MACHINE LEARNING 1 COURSEWORK

General instructions

Set: Monday 17th April, 9:00am

Due: Friday 28th April, 19:00

Estimated time required: 15 hours, assuming you have studied all of the problem sheets.

Submission: Please write all your Python *codes* in one Jupyter notebook **in the order of questions**. Save the notebook with a name that clearly identifies you as the author, such as `sm007.ipynb` or `smith.ipynb`. Please scan or typeset your *written reports* in Q4, Q6 and Q7 such that you can produce a single **PDF file**. Give it a personalised name, such as `smith.pdf`. No marks will be lost if the reports are not typeset, provided they are legible.

Upload both the `.ipynb` and the `.pdf` files on Moodle at the Coursework Submission point by the due deadline.

Conditions: This is an individual coursework, so you should **not** discuss it with other students. As usual for courseworks, the marking will be **not** anonymous. Please use your name (not the candidate number) in the files.

Value: This coursework carries 25% of the total marks for the unit.

Length: There is no minimum or maximum length for this assignment. Indicative guidance on approximate lengths for reports is provided in section 2. In marking emphasis will be placed upon correct, well-structured and commented code, and correct, clear and precise argument in the written work.

Support and advice: You can ask **general** questions to the lecturer on the Padlet forum, and to the tutors during the labs. For example: you may ask "what `ValueError` means in Python in general". You may **not** ask "what should I do to *my* code to make it go away".

Feedback: You will receive feedback within a maximum of three weeks following the submission deadline. The feedback will consist of your *provisional* coursework mark and written comments pointing out mistakes and what could be done better.

Late submission of coursework: If there are valid circumstances preventing you from meeting the deadline, your Director of Studies may grant you an extension to the specified submission date, if it is requested before the deadline. Forms to request an extension are available on SAMIS.

- If you submit a piece of work after the submission deadline but within five working days, and no extension has been granted, the maximum mark possible will be the pass mark.
- If you submit work more than five working days after the submission deadline, you will normally receive a mark of 0 (zero), unless you have been granted an extension.

Academic integrity statement: Academic misconduct is defined by the University as "the use of unfair means in any examination or assessment procedure". This includes (but is not limited to) cheating, collusion, plagiarism, fabrication, or falsification. The University's Quality Assurance Code of Practice, QA53 Examination and Assessment Offences, sets out the consequences of committing an offence and the penalties that might be applied.

1. Background

The actual coursework questions are listed in Section 2. However, they refer to algorithms and equations set in this mathematical background, so please study it before attempting the questions.

1.1. **Matrix Completion.** Matrix Completion aims to compute an unknown low-rank $n \times n$ matrix $UW^\top$ from a given dataset containing only some of the elements of this matrix, perturbed by noise. This can be achieved by solving a regression problem on the factors $U, W \in \mathbb{R}^{n\times r}$:

$$U_*, W_* = \arg\min_{U,W\in\mathbb{R}^{n\times r}} L_{\mathcal{I},Y}(U,W), \tag{1}$$

$$L_{\mathcal{I},Y}(U,W) = \frac{1}{m}\sum_{(i_1,i_2)\in\mathcal{I}} \left(\sum_{j=0}^{r-1} u_{i_1,j}w_{i_2,j} - y_{i_1,i_2}\right)^2 \tag{2}$$

where $u_{i,j}, w_{i,j}$ are elements of matrices U and W, respectively, $\mathcal{I}$ is a known set of m pairs (i_1, i_2) of row and column indices of the matrix which are observed in the given data y_{i_1,i_2}, and $r < n$ is the rank of the sought matrix. Despite some similarity to Principal Component Analysis, Matrix Completion is different in two crucial aspects: we search only *factors* U, W constituting a rank-r matrix $UW^\top$ instead of this entire matrix, and we do so by accessing only $m \le n^2$ elements of the matrix.

1.2. **Selection of the sampling set $\mathcal{I}$.** Traditionally used is the so-called *Bernoulli* distribution, where each index pair $(i_1, i_2) \in \{0, \dots, n-1\}^2$ is sampled with the same probability $p \le 1$ as shown in Algorithm 1. The value of p is also called *undersampling factor*, since Algorithm 1 produces on average $\mathbb{E}[m] = pn^2$ data samples. In Python we can store $\mathcal{I}$ as a $m \times 2$ matrix, $\mathcal{I} \in \mathbb{N}_0^{m\times 2}$, where $\mathbb{N}_0 = \{0\} \cup \mathbb{N}$.

