Homework 5

10-605/10-805: Machine Learning with Large Datasets

Due Wednesday, November 15th at 11:59 PM Eastern Time

Instructions: Submit your solutions via Gradescope, *with your solution to each subproblem on a separate page*, i.e., following the template below.

Note that Homework 5 consists of two parts: this written assignment, and a programming assignment. Remember to fill out the collaboration section found at the end of this homework as per the course policy.

All students are required to complete **all** sections of the homework. Your homework score will be calculated as a percentage over the maximum points you can earn, which is 100. The grading breakdown is as follows:

1. Written Section *[60 points]*
2. Programming Section *[40 points]*

Programming: The programming in this homework **is** autograded so make sure that you upload your notebooks to the Programming submission slot and do not add any additional code boxes.

1 Written Section [60 Points]

1.1 Simple Neural Network (20 pts)

Consider a feedforward neural network, defined by the following composition of functions:

$$\mathbf{q} = W^{(1)}x + \mathbf{b^{(1)}} \tag{1}$$
$$\mathbf{h} = ReLU(\mathbf{q}) = \max(\mathbf{q}, 0) \tag{2}$$
$$\mathbf{p} = W^{(2)}\mathbf{h} + \mathbf{b^{(2)}} \tag{3}$$
$$\mathbf{L}(\mathbf{y}, \mathbf{p}) = (\mathbf{p} - \mathbf{y})^T(\mathbf{p} - \mathbf{y}) \tag{4}$$

Here $x \in \mathbb{R}^D$ is an input observation/data point, $W^{(1)} \in \mathbb{R}^{H^{(1)} \times D}$ is a set of weights corresponding to the first (input) layer, $\mathbf{b^{(1)}} \in \mathbb{R}^{H^{(1)}}$ is an offset term corresponding to the first layer, $W^{(2)} \in \mathbb{R}^{H^{(2)} \times H^{(1)}}$ is a set of weights corresponding to the second layer, and $\mathbf{b}^{(2)} \in \mathbb{R}^{H^{(2)}}$ is an offset term corresponding to the second layer. Note also that the max in step 2 is applied element-wise.

Our goal is to find the gradient of the loss, L, with respect to the weights W and b using backpropogation. We will explore backpropagation on the simple neural network above. Note that you may need the Kronecker delta:

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

(a) *[5 points]* First derive the following five terms:

$$\frac{\partial L}{\partial p_j}, \frac{\partial p_i}{\partial W^{(2)}_{j,k}}, \frac{\partial L}{\partial W^{(2)}_{j,k}}, \frac{\partial L}{\partial b^{(2)}_j}, \frac{\partial p_i}{\partial h_j}$$

$$\begin{aligned} \frac{\partial L}{\partial p_j} &= \frac{\partial}{\partial p_j}((\mathbf{p} - \mathbf{y})^T(\mathbf{p} - \mathbf{y})) \\ &= \frac{\partial}{\partial p_j}\sum_i (p_i - y_i)^2 \\ &= \sum_i \frac{\partial}{\partial p_j}(p_i - y_i)^2 \\ &= \sum_i \delta_{ij} 2(p_i - y_i) \\ &= 2(p_j - y_j) \end{aligned}$$

$$
\begin{aligned}
\frac{\partial p_i}{\partial W^{(2)}_{j,k}} &= \frac{\partial}{\partial W^{(2)}_{j,k}} W^{(2)}_i \mathbf{h} + \mathbf{b^{(2)}} \\
&= \frac{\partial}{\partial W^{(2)}_{j,k}} (\mathbf{b^{(2)}_i} + \sum_l W^{(2)}_{il} \mathbf{h}_l) \\
&= \frac{\partial}{\partial W^{(2)}_{j,k}} (\sum_l W^{(2)}_{il} \mathbf{h}_l) \\
&= \delta_{ij} \delta_{lk} (\sum_l \mathbf{h}_l) \\
&= \delta_{ij} h_k
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial W^{(2)}_{j,k}} &= \frac{\partial L}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial W^{(2)}_{j,k}} \\
&= \sum_i \frac{\partial L}{\partial p_i} \frac{\partial p_i}{\partial W^{(2)}_{j,k}} \\
&= \sum_i 2(p_i - y_i) \delta_{ij} h_k \\
&= 2(p_j - y_j) h_k
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial b^{(2)}_j} &= \frac{\partial L}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial b^{(2)}_j} \\
&= \sum_i \frac{\partial L}{\partial p_i} \frac{\partial p_i}{\partial b^{(2)}_j} \\
&= \sum_i \delta_{ij} 2(p_i - y_i) \\
&= 2(p_j - y_j)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial p_i}{\partial h_j} &= \frac{\partial}{\partial h_j} (W^{(2)}_i \mathbf{h}_i + \mathbf{b^{(2)}}_i) \\
&= \frac{\partial}{\partial h_j} (\mathbf{b^{(2)}}_i + \sum_k W^{(2)}_{ik} \mathbf{h}_k) \\
&= \frac{\partial}{\partial h_j} (\sum_k W^{(2)}_{ik} \mathbf{h}_k) \\
&= \delta_{jk} (\sum_k W^{(2)}_{ik}) \\
&= W^{(2)}_{ij}
\end{aligned}
$$

