Assignment 2 - Artificial Intelligence

Theodora Gaiceanu

February 21, 2022

1 Description of the solution

- **Linear Regression** - Preprocessing

  I trained the data using the French dataset.First of all, one needs to create the **X** matrix and the **y** vector. **X** is an array with as many rows as the dataset and 2 columns, the first column being only for the intercept and the second column is actually the first column from the dataset. **y** has as many rows as the dataset and only one column, which has the values of the second column from the dataset.

  Then one needs to scale the arrays so that all the values are inside the interval $[0, 1]$. One simple method to scale the arrays is to divide them by their maximum value. Therefore, we will have that $X_{scaled} = \frac{X}{max(X)}$ and $y_{scaled} = \frac{y}{max(y)}$.

- **Linear Regression - Batch Gradient Descent** According to [1], the batch gradient descent takes into account all the examples of the dataset. Therefore, its update rule is the following:

$$w = w + \frac{\alpha}{q} X^T (y - Xw), \tag{1}$$

  where w is the weight vector, X is the feature vector, y is the output vector, α is the learning rate, q is the number of observations from X.

  The function *fit_batch* implements the batch descent algorithm. It has as parameters the following: X, y, α, w, the number of epochs used for training, the precision for the training (how small should be the product $X^T * (y - X * w)$. First, the learning rate α is divided by the number of observations. Then, for each epoch, the loss is computed (the difference between the output y and the prediction $X^T * w$. The update rule is computed according to Equation (1) and one checks if the L2 norm of the product $X^T * (y - X * w)$ is smaller than the precision (if the error if small enough). If it is, then the algorithm should stop and output the trained weights.

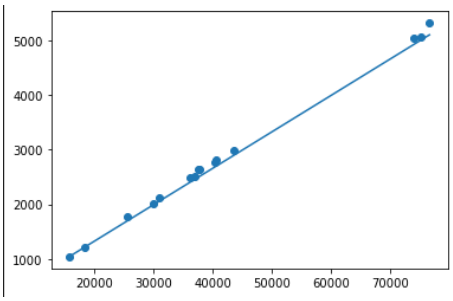

Figure 1: BGD Results for the French data

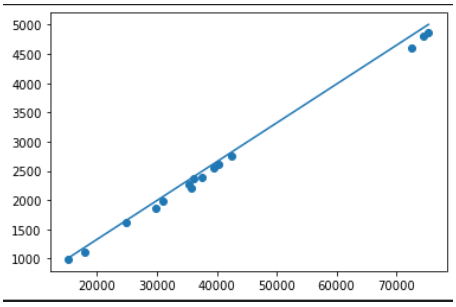

Figure 2: BGD Results for the English data

For training, I used a learning rate of 0.0001 and 2180 epochs. I chose a very accurate precision, of 1^{-12}. I used the French dataset for training and I evaluated the results on both French and English datasets. The resulted weights can be observed in Listing 1.

Listing 1: Batch gradient descent - resulted weights

```
Epoch reached 2179
Weights [[0.10055643]
 [0.06642213]]
```

As it can be observed in Figures 1 and 2, the model fits perfectly the French data. For the English data, the slope is a little shifted up, but it still intersects the majority of the data points.

- **Linear Regression - Stochastic Gradient Descent** According to [1], the stochastic gradient descent does a weight update using one example at a time. Therefore, its update rule is the following:

$$w = w + \alpha(y^j - x^j w)x^j, \tag{2}$$

where w is the weight vector, x^j is the j^{th} observation of the feature vector, y^j is the j^{th} value of the output vector, α is the learning rate.

The function *fit_stoch* implements the stochastic descent algorithm. It has as parameters the following: X, y, α, w, the number of epochs used for training, the precision for the training (how small should be the product $X_j^T * (y_j - X_j * w)$. For each considered epoch, the indexes of the observations X are shuffled randomly. Then, for each observation, the prediction is computed as the product of $X_j * w$. The loss is computed as the difference between the real value y_j and the prediction. The update rule is computed according to Equation (2) and one checks if the L2 norm of the product $X_j^T * (y_j - X_j * w)$ is smaller than the precision (if the error if small enough). If it is, then the algorithm should stop and output the trained weights.

