w. g is used to denote the gradient of an objective function. In order to avoid the division by zero in all algorithms, we add a vector with the same dimension of the vector in the denominator with all the element equal to ε. Its value is insignificantly small, in our case $1e^{-8}$. Depending on the context it can be a value or a vector with the same value on all the dimensions.

1.2 The model

In order to solve the classification problem we use a feed-forward neural network with two hidden linear layers of the same size. After each hidden layer we apply a ReLU function and a dropout layer with probability 0.2. Finally, we apply the softmax function to our final output vector and we select the label with the highest probability.

Our model has the following architecture:


INPUT LAYER (784)
HIDDEN LAYER 1 (64) + ReLU + DROPOUT (0.2)
HIDDEN LAYER 2 (64) + ReLU + DROPOUT (0.2)
OUTPUT LAYER (10)

2. The optimizers

When we talk about hyper-parameters, we talk about tuning them manually before training. One of the most important hyper-parameters is the learning rate (η). The learning rate reflects how much we allow the parameter (θ) to follow the opposite direction of the gradient estimate (g). However, it is a difficult task to set the optimal value for the learning rate because if we set it too small then it's loss takes a long time to move towards minima (local or global) with a decreasing trend and if we set it too large then the parameter will move all over the function and may never achieve acceptable loss at all.

Some popular optimizers like Gradient Descent, SGD, and mini-batch SGD use the same learning rate for all the parameters. Although a high-dimensional non-convex nature of neural networks could lead to various sensitivity on each dimension. The learning rate could be too small in some dimensions and could be too large in another dimension. One solution to this problem would be to use different learning rates at each dimension but as we know that deep neural networks could have thousands of dimensions, it is hard to choose different learning rates for all the dimensions. The following optimizers try to find a solution to this problem.

2.1 Adaptive moment estimation algorithm (Adam)

One of the most famous and widely used optimization algorithms nowadays is Adam. It is a combination of RMSprop and SGD with momentum optimizers. It uses the adaptive learning rate derived from the former and the adaptive momentum from the latter. In other words, we have a different learning rate for each connection weight and an exponentially decaying average of previous gradients. It computes these two adaptations from estimates of first (mean) and second (uncentered variance) moments of the gradients:

$$\mathbb{E}[m_t] = \mathbb{E}[g_t] \qquad m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

$$\mathbb{E}[v_t] = \mathbb{E}[g_t^2] \qquad v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$$

The two betas are almost always fixed to $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The two moments are biased towards 0, since we initialize them with 0. Now, we will show the correction for the second moment estimate, the derivation of the first one is analogous. First of all, we rewrite our second moment estimate in this way:

$$v_t = (1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} \cdot g_i^2$$

Now, we consider the expectation of the left-hand and right-hand sides of the previous equation:

$$\begin{aligned}\mathbb{E}[v_t] &= \mathbb{E}\left[(1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} \cdot g_i^2\right] \\ &= \mathbb{E}[g_t^2] \cdot (1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} + \zeta \\ &= \mathbb{E}[g_t^2] \cdot (1-\beta_2^t) + \zeta\end{aligned}$$