(b) *[3 points]* Now derive the following three terms:

$$\frac{\partial h_j}{\partial q_i}, \frac{\partial L}{\partial h_i}, \frac{\partial L}{\partial q_i}$$

$$\begin{aligned}
\frac{\partial h_j}{\partial q_i} &= \frac{\partial}{\partial q_i}(ReLU(q_j)) \\
&= \frac{\partial}{\partial q_i}(\max(\mathbf{q_j}, 0)) \\
&= \delta_{ij}\frac{\max(q_j, 0)}{q_j}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial L}{\partial h_i} &= \frac{\partial L}{\partial \mathbf{p}}\frac{\partial \mathbf{p}}{\partial h_i} \\
&= \sum_j 2(p_j - y_j)W^{(2)}_{ji} \\
&= 2(\mathbf{p} - \mathbf{y})^T W^{(2)}_{:,i}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial L}{\partial q_i} &= \frac{\partial L}{\partial \mathbf{h}}\frac{\partial \mathbf{h}}{\partial q_i} \\
&= \sum_k \sum_j 2(p_k - y_k)W^{(2)}_{kj}\frac{\partial h_j}{\partial q_i} \\
&= \sum_k \sum_j 2(p_k - y_k)W^{(2)}_{kj}\delta_{ij}\frac{\max(q_j, 0)}{q_j} \\
&= \sum_k 2(p_k - y_k)W^{(2)}_{ki}\frac{\max(q_i, 0)}{q_i} \\
&= 2(\mathbf{p} - \mathbf{y})^T W^{(2)}_{:,i}\frac{\max(q_i, 0)}{q_i}
\end{aligned}$$

(c) *[4 points]* Finally, derive the following four terms, which completes what we set out to calculate:

$$\frac{\partial q_i}{\partial W^{(1)}_{j,k}}, \frac{\partial q_i}{\partial b^{(1)}_j}, \frac{\partial L}{\partial W^{(1)}_{j,k}}, \frac{\partial L}{\partial b^{(1)}_i}$$

$$\begin{aligned}
\frac{\partial q_i}{\partial W^{(1)}_{j,k}} &= \frac{\partial}{\partial W^{(1)}_{j,k}}(W^{(1)}_i x + \mathbf{b_i^{(1)}}) \\
&= \frac{\partial}{\partial W^{(1)}_{j,k}}(\mathbf{b_i^{(1)}} + \sum_l W^{(1)}_{il} x_l) \\
&= \frac{\partial}{\partial W^{(1)}_{j,k}}(\sum_l W^{(1)}_{il} x_l) \\
&= \delta_{ij} \sum_l \delta_{kl} x_l \\
&= \delta_{ij} x_k
\end{aligned}$$

$$\begin{aligned}
\frac{\partial q_i}{\partial b^{(1)}_j} &= \frac{\partial}{\partial b^{(1)}_j}(W^{(1)}_i x + \mathbf{b}^{(1)}_i) \\
&= \frac{\partial}{\partial b^{(1)}_j}(\mathbf{b}^{(1)}_i) \\
&= \delta_{ij}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial L}{\partial W^{(1)}_{j,k}} &= \frac{\partial L}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial W^{(1)}_{j,k}} \\
&= \sum_i 2(\mathbf{p} - \mathbf{y})^T W^{(2)}_{:,i} \frac{\max(q_i, 0)}{q_i} \frac{\partial q_i}{\partial W^{(1)}_{j,k}} \\
&= \sum_i 2(\mathbf{p} - \mathbf{y})^T W^{(2)}_{:,i} \frac{\max(q_i, 0)}{q_i} \delta_{ij} x_k \\
&= 2(\mathbf{p} - \mathbf{y})^T \sum_i W^{(2)}_{:,i} \frac{\max(q_i, 0)}{q_i} \delta_{ij} x_k \\
&= 2(\mathbf{p} - \mathbf{y})^T W^{(2)}_{:,j} \frac{\max(q_j, 0)}{q_j} x_k
\end{aligned}$$