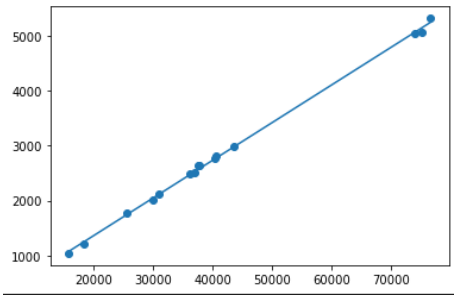

Figure 3: SGD Results for the French data

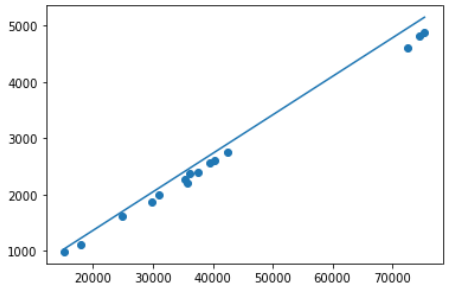

Figure 4: SGD Results for the English data

For training, I used the French dataset. The learning rate is 0.00001 (a bit smaller than the learning rate used for Batch Gradient Descent) and the number of epochs used is 1500 (less epochs needed compared to the Batch Gradient Descent, Stochastic Gradient Descent converged faster) . I used again a very precise error limit, 1^{-10}. First, I tested the trained weights on the French dataset and after I tested them on the English dataset as well. The resulted weights can be observed in Listing 2. It can be noticed that the values are very similar to the weights obtained when using Batch Gradient Descent.

Listing 2: Stochastic gradient descent - resulted weights

```
Epoch reached 1499
Weights [[0.10331224]
 [0.06828671]]
```

Again, from Figures 3 and 4, the model fits the French dataset perfectly and touches the most points for the English dataset. The results are similar to batch gradient descent model.

- **Linear Regression - Observations and possible improvements** In general, both batch and stochastic gradient descent can fit the data in a satisfactory manner. Anyway, as it is known from [1], stochastic gradient descent is more efficient as it updates the weights one observation at a time. This fact is not so evident for this example, as the dataset is not so big, but it can make a difference when using it for more complex problems. Another remark valid for both algorithms is that one should pay attention at the learning rate. A big learning rate can skip the minimum, while a too small learning rate can make the algorithms get stuck in a local minimum. There is no general rule for choosing a good value for the learning rate, one should just try different values and observe the results. A possible improvement to this solution is to use stochastic gradient descent with minibatches. This one can assure a more stable convergence, as it reduces the variance of the weights updates and is also efficient.

- **Classification - The Perceptron** According to [2], the perceptron up-

date rule is:

$$w_i = w_i + \alpha(y - h_w(x))x_i, \tag{3}$$

where w is the weight, x is the features matrix, α is the learning rate, and $h_w(x)$ is the threshold function.

Therefore, one needs a *predict* function that returns the prediction given the features matrix X and the weight vector w (these two are the parameters of the function). So one must iterate through all the rows of X and compute the product $X_r oww$ for each row. If this one is higher than 0, than the predicted class is 1, otherwise it is 0. The function returns the vector of predicted classes.

Then one need a *fit* function that returns the updated weights given the X matrix of observations, the y output vector, the maximum number of epochs, the maximum accepted number of misclassifications. The algorithm used is Stochastic Gradient Descent. For each considered epoch, the indexes of the observations X are shuffled randomly. Then, for each observation, the prediction is computed by calling the function *predict* with the current observation row and the weight vector. The loss is computed as the difference between the real current value y_i and the prediction. The update rule is computed according to Equation (3). Then one sums all the differences between the predictions and the target values in order to determine the number of misclassifications. If this one is equals to the number of accepted misclassifications, then the aglorithm is stopped and the function returns the trained weights.

