Homework 4

10-405/605: Machine Learning with Large Datasets

Due Monday, March 24th at 11:59PM Eastern Time

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

IMPORTANT: Be sure to highlight where your solutions are for each question when submitting to Gradescope otherwise you will be marked 0 and will need to submit regrade request for every solution un-highlighted in order for fix it!

Note that Homework 4 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 the following:

1. Written Section 1 *[28 points]*
2. Programming Section 1 *[40 points]*
3. Programming Section 2 *[40 points]*

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 108.

Programming: The programming in Programming Section 1 will be autograded on Gradescope. The programming in Programming Section 2 will **NOT** be autograded. However, you are required to upload both your completed notebooks (section 1 and section 2) to Gradescope, otherwise you will not receive credit for the programming sections.

1 Written [28 Points]

1.1 Simple Neural Network

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

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

Here $\mathbf{x} \in \mathbb{R}^D$ 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 a bias 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 a bias term corresponding to the second layer. Note also that the max in step 2 is applied element-wise.

We will consider two ways to build a gradient-descent learning system. The first way will be to find the symbolic gradient of the loss, L, with respect to the weights $W^{(\ell)}$ and biases $b^{(\ell)}$ for $\ell = 1, 2$, using back-propagation. To write down symbolic derivatives, it may be convenient to use the Kronecker delta:

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

(a) *[4 points]* Symbolic differentiation for $(\frac{\partial L}{\partial W^{(2)}})$: First, derive $\frac{\partial L}{\partial p_j}$, and $\frac{\partial p_i}{\partial W^{(2)}_{j,k}}$. Then, derive $\frac{\partial L}{\partial W^{(2)}_{j,k}}$ following techniques from lecture. Then write down an expression for the matrix $\frac{\partial L}{\partial W^{(2)}}$.

(b) *[3 points]* Symbolic differentiation for $(\frac{\partial L}{\partial b^{(2)}})$: First, derive $\frac{\partial p_i}{\partial b^{(2)}_j}$. Then, derive $\frac{\partial L}{\partial b^{(2)}_j}$, using already calculated solutions. Then write down an expression for the vector $\frac{\partial L}{\partial b^{(2)}}$.

(c) *[4 points]* Symbolic differentiation for $(\frac{\partial L}{\partial q})$. First, derive $\frac{\partial p_i}{\partial h_j}$ and $\frac{\partial h_j}{\partial q_i}$. Then, derive $\frac{\partial L}{\partial h_i}, \frac{\partial L}{\partial q_i}$ from previously calculated results. Then write down expressions for the vector $\frac{\partial L}{\partial q}$.

(d) *[5 points]* Symbolic differentiation for $(\frac{\partial L}{\partial W^{(1)}}, \frac{\partial L}{\partial b^{(1)}})$: First, derive $\frac{\partial q_i}{\partial W^{(1)}_{j,k}}, \frac{\partial q_i}{\partial b^{(1)}_j}$. Then, derive $\frac{\partial L}{\partial W^{(1)}_{j,k}}, \frac{\partial L}{\partial b^{(1)}_i}$ using already calculated solutions, following techniques from lecture. Then write down expressions for the matrix $\frac{\partial L}{\partial W^{(1)}}$ and the vector $\frac{\partial L}{\partial b^{(1)}}$.

(e) *[2 points]* For a learning rate of alpha, how would you update the parameters $W^{(1)}_{j,k}$ if you were using stochastic gradient descent on the input $\mathbf{x}^*, \mathbf{y}^*$? The expression should be in terms of $\mathbf{x}^*$ and $\mathbf{y}^*$ but you may also use other intermediate computations like $\mathbf{q}^* = W^{(1)}\mathbf{x}^* + \mathbf{b}$, $\mathbf{h}^*$, and etc.

(f) *[2 points]* Define a Wengert list with 4 steps, using the equations from Equations 1-4, using output letter q, h, p, and L. (Hint: You will need a function with 3 inputs for steps 1 and 3).