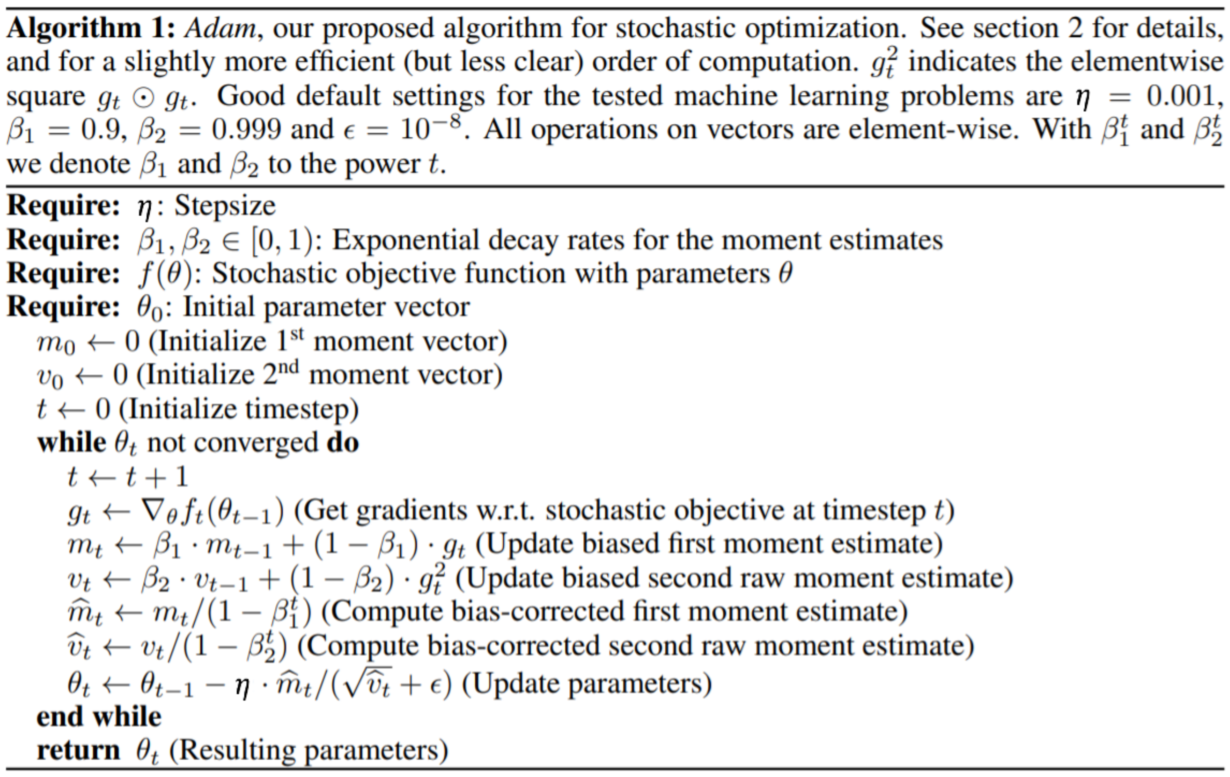
In the second line we approximate g_i with g_t, so we can take it out of sum, since it does not now depend on i . In order to do this, we add an error term ζ that is equal to zero if the second moment is stationary. In the last line we just use the formula for the sum of a finite geometric series. Finally, to obtain our impartial estimator, we simply apply the element wise division by $(1 - \beta_2^t)$. In the end, our two unbiased estimators are:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

The final update rule is:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

The pseudocode of Adam is the following:



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\eta$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \eta \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

2.2 Adaptive gradient algorithm（AdaGrad)

One of the earlier algorithms that has been used to adapt the learning rate to the parameters for each dimension is the AdaGrad algorithm. We can see below the equation for the parameter update that is used in the algorithm:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\varepsilon I + diag(G_t)}} \cdot g_t,$$

It modifies the general learning rate η at each time step t for every parameter $\theta(i)$ to be updated, ε is some small quantity that used to avoid the division of zero, I is the identity matrix and g_t is the gradient estimate in time-step $t$that we can get with the equation:

$$g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(f_i(\theta), y_i)$$

The core of this algorithm is in the matrix G_t, which is the sum of the outer product of the gradients until time-step t, which is determined by equation:

$$G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^\top .$$

Besides, we can use the full matrix G_t in the parameter update, but computing the square root of the full matrix is difficult at high dimensions. Although, computing the square root and the inverse of only the diagonal $diag(G_t)$ is feasible.

After multiplying the effective learning rate matrix with the gradient estimate vector yields the update rule of AdaGrad. Since

$$G_t^{(i,i)} = \sum_{\tau=1}^{t} (g_\tau^{(i)})^2,$$

we can write:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\varepsilon + \sum_{i=1..t} g_i^2}} g_t$$

where $\sum_{i=1..t} g_i^2$ is the sum of squared gradient estimates during the training. In this new equation $\sum_{i=1..t} g_i^2$ represents $diag(G_t)$, which just the specialized case of a more general case with the full matrix G_t.

The Adagrad algorithm performs best for sparse data because it makes big updates for less frequent parameters and a small step for frequent parameters. One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate and results in greater progress along gently sloped directions. AdaGrad's main weakness is its accumulation of the squared gradients in the denominator. Since every added term is positive, the accumulated sum keeps growing during training. This, in turn, causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

The pseudocode of AdaGrad is the following:

```
**Algorithm**: AdaGrad. $g_t$ is the gradient at the time step t, $G_t$ is the matrix with the sum of the outer product of the gradients until timestep t, $\eta$ is the step size and $\varepsilon = 10^{-8}$.

**Require:** $\eta$: Stepsize
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
    $t \leftarrow 0$ (Initial timestep)
    $G_t \leftarrow 0$ (Intial $G$ matrix)
    **while** $\theta_t$ not converged **do**
        $t \leftarrow t + 1$
        $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t stochastic objective at timestep $t$)
        $G_t = G_{t-1} + g_t g_t^T$
        $\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\varepsilon I + diag(G_t)}} g_t$
    **end while**
```

2.3 Deep Frank-Wolf algorithm (DFW)

Stochastic gradient descent (SGD) has been the most generally applied optimization algorithm for deep neural networks due to its excellent performance on most of the learning tasks. In order to improve the validation performance the learning rate scheduling technique is required to adjust its value during training by reducing it according to a predefined schedule. However, there are no functions that can be used to design schedules that are hyper-parameter free and guaranteed to converge to the optimal solution of the problem of a deep neural network.