I used a learning rate of 0.1, a maximum number of epochs of 1000 (but the algorithm converged after only 33 epochs) and no accepted misclassifications. The resulted weights can be seen in the Listing below.

Listing 3: Perceptron Classification - resulted weights

```
Epoch 33
array([[ 0.        ],
       [-0.41593223],
       [ 0.43814006]])
```

The results of the training can be observed in Figure 5. The decision boundary separates the French data (red dots) from the English data (blue dots). The separation is almost perfect, excepting the dots from the lower left corner, which are almost impossible to separate with a linear decision boundary, as they are intercalated.

- **Classification - The Perceptron - Evaluation** In order to evaluate the algorithm, one need to implement a leave-one-out cross validation method. This function, named *leave_one_out_cross_val* has three parameters: the observations matrix, the output vector and the function used for

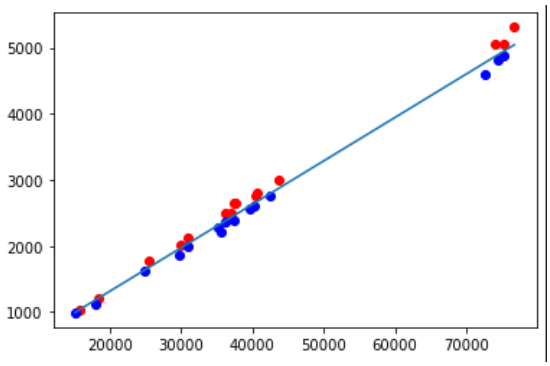

Figure 5: Perceptron classifications - visualisation of the results

training the weights. As we are asked to train 30 models, one iterates 30 times. Each time, one has to save an observation row and an output value for testing and the remaining 29 values for training. Then one calls the function used for training the weights for the current training observation row and output value. After that, one gets the prediction for the current testing observation row and the resulted weights. If the prediction corresponds to the current testing output value, the number of correct values is incremented. The function returns the percentage of correct values.

The result of the evaluation can be seen in Listing 4. For each model (fold), one can observe the number of epochs needed for training and if the resulted values are good or bad. The cross-validation accuracy is 96.67%, which is exactly $\frac{29}{30}$ (this means that all the models gave the correct results).

Listing 4: Perceptron Classification - Cross-validation results

```
Epoch 35
Fold 0 of 30: Correct
Epoch 12
Fold 1 of 30: Correct
Epoch 21
Fold 2 of 30: Correct
Epoch 51
...
Epoch 24
Epoch 24
Fold 28 of 30: Correct
Epoch 38
Fold 29 of 30: Correct
Cross-validation accuracy (stochastic): 0.9666666666666667
```

- **Classification - Logistic Regression** According to [2], the logistic regression update rule is:

$$w_i = w_i + \alpha(y - h_w(x))h_w(x)(1 - h_w(x))x_i, \tag{4}$$

where w is the weight, x is the features matrix, α is the learning rate, and $h_w(x)$ is the logistic function.

First, one needs to define the *logistic* function given an argument x. This function should return the mathematical expression of the logistic function, $\frac{1}{1+\exp -x}$ or 0, in case of an Overflow error.

Then one need a function that should return a vector with the probabilities $P(1|x_i)$ for all the rows of X. This function is named *predict_proba* and it has the observation matrix and the weight vector as arguments. One iterates through all rows of the observation matrix and store the product of the current observation row and the weight vector. The function returns a vector with the logistic function values of the before mentioned product.

Another *predict* function is needed. This one has again the same arguments (the observation matrix and the weight vector), but this one works like a threshold. It iterates through all the $P(1|x)$ predictions obtained from the *predict_proba* function. It checks if the value is higher than 0.5 and if it is, it appends a 1 value inside a vector of predictions. If it is not, it appends a 0 value inside the vector of predictions. The function returns the vector of predictions.