$$
\begin{aligned}
\frac{\partial L}{\partial b_i^{(1)}} &= \frac{\partial L}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial b_i^{(1)}} \\
&= \sum_j 2(\mathbf{p} - \mathbf{y})^T W_{:,j}^{(2)} \frac{\max(q_j, 0)}{q_j} \frac{\partial q_j}{\partial b_i^{(1)}} \\
&= \sum_j 2(\mathbf{p} - \mathbf{y})^T W_{:,j}^{(2)} \frac{\max(q_j, 0)}{q_j} \delta_{ij} \\
&= 2(\mathbf{p} - \mathbf{y})^T W_{:,i}^{(2)} \frac{\max(q_i, 0)}{q_i}
\end{aligned}
$$

(d) *[4 points]* Describe how backpropagation can efficiently compute these gradients, using the expressions you derived in part (c) for $\frac{\partial L}{\partial W^{(1)}_{j,k}}$ and $\frac{\partial L}{\partial b^{(1)}_i}$ as supporting evidence.

Backpropagation can efficiently calculate these derivatives for multiple reasons. First of all, the partial derivatives with respect to $W^{(1)}_{j,k}$ only need to be calculated if $q_j > 0$. And for $b^{(1)}_i$ the partial derivatives only have to be calculated if $q_i > 0$. Secondly, the partials with respect to $W^{(1)}_{j,k}$ and $b^{(1)}_i$ share most of their terms. The only difference between the two is that the derivative with respect to $W^{(1)}_{j,k}$ has an extra x_k term.

$$\frac{\partial L}{\partial b^{(1)}_i} = 2(\mathbf{p} - \mathbf{y})^T W^{(2)}_{:,i} \frac{\max(q_i, 0)}{q_i}$$

$$\frac{\partial L}{\partial W^{(1)}_{i,j}} = 2(\mathbf{p} - \mathbf{y})^T W^{(2)}_{:,i} \frac{\max(q_i, 0)}{q_i} x_j$$

Thus the shared terms: $2(\mathbf{p} - \mathbf{y})^T W^{(2)}_{:,i}$ only need to be calculated once. Then this expression can be used to evaluate both partial derivatives.

(e) *[4 points]* How do your experiences from (a)-(d) help motivate the benefits of DL frameworks like TensorFlow or PyTorch. Note: this can be a 1-2 sentence answer.

Many of the calculations for calculating gradients are repeated and can be easily done in parallel. So having DL frameworks helps by preventing unnecessary repeat calculations and helping to parallelize calculations.

1.2 Curse of Dimensionality (20 pts)

We define the ϵ-cover of a search space S as a subset $E \subset S$:

$$E = \{x \in \mathbb{R}^n \mid \forall y \in S, \exists x \ s.t. \ \|y - x\|_\infty \leq \epsilon\}$$

In words, every coordinate of every point in our search space S is within ϵ of a point in our set E. The ϵ-covering number is the size of the smallest such set that provides an ϵ-cover of S.

$$N(\epsilon, S) = \min_E\{|E| : E \text{ is an } \epsilon\text{-cover of } S\}$$

Let's work through a concrete example to make the concept clearer: let's say we want to tune the learning rate for gradient descent on a logistic regression model by considering learning rates in the range of $[1, 2]$. We want to have $\epsilon = .05$-coverage of our search space $S = [1, 2]$. Then $E = \{1.05, 1.15, 1.25, 1.35, 1.45, 1.55, 1.65, 1.75, 1.85, 1.95\}$. We see that any point in $[1, 2]$ is within 0.05 of a point in E. So E provides ϵ-coverage of S. The ϵ-covering number is 10. The questions below will ask you to generalize this idea.