Although several schedules were developed that show good results on their particular learning tasks, there is no general methodology. To solve this problem, adaptive gradient methods have been developed and borrowed from online convex optimization. However, such methods achieve worse generalization as compared to SGD.

This performance gap between existing adaptive methods and SGD can be fulfilled by an optimization algorithm called Deep Frank-Wolfe (DFW) which is an extension of the Frank-Wolfe (FW) algorithm specifically used for deep neural networks. Each iteration of the algorithm exploits more information about the learning objective, while preserving the same computational cost as SGD. An optimal step-size is computed in closed-form by using the FW algorithm in the dual so no hand-designed schedule for the learning rate is needed.

The step of SGD algorithm with a regularization $\rho(w)$ can be represented by

$$\mathbf{w}_{t+1} = \underset{\mathbf{w}\in\mathbb{R}^p}{\arg\min}\left\{\frac{1}{2\eta_t}\|\mathbf{w}-\mathbf{w}_t\|^2 + \mathcal{T}_{\mathbf{w}_t}\rho(\mathbf{w}) + \mathcal{T}_{\mathbf{w}_t}[\mathcal{L}_j(\mathbf{f}_j(\mathbf{w}))]\right\}$$

where the objective has been linearized by the first-order Taylor expansion around the point w_t: $\mathcal{T}_{w_t} f(w) = f(w_t) + (\partial f(w)|_{w_t})^T(w - w_t)$. The Deep Frank-Wolfe algorithm introduces the proximal problem that linearizes the regularization and the model, but not the loss function. So on every step in order to perform an update on the parameters we face the problem of finding a minimum

$$\min_{\mathbf{w}\in\mathbb{R}^p}\left\{\frac{1}{2\eta_t}\|\mathbf{w}-\mathbf{w}_t\|^2 + \mathcal{T}_{\mathbf{w}_t}\rho(\mathbf{w}) + \mathcal{L}_j(\mathcal{T}_{\mathbf{w}_t}\mathbf{f}_j(\mathbf{w}))\right\}$$

The described approach can be applied to any loss function that is convex and piecewise linear. η_t is a learning rate; the learning rate is considered fixed during the steps $\eta_t = \eta$ as the algorithm achieves a good performance by linearizing the regularization and the model, but not the loss function.

The scheme of the algorithm is following:

```
Algorithm    The Deep Frank-Wolfe Algorithm
Require: proximal coefficient $\eta$, initial point $\mathbf{w}_0 \in \mathbb{R}^p$, momentum coefficient $\mu$, number of epochs
 1: $t = 0$
 2: $\mathbf{z}_0 = 0$                                                   ▷ Momentum velocity (initialization)
 3: for each epoch do
 4:     for each mini-batch $\mathcal{B}$ do
 5:         Receive data of mini-batch $(\mathbf{x}_i, y_i)_{i \in \mathcal{B}}$
 6:         $\forall i \in \mathcal{B},\ \mathbf{b}_t^{(i)}(\mathbf{w}_t) = (f_{\mathbf{x}_i,\bar{y}}(\mathbf{w}_t) - f_{\mathbf{x}_i,y_i}(\mathbf{w}_t) + \Delta(\bar{y}, y_i))_{\bar{y} \in \mathcal{Y}}$    ▷ Forward pass
 7:         $\forall i \in \mathcal{B},\ \mathbf{s}_t^{(i)} \leftarrow \texttt{get\_s}(\mathbf{b}_t^{(i)}(\mathbf{w}_t))$    ▷ Dual direction
 8:         $\boldsymbol{\delta}_t = \partial \left( \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (\mathbf{s}_t^{(i)})^\top \mathbf{b}_t^{(i)}(\mathbf{w}) \right) \Big|_{\mathbf{w}_t}$    ▷ Derivative of (smoothed) loss function
 9:         $\mathrm{r}_t = \partial \rho(\mathbf{w}) \big|_{\mathbf{w}_t}$    ▷ Derivative of regularization
10:         $\gamma_t = (-\eta \boldsymbol{\delta}_t^\top \mathrm{r}_t + \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (\mathbf{s}_t^{(i)})^\top \mathbf{b}_t^{(i)}(\mathbf{w}_t) / (\eta \|\boldsymbol{\delta}_t\|^2)$ clipped to [0, 1]    ▷ Step-size
11:         $\mathbf{z}_{t+1} = \mu \mathbf{z}_t - \eta \gamma_t (\mathrm{r}_t + \boldsymbol{\delta}_t)$    ▷ Velocity accumulation
12:         $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left[ \mathrm{r}_t + \gamma_t \boldsymbol{\delta}_t \right] + \mu \mathbf{z}_{t+1}$    ▷ Parameters update
13:         $t = t + 1$
14:     end for
15: end for
```