Algorithm 1 Bernoulli sampling (**will be provided**)

Require: Matrix size $n \in \mathbb{N}$, probability value $p \in (0, 1]$.
1: Initialise $\mathcal{I} = \emptyset$.
2: **for** $i_1 = 0, \dots, n-1$ **do**
3: **for** $i_2 = 0, \dots, n-1$ **do**
4: Sample a random number $\xi \sim \mathcal{U}(0,1)$ uniformly distributed on $(0, 1)$.
5: **if** $\xi < p$ **then**
6: Append the current index pair to the set, $\mathcal{I} = \mathcal{I} \cup (i_1, i_2)$.
7: **end if**
8: **end for**
9: **end for**
10: **return** Index sampling set $\mathcal{I}$.

1.3. **Gradient of the matrix completion loss.** We will use both Gradient Descent and Stochastic Average Gradient algorithms. So let us first differentiate the pointwise loss

$$\ell_{i_1,i_2}(U,W) := \left(\sum_{j=0}^{r-1} u_{i_1,j}w_{i_2,j} - y_{i_1,i_2}\right)^2.$$

The full calculation is shown in Appendix A. In matrix form, we can write

$$V^u := \frac{\partial \ell_{i_1,i_2}(U,W)}{\partial U} = 2(\mathbf{u}_{i_1}\mathbf{w}_{i_2}^\top - y_{i_1,i_2}) \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \\ w_{i_2,1} & \cdots & w_{i_2,r} \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{bmatrix}, \tag{3}$$

where the nonzero row appears in the i_1-th position, and $\mathbf{u}_{i_1}, \mathbf{w}_{i_2} \in \mathbb{R}^{1\times r}$ are i_1-th and i_2-th rows of U and W, respectively. Similarly,

$$V^w := \frac{\partial \ell_{i_1,i_2}(U,W)}{\partial W} = 2(\mathbf{u}_{i_1}\mathbf{w}_{i_2}^\top - y_{i_1,i_2}) \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \\ u_{i_1,1} & \cdots & u_{i_1,r} \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{bmatrix}, \tag{4}$$

where the nonzero row appears in the i_2-th position.

Note that we can accelerate computations significantly by not assembling zeros in V^u and V^w. Firstly, compute only the i_1-th row of V^u and i_2-th row of V^w,

$$\mathbf{v}^u = 2(\mathbf{u}_{i_1}\mathbf{w}_{i_2}^\top - y_{i_1,i_2})\mathbf{w}_{i_2}, \tag{5}$$

$$\mathbf{v}^w = 2(\mathbf{u}_{i_1}\mathbf{w}_{i_2}^\top - y_{i_1,i_2})\mathbf{u}_{i_1}. \tag{6}$$

Stochastic Average Gradient can use the row vectors $\mathbf{v}^u$ and $\mathbf{v}^w$ directly. To implement the full gradient (for the Gradient Descent method), we need to sum the pointwise loss gradients (3), (4) over all data (pairs in $\mathcal{I}$). However, we can still add only nonzero rows in each step, as shown in Algorithm 2.

Algorithm 2 Full gradient of (2).

Require: Matrices $U, W \in \mathbb{R}^{n\times r}$, data matrix $Y \in \mathbb{R}^{n\times n}$, index set $\mathcal{I} \in \mathbb{N}_0^{m\times 2}$.

Initialise $G^u, G^w = 0 \in \mathbb{R}^{n\times r}$.
for $(i_1, i_2) \in \mathcal{I}$ **do**
  Compute $\mathbf{v}^u$ and $\mathbf{v}^w$ as shown in (5), (6).
  Increment i_1-th row of G^u: $\mathbf{g}^u_{i_1} = \mathbf{g}^u_{i_1} + \mathbf{v}^u$
  Increment i_2-th row of G^w: $\mathbf{g}^w_{i_2} = \mathbf{g}^w_{i_2} + \mathbf{v}^w$
end for
$G^u = G^u/m$, $G^w = G^w/m$.
return $G^u = \nabla_U L_{\mathcal{I},Y}(U,W)$ and $G^w = \nabla_W L_{\mathcal{I},Y}(U,W)$.