Last but not least, one needs a *fit* function that has as parameters the observations matrix, the output vector, the learning rate, the maximum number of epochs, the permitted error and returns the trained weights. For this task I used the Batch Gradient Descent version. For each epoch, the prediction is computed by calling the *predict_proba* function with arguments X and w. The update rule is computed according to Equation (4) and one checks if the sum of the three terms of the gradient (the product $X^T((y - h_w(x)h_w(x)(1 - h_w(x))))$ is smaller than the precision (if the error if small enough). If it is, then the algorithm should stop and output the trained weights.

The resulted weights can be observed in Listing 5. I used a learning rate of 0.5 (compared to 0.1 for the perceptron), a maximum number of 5000 epochs and a precision of 1^{-5}. Anyway, the training took much longer compared to the perceptron case (4999 epochs vs. 33). This was expected in a way, as for the perceptron I used the Stochastic Descent version.

Listing 5: Logistic Regression Classification - resulted weights

```
Epoch 4999
array([[ -0.14283107],
       [-48.30246758],
       [ 50.67207226]])
```

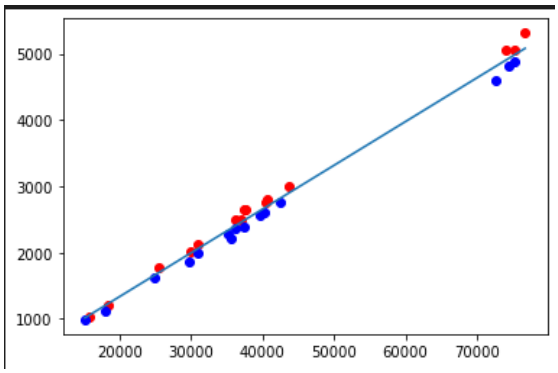

Figure 6: Logistic regression classifications - visualisation of the results

The decision boundary for the Logitic Regression can be seen in Figure 6. Again, the red points from the lower left corner are not separated well, as they are too similar to the blue points. But, for the other points, the classification is done correctly, similar to the perceptron case.

- **Classification - Logistic Regression - Evaluation** The evaluation of the algorithm is done by using again the leave-one-out cross validation method. The method follows exactly the same steps described above for the perceptron evaluation. So again there are 29 models that are trained and there is one model used for evaluation. The results of the cross-validation evaluation can be observed in the Listing below.

Listing 6: Logistic Regression Classification - Cross-validation results

```
Epoch 4999
Fold 0 of 30: Correct
Epoch 4999
Fold 1 of 30: Correct
Epoch 4999
Fold 2 of 30: Correct
Epoch 4999
...
Fold 28 of 30: Correct
Epoch 4999
Fold 29 of 30: Wrong
Cross-validation accuracy (batch): 0.9333333333333333
```

The resulted accuracy is 93.33%, as two models give a wrong classification (compared to the perceptron case, where all the results were correct). Also, one should observe that all the trainings last 4999 epochs (compared to the perceptron case, where the majority of trainings lasted between 24 and 35 epochs, the longest one taking 52 epochs). This is in accordance to what it was noticed for the regression part also (the stochastic variant of the gradient descent converges quickly). The learning rates used are somehow similar this time (0.1 for the perceptron and 0.5 for the logistic

regression). Also, for the logistic regression, one can inspect the logistic surface. This one can be observed in Figure 7. The output of the prediction (the logistic function) is a probability of belonging to the class labeled as 1 (which is the red class from the surface - the French class). If the input is in the center of the separation region, it has a probability of 0.5 and it becomes closer to 1 (if the input is a French chapter) or to 0 (if the input is an English chapter). The logistic function is differentiable.