In order to write a Lagrangian dual we introduce the following notation. For any $y, \bar{y} \in Y$ let

$$\Delta(\bar{y}, y) = \begin{cases} 0 & \text{if } \bar{y} = y, \\ 1 & \text{otherwise.} \end{cases}$$

$$a_y = \partial \rho(w)|_{w_0} + \partial f_{i,y}(w)|_{w_0} - \partial f_{i,y_i}(w)|_{w_0}$$

$$b_y = f_{i,y}(w_0) - f_{i,y_i}(w_0) + \triangle\,(y, y_i)$$

Then the dual for the initial problem will be given as:

$$\max_{\boldsymbol{\alpha} \in \mathcal{P}} \left\{ -\frac{1}{2\eta} \|A\boldsymbol{\alpha}\|^2 + \mathbf{b}^\top \boldsymbol{\alpha} \right\}$$

Where $A = (\eta a_y)_{y \in Y}$, $b = (b_y)_{y \in Y}$, $\mathcal{P} = \{\alpha \in \mathbb{R}_+^m : \sum_{y \in Y} \alpha_y = 1\}$. Given the variables α the primal will be computed as $w = w_t - A\alpha$.

If a single step is performed on the dual, it's conditional gradient is given by $-\partial \rho(w) + \partial(\mathcal{L}_j(f_j(w)))|_{w_t}$. Given the step-size γ_t, the update can be written as:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left[\partial \rho(\mathbf{w}) \big|_{\mathbf{w}_t} + \gamma_t \partial \mathcal{L}_j(\mathbf{f}_j(\mathbf{w})) \big|_{\mathbf{w}_t} \right]$$

An optimal step-size $\gamma_t \in [0, 1]$ is computed in a closed-form on every step. While doing a step in the direction of s_t in the dual, the step $\alpha_{t+1} = \alpha_t + \gamma_t(s_t - \alpha_t)$ is done. So the task is to maximize $\frac{1}{2\eta} ||A\alpha_{t+1}||^2 + b^T \alpha_{t+1}$. In order to find the optimal γ_t, we need to compute the derivative of this expression with respect to γ_t and set it to zero. This will yield an expression of γ_t that can be presented in the following form:

$$\gamma_t = \frac{-\eta \boldsymbol{\delta}_t^\top \mathrm{r}_t + \mathbf{s}_t^\top \mathbf{b}}{\eta \|\boldsymbol{\delta}_t\|^2} \textit{ clipped to [0, 1]}$$

If $\gamma_t = 1$, the update of parameters is the same as the one obtained by SGD. To accelerate the algorithm, the Nesterov Momentum is adapted to the DFW algorithm. The velocity z_t accumulates the step along the conditional gradient, scaled by $\eta\gamma_t$:

$$\mathbf{z}_{t+1} = \mu \mathbf{z}_t - \eta\gamma_t(\mathrm{r}_t + \boldsymbol{\delta}_t).$$

So the final parameters update is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left[\partial\rho(\mathbf{w})\big|_{\mathbf{w}_t} + \gamma_t \partial\mathcal{L}_j(\mathbf{f}_j(\mathbf{w}))\big|_{\mathbf{w}_t} \right] + \mu \mathbf{z}_t + 1$$

We replace $\mathcal{T}_{w_0} f_x(w)$ by $f_x(w_0)$, which makes computations inexpensive. Since $||\mathcal{T}_{w_0} f_x(w) - f_x(w_0)|| = O(||w_t - w_0||)$ and this is typically very small value, we can consider that $\mathcal{T}_{w_0} f_x(w) \simeq f_x(w_0)$.

3. Dataset

In order to test our algorithms we use two dataset: Digit MNIST and Fashion MNIST. Both consist of 60,000 samples for the train set and 10,000 for the test set. Each sample is a 28x28 grayscale image, so they have 784 features, with an associated label belonging to one of 10 possible classes. In the case of Digit MNIST these are digits from 0 to 9, in the case of Fashion MNIST it refers to clothing items.

We split the train set in two parts in order to obtain a validation set of 10,000 items to tune the learning rate hyperparameter. We use a grid search with 50 epochs for each algorithm to obtain the best learning rates. Then, we train our three best models, one for each optimizer, on the entire train set. Finally, we test our models on the test set. We compare them considering the final loss value and the accuracy.