1.4. **Test dataset and stopping criterion.** For simplicity (and debugging), the full matrix $Y \in \mathbb{R}^{n\times n}$ will be provided. This allows us to stop the algorithm using a simple training-test data splitting:

- Using Algorithm 1, sample an extended index set $\mathcal{I}_{ext} \in \mathbb{N}_0^{N\times 2}$.
- Let the test set $\mathcal{I}_{test}$ be μ ($1 \ll \mu \ll N$) index pairs from $\mathcal{I}_{ext}$ selected at random.
- Let the training set $\mathcal{I} = \mathcal{I}_{ext} \backslash \mathcal{I}_{test}$ be the remaining $m = N - \mu$ pairs.

Algorithm 3 Gradient Descent (GD) method for Matrix Completion

Require: Initial guess $U_0, W_0 \in \mathbb{R}^{n\times r}$, data matrix $Y \in \mathbb{R}^{n\times n}$, training index set $\mathcal{I} \in \mathbb{N}_0^{m\times 2}$, test index set $\mathcal{I}_{test} \in \mathbb{N}_0^{\mu\times 2}$, learning rate $t > 0$, stopping tolerance $\varepsilon > 0$, maximal number of iterations $K \in \mathbb{N}$.

Initialise $U = U_0$, $W = W_0$.
for $k = 0, 1, \ldots, K-1$ **do**
  if $L_{\mathcal{I}_{test},Y}(U,W) < \varepsilon$ **then**
    break
  end if
  Compute G^u and G^w using Algorithm 2.
  Update iterates $U = U - tG^u$ and $W = W - tG^w$.
end for
return U, W.

Now in the course of iterations, we compute the loss value on the test dataset, $L_{\mathcal{I}_{test},Y}(U,W)$ as defined in (2), and stop the algorithm when $L_{\mathcal{I}_{test},Y}(U,W)$ becomes smaller than a desired tolerance $\varepsilon > 0$. The Gradient Descent can now be summarised as shown in Algorithm 3.

1.5. **Choice of the initial guess.** Note from (3), (4) that we cannot take $U_0 = W_0 = 0$ as the initial guess: in this case the initial gradient is zero too, and the algorithm cannot progress. Instead, we prepare U_0 and W_0 in the form

$$U_0 = W_0 = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \in \mathbb{R}^{n\times r}.$$

A function to do this will be provided.

1.6. **Accelerated algorithm: stochastic average gradient.** The Stochastic Average Gradient (SAG) method combines the best features of GD and Stochastic Gradient Descent in terms of computing speed, at the expense of more memory consumption. Instead of updating the iterates U and W directly with each pointwise loss gradient, we update the average approximation of the full gradient,

$$\widetilde{G}^u := \frac{1}{m}\sum_{i=0}^{m-1} \widetilde{V}_i^u \approx G^u, \qquad \widetilde{G}^w := \frac{1}{m}\sum_{i=0}^{m-1} \widetilde{V}_i^w \approx G^w,$$

where $\widetilde{V}_i^u, \widetilde{V}_i^w \in \mathbb{R}^{n\times r}$ are pointwise loss gradients computed at some earlier iteration where i was sampled. However, since the pointwise loss gradients are nonzero in only one row (3), (4), we can save memory and store only row vectors $\widetilde{\mathbf{v}}_i^u \in \mathbb{R}^r$ and $\widetilde{\mathbf{v}}_i^w \in \mathbb{R}^r$ which are computed similarly to (5), (6). The corresponding pseudocode of the SAG method is shown in Algorithm 4.

Algorithm 4 Stochastic Average Gradient (SAG) method for Matrix Completion

Require: Initial guess $U_0, W_0 \in \mathbb{R}^{n\times r}$, data matrix $Y \in \mathbb{R}^{n\times n}$, training index set $\mathcal{I} \in \mathbb{N}_0^{m\times 2}$, test index set $\mathcal{I}_{test} \in \mathbb{N}_0^{\mu\times 2}$, learning rate $t > 0$, stopping tolerance $\varepsilon > 0$, maximal number of iterations $K \in \mathbb{N}$.