(a) *[6 points]* Find the size of a set needed to provide ϵ-coverage (ϵ-covering number) of $S = [0, 1] \times [0, 2]$, the Cartesian product of two intervals. Note that we want this to hold for a generic ϵ.

$$\lceil \frac{1}{2\epsilon} \rceil \lceil \frac{2}{2\epsilon} \rceil$$
$$= \lceil \frac{1}{2\epsilon} \rceil \lceil \frac{1}{\epsilon} \rceil$$

where $\lceil x \rceil$ means to take the ceiling of x.

(b) *[6 points]* Find the ϵ-covering number of $S = [a_1, b_1] \times \ldots \times [a_d, b_d]$, the Cartesian product of d arbitrary closed intervals. Specifically, write down an expression $N(\epsilon, S)$ as a function of a_i, b_i, and ϵ.

$$N(\epsilon, S) = \prod_{i=1}^{d} \left\lceil \frac{b_i - a_i}{2\epsilon} \right\rceil$$

(c) *[8 points]* With respect to your answer in 1.2 (b), intuitively, how is the covering number related to the volume of S? Moreover, on what order does the covering number grow with respect to the dimension d? What does this say about the volume of S as a function of dimensionality?

The covering number is proportional to the volume of S. The covering number grows exponentially with respect to the number of dimensions d. Every time d increases, the cover number grows by a factor proportional to the size of the new dimension. This means the volume of S grows exponentially with the number of dimensions d.

1.3 Grid versus Random Search (20 pts)

There are two basic strategies commonly employed in hyperparameter search: grid search and random search. Grid search is defined as choosing an independent set of values to try for each hyperparameter and the configurations are the Cartesian product of these sets. Random search chooses a random value for each hyperparameter at each configuration. Imagine we are in the (extreme) case where we have d hyperparameters all in the range $[0, 1]$, but only *one* (h_1) of them has any impact on the model, while the rest have *no* effect on model performance. For simplicity, assume that for a fixed value of h_1 that our training procedure will return the exact same trained model regardless of the values of the other hyperparameters.

(a) *[8 points]* Imagine that we consider q hyperparameter configurations, which are enough to provide ϵ-coverage for grid search. How many distinct models will this training procedure produce? What is this as a fraction of the total number of configurations?

$$\text{Number of Distinct Models} = \left\lceil \frac{1}{2\epsilon} \right\rceil$$

$$\begin{aligned} \text{Number of Models} = q &= \prod_{i=1}^{d} \left\lceil \frac{1}{2\epsilon} \right\rceil \\ &= \left\lceil \frac{1}{2\epsilon} \right\rceil^{d} \end{aligned}$$

$$\begin{aligned} \frac{\text{Number of Distinct Models}}{\text{Number of Models}} &= \left\lceil \frac{1}{2\epsilon} \right\rceil \Big/ \left\lceil \frac{1}{2\epsilon} \right\rceil^{d} \\ &= \left\lceil \frac{1}{2\epsilon} \right\rceil \left\lceil \frac{1}{2\epsilon} \right\rceil^{-d} \\ &= \left\lceil \frac{1}{2\epsilon} \right\rceil^{-d+1} \end{aligned}$$

(b) *[5 points]* Alternatively, if we consider q random configurations as part of random search, then what percent of the configurations will cause a change to the model? Why?

In a random search $(1 - \frac{1}{n})q$ of the random configurations will cause a change to the model. Where n is the number of values we allow h_1 to take on.

There is an infinite number of values between $[0, 1]$, so $(1 - \frac{1}{n})$ evaluates to 1. Since computers can't actually express all real-valued numbers between 0 and 1. The chance of having the model change is very close to 1, but not exactly 1.

Hence nearly 100% of the configurations will cause a change to the model.

(c) *[7 points]* Now let's change our point of view: instead of desiring ϵ-coverage, we have a fixed budget of $B \ll |E|$ configurations to try (where $|E|$ is an epsilon cover of S). Which search strategy—grid or random search, should we employ? Why?

Random search is better because the percentage of models that are different from the initial model is higher for random search than the percentage of models that are different for grid search. If we use grid search there is a high chance that none of the models we test are different from the initial model if the number of dimensions d is large. The chance of random search creating a different model does not depend on d. Instead it depends on the number of unique values we let h_1 take on. Since h_1 is a real-valued number in the interval $[0, 1]$, the chance of generating configurations that are different from the original is essentially 100%.

2 Programming Section [40 Points]

2.1 Introduction

The goal of this assignment is to gain familiarity with deep learning in TensorFlow. There are two parts:

In **Part 1**, you will implement *neural style transfer* in TensorFlow. Neural style transfer will involve building a system that can take in two images (one content image and one style image), and output a third image whose content is closest to the content of the content image while the style matches the style of the style image.