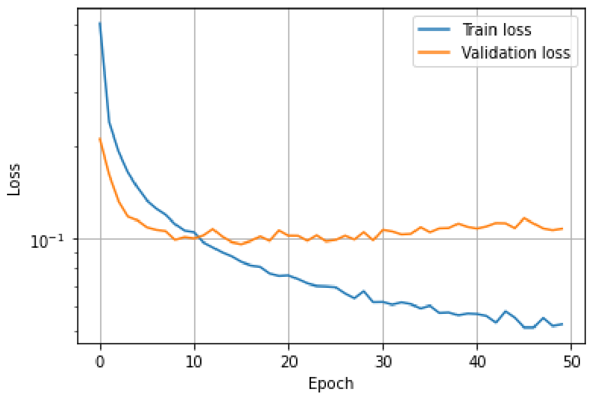
Considering the approach adopted in Berrada et al. (2018), we have chosen the achievement of 100 epochs as the stopping criterion. In the paper, 200 and 300 epochs were used in the experiments. For computational cost reasons we have reduced this number.

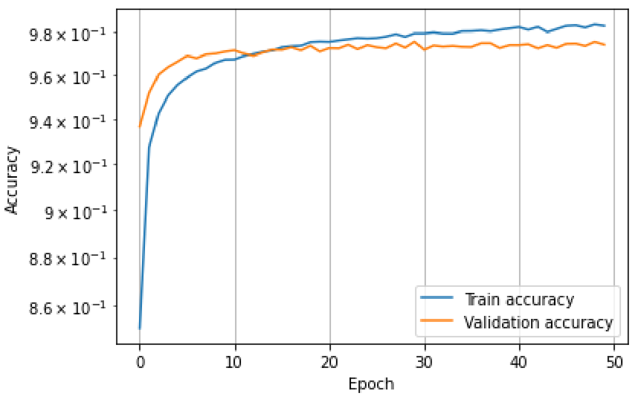
4. Results

In this section we present the results obtained with the three optimizers providing for each a graph on loss and accuracy with the respective best selected hyperparameter. A comparison is also proposed between the three optimizers on the progress of learning on the training set. Finally, in a conclusive table we show the results obtained on the test set.

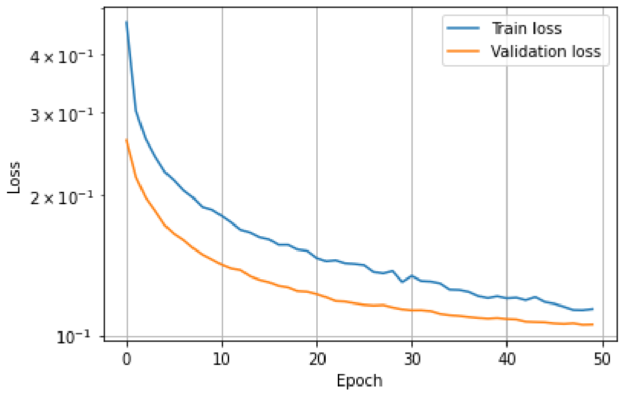
4.1 Digit MNIST

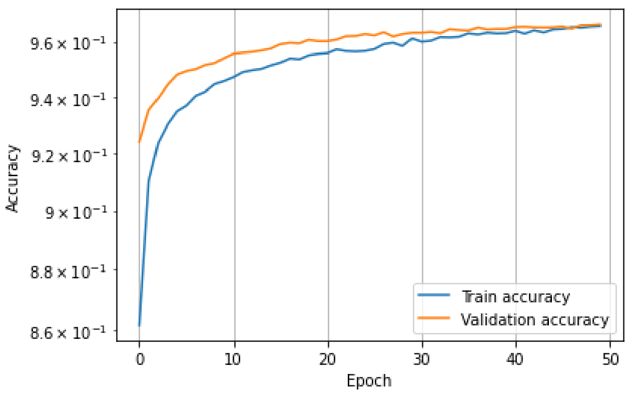
- **Adam**: the best starting value for the learning rate is 0.001. The best validation loss and validation accuracy are 0.108 and 97.41% respectively. The following plots represent the learning trend and the performance of the model on the validation set.