Initialise $U = U_0$, $W = W_0$, $\widetilde{V}^u, \widetilde{V}^w = 0 \in \mathbb{R}^{m\times r}$, $\widetilde{G}^u, \widetilde{G}^w = 0 \in \mathbb{R}^{n\times r}$
for $k = 0, 1, \ldots, K-1$ **do**
  if $L_{\mathcal{I}_{test},Y}(U,W) < \varepsilon$ **then**
    break
  end if
  Sample $i = 0, \ldots, m-1$ uniformly at random.
  Take (i_1, i_2) to be the ith pair from $\mathcal{I}$.
  Subtract old gradients $\widetilde{\mathbf{g}}_{i_1}^u = \widetilde{\mathbf{g}}_{i_1}^u - \frac{1}{m}\widetilde{\mathbf{v}}_i^u$ and $\widetilde{\mathbf{g}}_{i_2}^w = \widetilde{\mathbf{g}}_{i_2}^w - \frac{1}{m}\widetilde{\mathbf{v}}_i^w$.
  Compute $\widetilde{\mathbf{v}}_i^u = 2(\mathbf{u}_{i_1}\mathbf{w}_{i_2}^\top - y_{i_1,i_2})\mathbf{w}_{i_2}$ and $\widetilde{\mathbf{v}}_i^w = 2(\mathbf{u}_{i_1}\mathbf{w}_{i_2}^\top - y_{i_1,i_2})\mathbf{u}_{i_1}$.
  Add new gradients $\widetilde{\mathbf{g}}_{i_1}^u = \widetilde{\mathbf{g}}_{i_1}^u + \frac{1}{m}\widetilde{\mathbf{v}}_i^u$ and $\widetilde{\mathbf{g}}_{i_2}^w = \widetilde{\mathbf{g}}_{i_2}^w + \frac{1}{m}\widetilde{\mathbf{v}}_i^w$.
  Update iterates $U = U - t\widetilde{G}^u$ and $W = w - t\widetilde{G}^w$.
end for
return U, W.

2. Coursework

Preparation of files.

- **Copy** the skeleton notebook `CW.ipynb` and the data file `CW.npz` from `courses/MA20278/CW` to your `MA20278_Workspace` on the Jupyter server. Currently this notebook contains functions:
  - `syntheticY(n)` which computes a synthetic rank-2 $n \times n$ matrix.
  - `bernoulli(n,p)` which implements Algorithm 1.
  - `Loss(U,W,Y,I)` which computes the loss function (2).
  - `initial(n,r)` which computes U_0 and W_0 as shown in Section 1.5.

  Moreover, the notebook contains unit test functions for the pointwise and full gradients which you may find useful in debugging.
- Rename **your** copy of `CW.ipynb` (in `MA20278_Workspace`) to a name that clearly identifies you as the author, such as `smith.ipynb`. This will be the notebook where all questions should be implemented.

Q1: pointwise loss gradient. Implement a Python function `pointwise_gradient(U,W,Y,i1,i2)` with inputs:

- `U`: left factor matrix $U \in \mathbb{R}^{n\times r}$ as a numpy array of shape `(n, r)`,
- `W`: right factor matrix $W \in \mathbb{R}^{n\times r}$ as a numpy array of shape `(n, r)`,
- `Y`: data matrix $Y \in \mathbb{R}^{n\times n}$ as a numpy array of shape `(n, n)`,
- `i1`: a row index, an integer number from 0 to $n-1$ (inclusive),
- `i2`: a column index, an integer number from 0 to $n-1$ (inclusive),

which computes and returns (5) and (6):

- `vu`: $\mathbf{v}^u = 2(\mathbf{u}_{i_1}\mathbf{w}_{i_2}^\top - y_{i_1,i_2})\mathbf{w}_{i_2}$ as a numpy array of shape `(r,)`.
- `vw`: $\mathbf{v}^w = 2(\mathbf{u}_{i_1}\mathbf{w}_{i_2}^\top - y_{i_1,i_2})\mathbf{u}_{i_1}$ as a numpy array of shape `(r,)`.

[3 points]

Q2: full gradient. Implement Algorithm 2 in a Python function `full_gradient(U,W,Y,I)` with the same inputs as in **Q1** except `I`=$\mathcal{I}$ being a numpy array of shape `(m, 2)`. The function `full_gradient` should compute and return two matrices:

- `Gu` $= G^u$ as a numpy array of shape `(n, r)`.
- `Gw` $= G^w$ as a numpy array of shape `(n, r)`.

Use the function `pointwise_gradient` implemented in **Q1**.

Test that your `pointwise_gradient` and `full_gradient` produce correct matrices by running the test functions provided. If you work on the Jupyter server, you can just call `run_tests()` which will run and check all functions starting with the word "test". If you work elsewhere, you need to run `test_pointwise_gradient_1()`, ... `test_full_gradient_3()` individually. In this case, each test function should complete without printing anything. An error would indicate that the corresponding result is incorrect.