In **Part 2**, you will implement several of the optimization methods discussed in lecture including Gradient Descent, SGD, AdaGrad, and Adam, and train a linear model with them using Tensorflow 2.11. [*Note: you can't use tf.compat.v1.train.Optimizer or tf.keras.optimizers in the final evaluation.*]

2.2 Logistics

We provide the code template for this assignment in *two* Jupyter notebooks. Follow the instructions in the notebooks and implement the missing parts marked with '`<FILL_IN>`' or '`# YOUR CODE HERE`'. Most of the '`<FILL_IN>/YOUR CODE HERE`' sections can be implemented in just a few lines of code.

2.3 Getting lab files

You can obtain the notebooks `hw5_part1.ipynb` and `hw5_part2.ipynb` after downloading and unzipping the `Source code (zip)` from the course website.

To run these notebooks, you can upload the notebooks to your Google drive and open them with Google Colaboratory (Colab).

2.4 Preparing for submission

We provide several public tests via `assert` in the notebook. You may want to pass all those tests before submitting your homework. You can individually submit a notebook for debugging but **make sure to submit the notebook for your final submission to receive full credit.**

In order to enable auto-grading, please do not change any function signatures (e.g., function name, parameters, etc) or delete any cells. If you do delete any of the provided cells (even if you re-add them), the autograder will fail to grade your homework. If you do this, you will need to re-download the homework files and fill in your answers again and resubmit.

2.5 Submission

1. Download the notebooks from Colab to your local computer by going to `File -> Download .ipynb` and submit them to the corresponding Gradescope grader.

2. Submit the completed notebooks via Gradescope.

2.6 Part A: Neural Style Transfer in Tensorflow

In this part you will implement style transfer based on the paper: "A Neural Algorithm of Artistic Style": https://arxiv.org/pdf/1508.06576.pdf.

Basically, we will build a system that can take in two images (one input content image and one input style image), and output another image whose content is closest to the content of the content image while style is closest to the style of the style image.

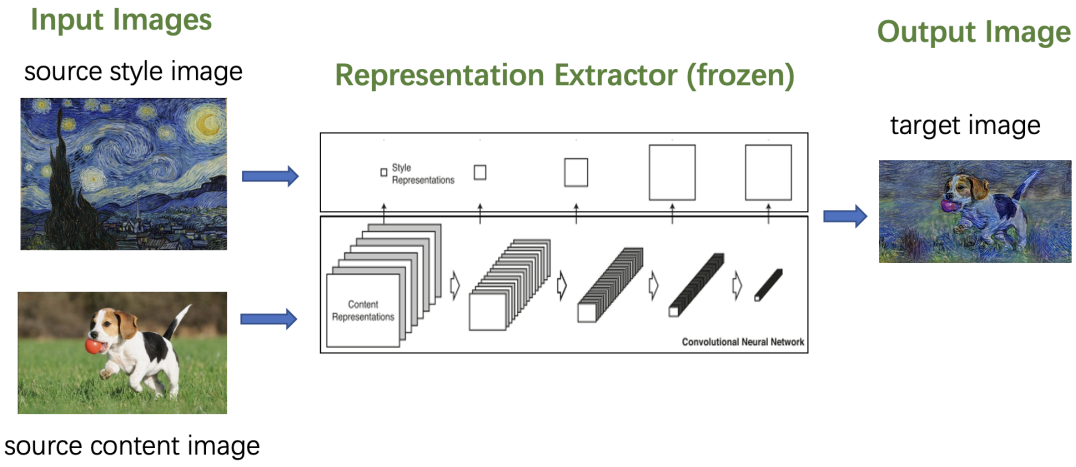

Figure 1: Overview of style transfer.

As depicted in Fig. 1, neural style transfer is able to generate an image whose content is close to the content image (a dog running on the grass with a ball in its mouth), but painted in the style of the style image (Van Gogh's "The Starry Night"). As the name suggests, it transfers the "style" of one image to another one. With this technique, we can obtain for an arbitrary image, its equivalent in different artwork styles. Towards this end, we consider two aspects of an image's representation, content representation and style representation. We make the assumption that we could have a pre-trained image encoder as a representation extractor that is able to disentangle content information and style information of an image. We also assume some intermediate layers capture content representation and some capture style representation. And the target image is expected to be close to the source content image in terms of the content representation, and close to the source style image in terms of the style representation.