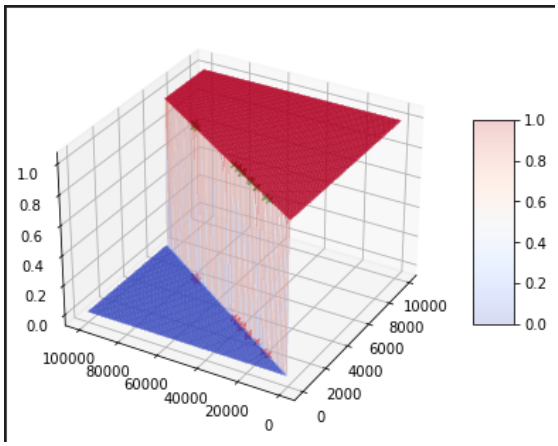

Figure 7: Logistic regression classifications - visualisation of the logistic surface

- **Classification - Logistic Regression - sklearn**

  Sklearn is a popular API that can be used for logistic regression. The function *LogisticRegression* creates a logistic regression model, the function *fit* trains the model and the function *predict* shows the predictions of the trained model. Using the same dataset, one gets the following predictions of the X with the built-in functions from Sklearn:

Listing 7: Logistic Regression Classification - Predicted classes of the X using Sklearn functions

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0.,0.,1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1.])
```

The class probabilites of the X using Sklearn and the function *predict_proba* (which is the output of the logistic of Xw) can be observed below:

Listing 8: Logistic Regression Classification - Predicted class probabilities of the X using Sklearn function *predict_proba*

```
array([[1.00000000e+00, 1.28980319e-30],
       [9.99999999e-01, 8.16295157e-10],
       [9.91302434e-01, 8.69756611e-03],
       [1.00000000e+00, 2.35657080e-12],
       ...
       [0.00000000e+00, 1.00000000e+00],
       [1.80182968e-02, 9.81981703e-01]])
```

- **Classification - Logistic Regression - Keras**

  Keras is another popular API that can be used for logistic regression. Here, the classes *Sequential* and *Dense* are used to build the model, so here a neural network is used. The optimizer used is Nadam and the loss used is binary crossentropy, as this is a classification problem. The *fit* function is used to train the model and the *predict* function is used to make predictions. By using the Keras built-in functions, one gets the following predictions:

Listing 9: Logistic Regression Classification - Predicted classes of the X using Keras function *predict* with the threshold 0.5

```
array([[0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1],
       [1]])
```

It can be observed that both Sklearn and Keras give the same predictions, which was expected as they both use the same dataset.

- **Classification - Logistic Regression - PyTorch**

  PyTorch is another popular API that can be used for solving logistic regression. The notebook uses again the same dataset to make the classifi-

cation. The PyTorch built-in functions are also used for normalizing and scaling the data. The notebook defines a neural network model equivalent to logistic regression and the training set is stored as PyTorch tensors. The used loss is binary cross entropy and the optimizer used is stochastic gradient descent. The model is trained using the built-in function *train*. The predictions can be seen below:

Listing 10: Logistic Regression Classification - Predicted classes of the X using PyTorch with the threshold 0.5

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1.])
```

It can be observed that the results are the same.

- **Classification - possible improvements**

  There are more ways to solve a classification problem. One possible improvement for the perceptron and for the logistic regression solution would be to use a different kind of optimizer. For example, one can use Adam as it is more stable and it converges faster. Or one can use stochastic gradient descent with minibatches, as it combines the advantages of both batch gradient descent and stochastic gradient descent. Moreover, one can also use neural networks models for solving the classification problem.

2 Paper Summary - An overview of gradient descent optimization algorithms

According to [3], gradient descent algorithms are first order algorithms (based on gradient) commonly used in the optimisation of neural networks. At this kind of algorithms, the minimum of a parameterized function $J(w)$ is found by going through the inverse direction of the gradient ($\nabla_w J(w)$). The necessary number of steps to find the extreme point is determined by the learning rate α. The most popular gradient descent algorithms are the following:

Batch gradient descent It is also called vanilla gradient descent. The iterative formulation is:

$$w = w - \alpha \nabla_w J(w) \tag{5}$$

As it could be observed from the mathematical expression, the algorithm is quite slow because it requires the computation of the gradients over the entire data set.