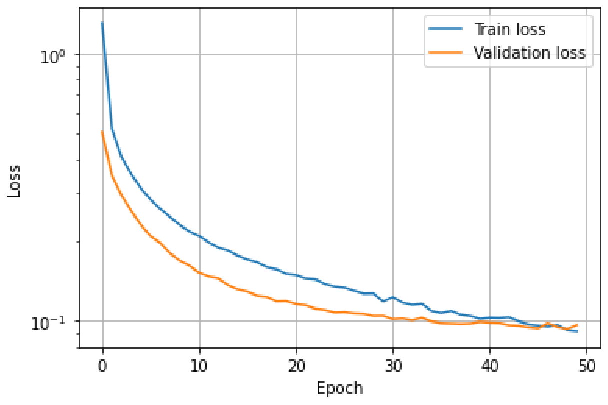


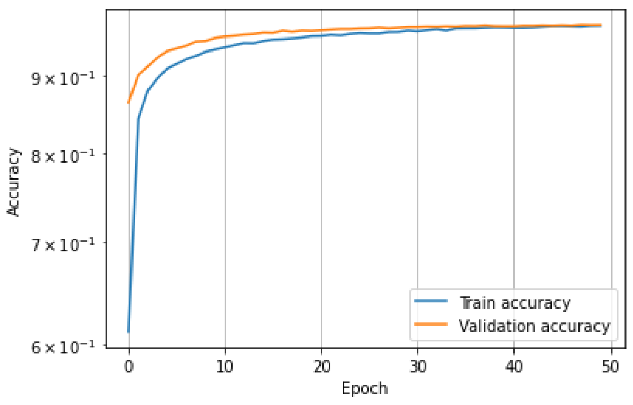
- **AdaGrad**: the best starting value for the learning rate is 0.01. The best validation loss and validation accuracy are 0.105 and 96,63% respectively.



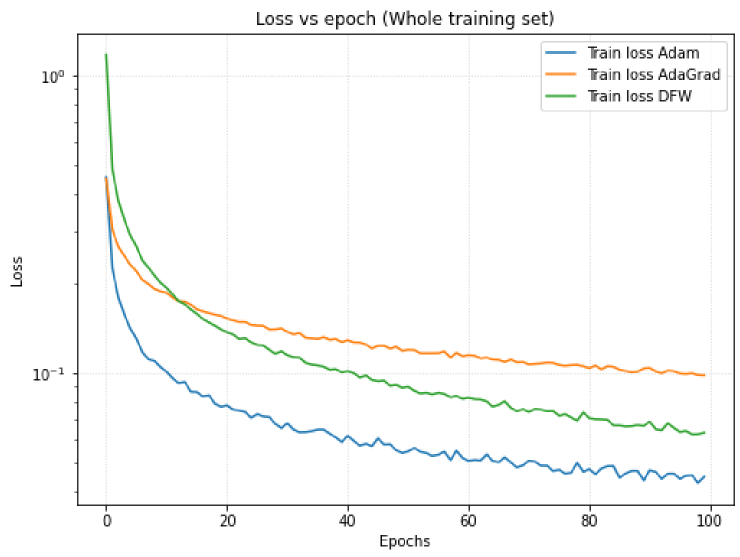


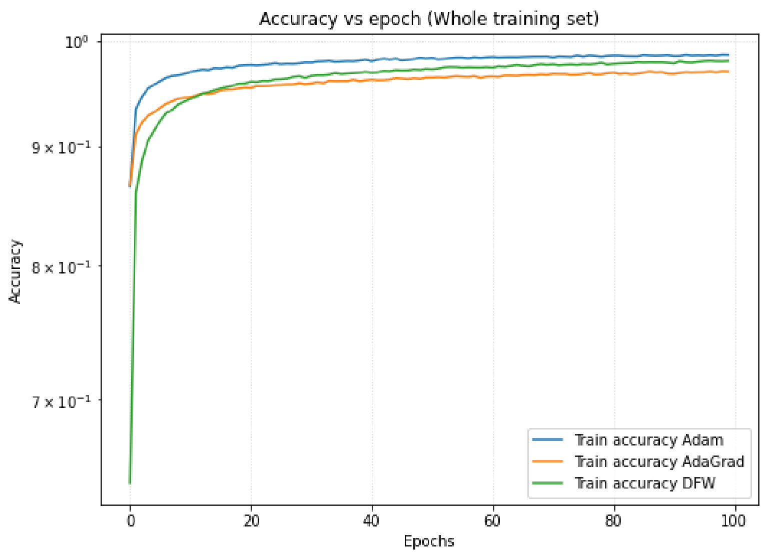
- **Deep Frank-Wolf**: the best starting value for the learning rate is 0.01. The best validation loss and validation accuracy are 0.096 and 97.21% respectively.





In the following part we have the comparison between the three algorithms and the table with the final results on the test set.