[3 points]

Q3: gradient descent. Implement a Python function `gd(U0, W0, Y, I, Itest, t=1, eps=1e-6, K=100)` with the following inputs:

- `U0`: initial guess of the left factor U as a numpy array of shape `(n, r)`.
- `W0`: initial guess of the right factor W as a numpy arrays of shape `(n, r)`.
- `Y`: data matrix $Y \in \mathbb{R}^{n\times n}$ as a numpy array of shape `(n, n)`.
- `I`: a matrix $\mathcal{I} \in \mathbb{N}_0^{m\times 2}$ of training indices (numpy array of shape `(m, 2)`).
- `Itest`: a matrix $\mathcal{I}_{test} \in \mathbb{N}_0^{\mu\times 2}$ of test indices.
- `t`: learning rate $t > 0$ with the default value 1.
- `eps`: stopping threshold $\varepsilon > 0$ with the default value 10^{-6}.
- `K`: maximal number of iterations $K > 0$ with the default value 100.

The function `gd` should implement Algorithm 3 and return two numpy arrays of shape `(n, r)` each, storing the two factor matrices $U, W \in \mathbb{R}^{n\times r}$ computed in the last iteration. Use the function implemented in **Q2** and the `Loss` function.

Moreover, your code inside the function `gd` should collect the test loss values $L_{\mathcal{I}_{test},Y}(U,W)$ from all iterations carried out until the algorithm has stopped, and plot them in the logarithmic scale as a function of the iteration number k.

[3 points]

Q4: testing GD on full data. Test that the `gd` function works first on a complete noiseless low-rank matrix. For this, use the function `syntheticY` with $n = 32$ to create a known rank-2 matrix Y, and the function `bernoulli` with undersampling factor $p = 1$ to produce $\mathcal{I}_{ext}$ containing all indices $i_1, i_2 = 0, \ldots, n-1$.

Initialise U_0, W_0 via the function `initial(n,2)`, and extract $\mu = 20$ distinct index pairs selected uniformly at random from $\mathcal{I}_{ext}$ into $\mathcal{I}_{test}$. Calculate the remaining training set as $\mathcal{I} = \mathcal{I}_{ext} \backslash \mathcal{I}_{test}$.

Call the `gd` function with these arguments, as well as $\varepsilon = 10^{-8}$ and $K = 5000$, and the learning rate t which you should identify as follows: try t in the range $1, 2, 4, 8, 16, 32, 64$, and choose the value which gives the fastest convergence of Algorithm 3. Rerun your test a few (3–5) times to see how reproducible are the results for different realisations of $\mathcal{I}_{test}$. Your code should produce finite real numbers without warnings or errors in all runs.

Write a report (approximately half a page should be sufficient) summarising the average results you have obtained:

- Which value of t gives the fastest convergence? What makes smaller and larger t not optimal?
- At which iteration k the algorithm stops with this optimal t?
- What types of convergence can be seen in the loss plot?
- What is the relative error $\frac{\|UW^\top - Y\|_F}{\|Y\|_F}$ of the entire result? How it relates to the final test loss?
  Hint: for any matrix $A \in \mathbb{R}^{n\times m}$ with elements $a_{i,j}$, $\|A\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}|a_{i,j}|^2}$ is the so-called Frobenius *norm. It can be computed by calling an appropriate norm function in* `numpy`.

[4 points]

Q5: stochastic average gradient. Implement a Python function `sag(U0, W0, Y, I, Itest, t=1, eps=1e-6, K=100)` with the same inputs as in **Q3**. The function `sag` should implement Algorithm 4 and return two numpy arrays of shape `(n, r)` each, storing the two factor matrices $U, W \in \mathbb{R}^{n\times r}$ computed in the last iteration. Moreover, your code inside the function `sag` should collect the test loss values $L_{\mathcal{I}_{test},Y}(U,W)$ from all iterations carried out until the algorithm has stopped, and plot them in the logarithmic scale as a function of the iteration number k.