Stochastic gradient descent (SGD) With respect to [3], the modification brought by this algorithm resides in the fact that the parameters are updated at each input $x(i)$ and output $y(i)$ of the training data set:

$$w = w - \alpha * \nabla_w J(w, x^{(i)}, y^{(i)}) \tag{6}$$

In this way, SGD is faster and it may be used in real time. The algorithm's frequent updates with high variance determine big fluctuations of the objective function.

Generally speaking, the fluctuations allow the algorithm to find a possibly better local minimum. On the other hand, these fluctuations make the convergence at a specific minimal point more complex.

Mini-batch gradient descent It combines the advantages of both batch gradient descent and stochastic gradient descent. Therefore, the algorithm assures a more stable convergence by reducing the variance of the parameter updates and it is efficient, as it can use deep learning libraries for matrix optimizations [3]. The updating rule is:

$$w = w - \alpha \nabla_w J(w; x^{(i:i+n)}; y^{(i:i+n)} \tag{7}$$

Momentum SGD has some problems in areas around local optima, problem that could make it get stuck. Momentum makes SGD faster in proper directions and dampens the oscillations in areas close to local minima. In order to do this, it adds a fraction γ of the update vector of the past time step to the current update vector.

$$v_t = \gamma v_{t-1} + \alpha \nabla_w J(w) \tag{8}$$

$$w = w - v_t \tag{9}$$

Nesterov accelerated gradient (NAG) Anyway, momentum does not know what comes next, for example it does not know when the hill slopes up again, so it does not slow down before this happens. NAG adapts to the surface. The term $w - \gamma v_{t-1}$ can be used to approximate the future position of the parameters. Therefore, the gradient can be computed taking into consideration the approximate future position of the parameters.

$$v_t = \gamma v_{t-1} + \alpha \nabla_w J(w - \gamma v_{t-1}) \tag{10}$$

$$w = w - v_t \tag{11}$$

Adagrad As it is presented in [3], the core of the algorithm is to adapt the learning rate with respect to the parameters: bigger updates are done for less frequent parameters and smaller updates are done for the more frequent ones. This principle makes the algorithm be used for sparse data sets.

Adagrad uses different learning rates for each parameter w_i at each time moment t. The general learning rate α is modified at each time moment t for each parameter w_i, with respect to the previous gradients that were computed for the parameter w_i.

$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_t + \epsilon}} * \nabla_w J(w_{t,i}), \tag{12}$$

G_t being a diagonal matrix where each element (i, i) on the first diagonal is the sum of the squared gradients at the time moment t, and ϵ is a term that prevents zero division (it is normally equal to 10^{-8}).

Adadelta As explained in [3], this method was developed with the purpose of eliminating the downside of Adagrad, produced by the decrease of the learning

rate. This way, the algorithm does not store the past squared gradients, but it restricts the window of the past gradients to a fixed dimension s.

s past squared gradients are not stored as it would decrease the efficiency of the algorithm. As an alternative, the sum of the gradients is recursively defined as a decaying average of the past squared gradients (recent results become more important than the results obtained in the past, meaning that their progress becomes more important). This way, the current mean $E[g2]_t$ at time moment t depends only on the past mean $(t-1)$ and the current gradient (t) (fraction γ, usually considered close to 0.9)

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{13}$$

$$\nabla w_t = \frac{\alpha}{E[g^2]_t + \epsilon} g_t \tag{14}$$

The authors of the algorithm have stated that the updates of the terms (g_t and w_t) do not match, meaning that the updates of the terms g_t have to have the same significance like the parameters w_t. In order to do so, they have defined an exponentially decaying average of the squared updates of parameters w_t.

$$E[w^2]_t = \gamma E[w^2]_{t-1} + (1-\gamma) w_t^2 \tag{15}$$

The RMS of the parameters' updates becomes:

$$RMS[w^2]_t = \sqrt{E[w^2]_{t-1} + \epsilon} \tag{16}$$

RMS is approximated with the updates of the parameters until the previous time moment because $RMS(\nabla w)_t$ is not known. After the learning rate α is replaced with $RMS(\nabla w)_{t-1}$, the iterative Adadelta relation is:

$$\nabla w_t = \frac{RMS[\nabla w_{t-1}]}{RMS[g_t]} * g_t \tag{17}$$

$$w_{t+1} = w_t + \nabla w_t \tag{18}$$

Moreover, it is not needed to set the default value for the learning rate at Adadelta, since it is discarded from the iterative relation.