Specifically, we utilize a pre-trained VGG model and keep its parameters frozen. We define the layers of interest within the model: content layers and style layers. For a given image, we refer to the output features of the content layers within the model with this image as the input, as the content representations of the image (w.r.t. the model). Like-wise, we define for an image its style representation (w.r.t. the model) as the gram matrix of the output features of the style layers within the model with this image as the input. For example, for a 5-layer perceptron, if we select the content layers to be the second and fourth layer, then the content representation of an image would be the output features from the second and fourth layer. The training objective is to find an output image that could minimize the combination of content loss and style loss, where content loss is the distance between the content representation of the input content image and the output image, and style loss is the distance between the style representation of the input style image and the output image. The only trainable tensor is the output image, which could be optimized via gradient descent.

There are 5 parts of the assignment:

2.6.1 Visualize data

The first part is written for you.

2.6.2 Prepare the data

The second part has the following functions:

1. `load_and_process_img()`: Load and process the image at the given path.
2. `deprocess_img()`: Perform inverse processing on the input image

2.6.3 Creating the model

The third part is to create the model:

1. Define content and style representations (we need to define the content layers and style layers)
2. `model_VGG()`: Builds the model (Use `tf.keras.applications.vgg19.VGG19` to get `style_outputs` and `content_outputs`)

2.6.4 Loss functions

The fourth part is to define and create the loss functions:

1. `compute_content_loss()`
   Use `tf.reduce_mean` and implement the formula: $L^l_{\text{content}}(p, x) = \sum_{i,j}(F^l_{ij}(x) - P^l_{ij}(p))^2$
2. `gram_matrix()`
   The gram matrix, G^l_{ij}, is the inner product between the vectorized feature map i and j in layer l. Remember to divide the `gram_matrix` by `input_tensor.shape[0]`
3. `compute_style_loss()`
   We need to implement the formula: $E_l = \sum_{i,j}(G^l_{ij} - A^l_{ij})^2$

Now that we have defined the loss functions, we need to develop the following functions:

1. `compute_features()`
   This function has following arguments:
   `model`: The model that we are using.
   `content_path`: The path to the content image.
   `style_path`: The path to the style image.
   Returns: The style features and the content features.
2. `calculate_loss()`
   Returns: `total_loss`, `style_loss`, `content_loss`
   We will give more emphasis to deep layers. For example, the weights for `block1conv1` can be 1, for `block2conv1` it can be 2, and so on `weight_per_style_layer = [1.0,2.0,3.0,4.0,5.0]`

2.6.5 Optimization loop

The final part is to optimize the loop so that we can get the best stylized image.

2.7 Part B: Optimization Methods

In this portion of the homework you will implement several common optimizers. You will first build a simple linear regression model and train it by calling existing optimizers provided by tf.keras. Then you will implement and evaluate your own optimizers. Finally you will have a better understanding of the mechanism of different optimizers and their effectiveness on simple models. Detailed instructions are included in the assignment notebook.

There are 4 optimizers you need to implement:

- Gradient Descent: update weights using all samples to reach the minimum of the loss function.
- Stochastic Gradient Descent: update weights using one sample at a time to reach the minimum of the loss function.
- AdaGrad: decrease the step size for coordinates with high gradient magnitudes with a cheap approximation of the hessian.
- Adam: combine momentum (move further in the correct direction and less in the wrong direction) and RMSprop (take a moving average of the coordinates of the gradient to put moreemphasis on the current gradients).

The instructions in the notebook will guide you to implement each of the optimizers. To understand the theory behind them, we encourage you to read the update rule provided in the notebook, as well as the details provided in on November 1st.

3 Collaboration Questions

1. (a) Did you receive any help whatsoever from anyone in solving this assignment?

   Yes

   (b) If you answered 'yes', give full details (e.g. "Jane Doe explained to me what is asked in Question 3.4")

   Gabriel Fonseca helped with the approach towards calculating gradients.

2. (a) Did you give any help whatsoever to anyone in solving this assignment?
   No

   (b) If you answered 'yes', give full details (e.g. "I pointed Joe Smith to section 2.3 since he didn't know how to proceed with Question 2")

3. (a) Did you find or come across code that implements any part of this assignment?
   No

   (b) If you answered 'yes', give full details (book & page, URL & location within the page, etc.).