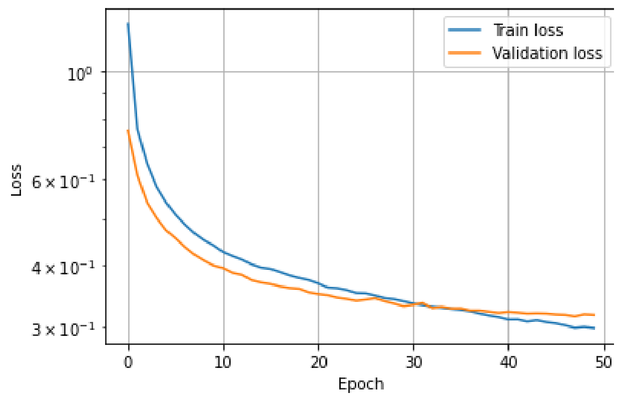


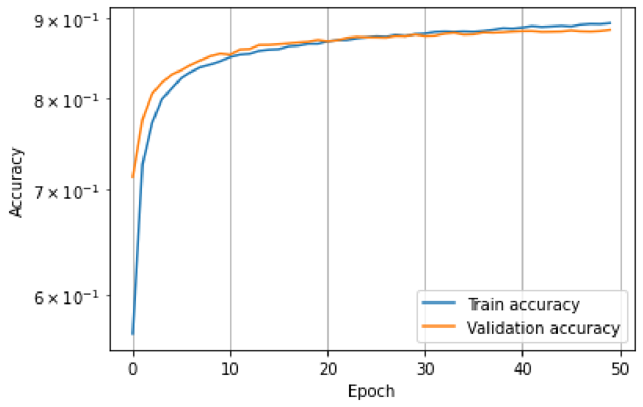


Optimizer	*Test loss*	*Test accuracy (%)*
Adam	0.129	97.52
AdaGrad	0.088	97.36
Deep Frank-Wolf	**0.083**	**97.54**

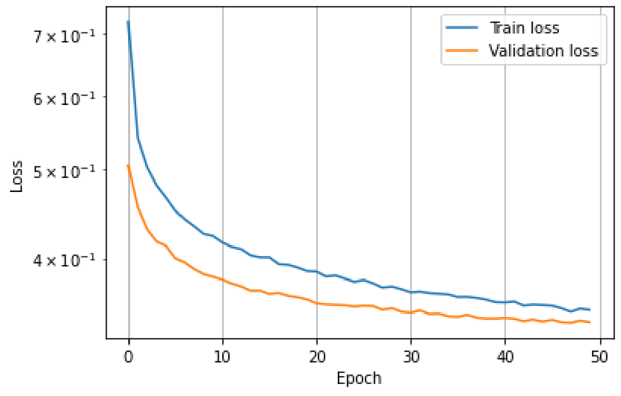
4.2 Fashion MNIST

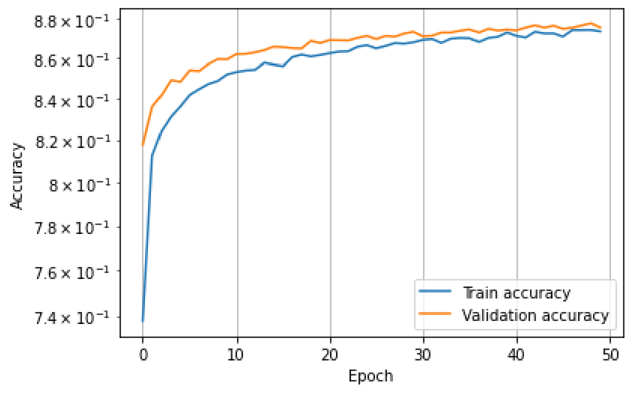
- **Adam**: the best starting value for the learning rate is 0.0001. The best validation loss and validation accuracy are 0.317 and 88.42% respectively. The following plots represent the learning trend and the performance of the model on the validation set.