RMSprop In [3], RMSprop is defined as a gradient descent method with adaptive learning rate. RMSprop and Adadelta were individually developed, both with the purpose of eliminating the downside of the decreasing learning rate of Adadelta. The iterative mathematical expression is:

$$E[g^2]_t = 0.9 E[g^2]_{t-1} + 0.1 g_t^2 \tag{19}$$

$$w_{t+1} = w_t + \frac{\nabla}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{20}$$

The authors suggest that γ should be 0.9, and α should be 0.001.

Adam – Adaptive Moment Estimation According to [3], Adam is another method that computes adaptive learning rates for each parameter. Similarly to Adadelta, an exponentially decaying average of the past squared gradients v_t is stored, but an exponentially decaying average of the past gradients

m_t is also stored, with the purpose of increasing the speed of the convergence by decreasing the amplitudes of the oscillations among local minima.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{21}$$

$$v_t = \beta_2 m_{t-1} + (1 - \beta_2) g_t^2 \tag{22}$$

m_t and v_t are estimators of the first momentum (mean), respectively estimator of the second momentum (uncentered variance) of the gradients. The authors have noticed that, when initializing m_t and v_t with null vectors, the estimators are biased towards 0, especially at the initial time moments or β_1 and β_2 when are closed to 1.

in order to eliminate the biases, the unbiased estimators are used. Further, the unbiased estimators are used and the iterative mathematical expression becomes

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\tilde{v}_t + \epsilon}} \tilde{m}_t \tag{23}$$

The authors of the method suggest the values 0.9 for β_1, 0.999 for β_2 and 10-8 for ϵ (ϵ is a term that prevents zero division).

In conclusion, unlike Adadelta or RMSprop, Adam has the advantage that it makes a bias correction.

AdaMax In [3], AdaMax is presented as an extension of Adam. The update of the factor v_t from Adam can be rewritten using the Euclidian norm (or l_2 norm) for the past gradients.The relation can also be generalized using the p-norm (l_p norm). But for big values of p norm, the norms become numerically unstable. It is known that l_1, l_2 and l_∞ norms have a stable behaviour. With this taken into consideration, it was proposed AdaMax, which uses l_∞ norm and shows that, by using this norm, v_t converges to a stable value. The factor is defined, which has the significance of the infinite norm of v_t.

$$u_t = \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty) |g_t|^\infty = max(\beta_2 v_{t-1}, |g_t|) \tag{24}$$

where $|g_t|^\infty$ is the infinite norm of g_t, and $|g_t|$ is the absolute values vector of the gradient components at timestep t.

The iterative equation for AdaMax is:

$$w_{t+1} = w_t - \frac{\alpha}{u_t} \tilde{m}_t \tag{25}$$

The authors of the method suggest the values 0.9 for β_1, 0.999 for β_2 and 0.002 for α.

Summarizing, one can say that AdaMax is an alternative of Adam that uses the infinite norm.

Nadam – Nesterov-accelerated Adaptive Moment Estimation As it is mentioned in [3], it combines the algorithms Adam and NAG (Nesterov Accelerated Gradient).

Therefore, the update equation becomes:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\tilde{v}_t + \epsilon}} (\beta_1 \tilde{m}_t + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t}) \tag{26}$$

References

[1] Nugues P. *Artificial Intelligence EDAP01*. Lund University, 2022. Chap. Machine Learning.

[2] Norvig P. Russel S. *Artificial Intelligence - A Modern Approach*. Pearson Education, 2021. Chap. 19. Learning from Examples. ISBN: 9780134610993.

[3] Ruder S. *An overview of gradient descent optimization algorithms*. 2017. URL: `https://arxiv.org/pdf/1609.04747.pdf`. (accessed: 13.02.2022).