[3 points]

Q6: testing SAG on full data. Using the same synthetic matrix as in **Q4**, full sampling set $\mathcal{I}_{ext}$ with $p = 1$, test sampling set $\mathcal{I}_{test}$ with $\mu = 20$, $\mathcal{I} = \mathcal{I}_{ext} \backslash \mathcal{I}_{test}$, $\varepsilon = 10^{-8}$, and U_0, V_0 as suggested in Section 1.5, test the `sag` function. However, in contrast to **Q4**, now let `sag` carry out up to $K = 50000$ iterations, and choose the learning rate t which gives the fastest convergence from the range $1/64, 1/32, 1/16, 1/8, 1/4, 1/2, 1$. Rerun your tests a few (3–5) times to make sure the results are reproducible. Your code should produce finite real numbers without warnings or errors in all runs.

Write a report (approximately half a page should be sufficient) summarising the average results you have obtained:

- Which value of t gives the fastest convergence? How different is the optimal t for `sag` compared to that of `gd`?
- At which iteration k `sag` stops with this optimal t? Does it reach convergence within K iterations?
- What types of convergence can be seen in the loss plot?
- What is the relative error $\frac{\|UW^\top - Y\|_F}{\|Y\|_F}$ of the entire result?

[4 points]

Q7: tests with real data. Write a Python code to load data from the file `CW.npz` provided. It contains one matrix `Y` which is a downscaled real X-ray absorption data from a spectromicroscopy experiment. Due to imprecise measurements, the matrix Y is not exactly representable by a rank-r matrix with any $r < n$. However, it can be approximated by a low-rank matrix with a reasonable accuracy.

Write a Python code to generate $\mathcal{I}_{ext}$ of undersampling ratio $p = 1/3$, $\mathcal{I}_{test}$ with $\mu = 20$ index pairs, $\mathcal{I} = \mathcal{I}_{ext} \backslash \mathcal{I}_{test}$, set $r = 4$ and generate the initial guess $U_0, W_0 \in \mathbb{R}^{n\times r}$ using `initial(n,r)`.

Write a Python code to run both algorithms (`gd` and `sag`) with the data Y from `CW.npz`, first with ε and K from the previous questions. Find the smallest test loss $L_{low} \approx \min_{U,W} L_{\mathcal{I}_{test},Y}(U, W)$ you can realistically achieve (note that it's not 0 now). Now for each algorithm, set $\varepsilon = 2L_{low}$ and find the learning rate t which gives the fastest convergence. You may need to adjust the maximal number of iterations to achieve ε.

Rerun your tests a few (2–3) times to make sure the results are reproducible. Your code should produce finite real numbers without warnings or errors in all runs.

Write a Python code to plot Y and the most accurate $UW^\top$ you managed to compute as images.

Write a report (approximately half a page should be sufficient) summarising the average results you have obtained:

- What is L_{low}? (note that it's independent of a particular algorithm)
- Which value of t gives the fastest convergence, for each algorithm?
- At which iteration k each algorithm stops with this t?
- What types of convergence can be seen in the loss plots?
- What are the relative errors $\frac{\|UW^\top - Y\|_F}{\|Y\|_F}$ of the entire results?
- How does the behaviour of the algorithms compare overall to the full data tests in **Q4** and **Q6**?

[5 points]

Appendix A. Derivation of the pointwise loss gradient

Differentiating

$$\ell_{i_1,i_2}(U, W) := \left(\sum_{j=0}^{r-1} u_{i_1,j} w_{i_2,j} - y_{i_1,i_2} \right)^2 .$$

over the element $u_{i_1^*,j^*}$ for some $i_1^* = 0, \ldots, n-1$ and $j^* = 0, \ldots, r-1$. We get

$$\frac{\partial \ell_{i_1,i_2}(U, W)}{\partial u_{i_1^*,j^*}} = 2 \left(\sum_{j=0}^{r-1} u_{i_1,j} w_{i_2,j} - y_{i_1,i_2} \right) \sum_{j'=0}^{r-1} \left[w_{i_2,j'} \delta_{i_1,i_1^*} \delta_{j',j^*} \right],$$

where

$$\delta_{a,b} = \begin{cases} 1, & a = b, \\ 0, & \text{otherwise.} \end{cases}$$

This gives

$$\frac{\partial \ell_{i_1,i_2}(U, W)}{\partial u_{i_1^*,j^*}} = \begin{cases} 2 \left(\sum_{j=0}^{r-1} u_{i_1,j} w_{i_2,j} - y_{i_1,i_2} \right) w_{i_2,j^*}, & i_1 = i_1^*, \\ 0, & \text{otherwise.} \end{cases}$$