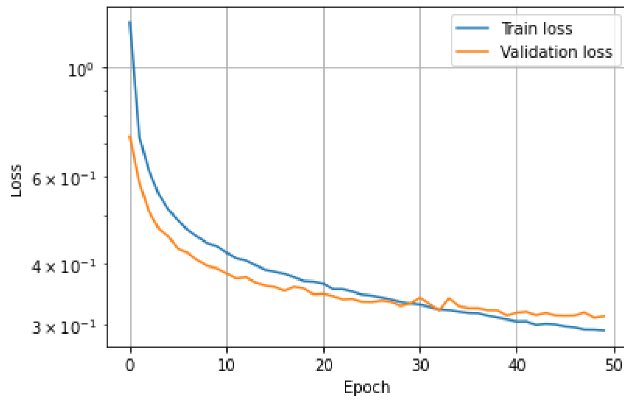


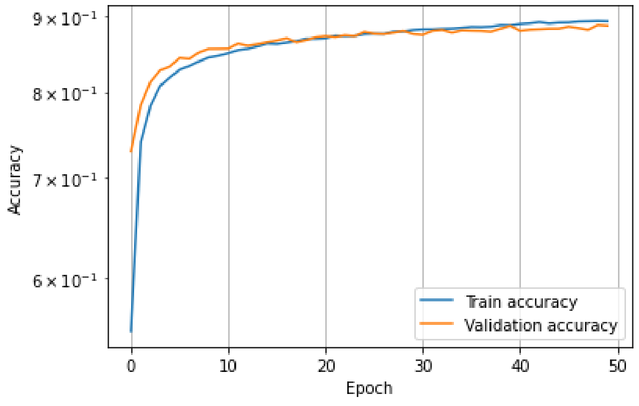
- **AdaGrad**: the best starting value for the learning rate is 0.01. The best validation loss and validation accuracy are 0.341 and 87,56% respectively.



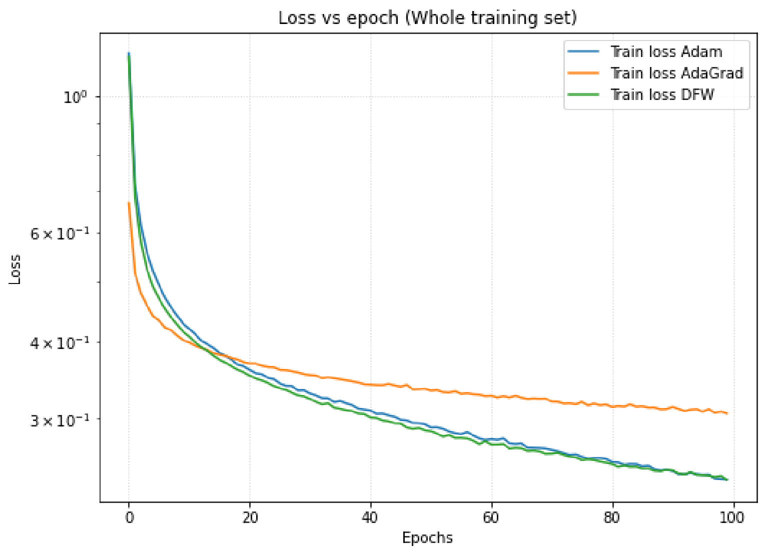


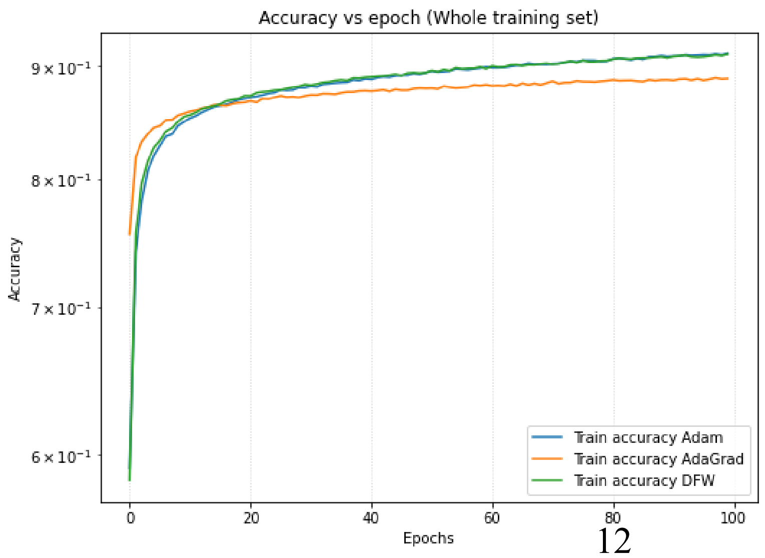
- **Deep Frank-Wolf**: the best starting value for the learning rate is 0.01. The best validation loss and validation accuracy are 0.312 and 88.71% respectively.





In the following part we have the comparison between the three algorithms and the table with the final results on the test set.





Optimizer	*Test loss*	*Test accuracy (%)*
Adam	**0.333**	**88.65**
AdaGrad	0.343	87.70
Deep Frank-Wolf	0.350	88.16

5. Conclusion

All optimizers showed good training behavior in both datasets. The results on the test sets can be considered satisfactory considering the simplicity of the model used. The comparison between optimizers on the training set showed that Adam and DFW perform better than Adagrad. Only in Digit MNIST, Adam is better than DFW. In the final tests, in Digit MNIST, DFW is the best, while in Fashion MNIST Adam presents the best results.

References

1. Berrada, L., Zisserman, A., & Kumar, M. P. (2018). Deep frank-wolfe for neural network optimization. *arXiv preprint arXiv:1811.07591*.
2. Duchi, J. C., Hazan, E. & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.*, 12, 2121-2159.
3. Kingma, D.P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. *CoRR, abs/1412.6980*.