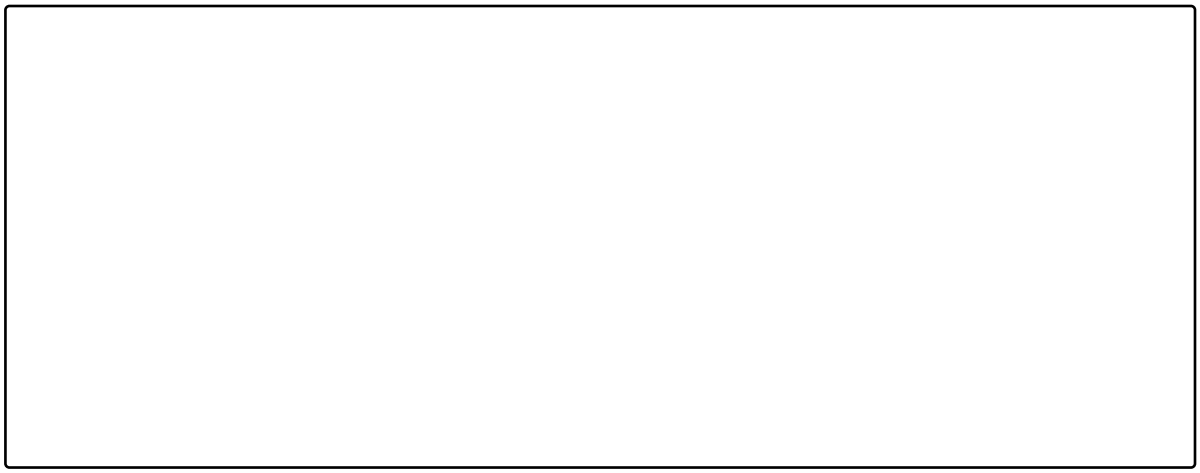
(g) *[4 points]* What are the backward functions (partial gradients with respect to an input) that must be computed directly to derive updates for the parameters? For each function, also indicate if it is equal to any symbolic derivative computed above.

(h) *[2 points]* Let's define:

$$\mathbf{x} = \begin{bmatrix} 2 \\ 7 \end{bmatrix}, \mathbf{y} = \begin{bmatrix} 4 \\ 5 \\ 8 \end{bmatrix}, W^{(1)} = \begin{bmatrix} 3 & 8 \\ 4 & 5 \end{bmatrix}, b^{(1)} = \begin{bmatrix} 4 \\ 10 \end{bmatrix}, W^{(2)} = \begin{bmatrix} 11 & 4 \\ 8 & 13 \\ 6 & 10 \end{bmatrix}, b^{(2)} = \begin{bmatrix} 9 \\ 2 \\ 15 \end{bmatrix}$$

Using these values and results from (d), find the values $\frac{\partial L}{\partial W^{(1)}_{1,1}}$ and $\frac{\partial L}{\partial b^{(1)}_{2}}$.

(i) *[2 points]* How do your experiences from 1.1 help motivate the benefits of DL frameworks like TensorFlow or PyTorch. [note: this can be a 1-2 sentence answer]



2 Programming Section 1: Optimization Methods [40 Points]

2.1 Introduction

The goal of this assignment is to gain familiarity with deep learning in PyTorch. 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 PyTorch. . You will build a simple linear regression model and train it by calling existing optimizers provided by PyTorch. 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. [***Note: you are not allowed to use any off-the-shelf optimizers in the final evaluation.***]
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 lecture.

2.2 Logistics

We provide the code template for this assignment in a Jupyter notebook. 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 notebook `hw4_part1.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.

3 Programming Section 2: LoRA [40 points]

3.1 Introduction

In this section, you will implement three distinct fine-tuning techniques for a pre-trained large language model (LLM): comprehensive fine-tuning, fine-tuning of the final layer, and **Lo**w-**R**ank **A**daptation (LoRA).

For large pre-trained models, fully retraining all parameters is often impractical due to the substantial demands on training time and memory. LoRA offers a parameter-efficient fine-tuning approach that allows for the adaptation of large pre-trained models with minimal computational overhead. LoRA accomplishes this by integrating trainable low-rank decomposition matrices into the model's architecture, focusing specifically on the layers of the Transformer model.

3.2 Dataset

In this experiment, you will fine-tune your model on the **S**tanford **S**entiment **T**reebank (SST), a widely used dataset for sentiment analysis in natural language processing. Introduced by Socher et al. in 2013, it is based on Rotten Tomatoes movie reviews and consists of 11,855 sentences parsed into 215,154 unique phrases. SST offers both fine-grained (SST-5) and binary (SST-2) sentiment labels. SST-5 includes five categories: very negative, negative, neutral, positive, and very positive, while SST-2 simplifies this to positive and negative by excluding neutral sentences. This dataset serves as a benchmark for evaluating NLP models' ability to assess sentiment by considering both individual words and sentence structure. In this experiment, you will use SST-5.

3.3 Getting lab files

After downloading and unzipping the `Source code (zip)` from the course website, you can obtain a the HW4 starter notebook. The `hw4_part2.ipynb` notebook is the file you will modify. Follow the `TODO` tags within the notebook and complete the instructions to finish the required code. You will only need to upload the completed notebook to Gradescope.

To run the notebook:

1. Upload the folder to your Google Drive.
2. Open `hw4_part2.ipynb` using Google Colaboratory (Colab).

3.4 Grading

Programming section 2 will be graded as follows:

- Empirical and free-response questions in this document [30 points]
- Manual grading of code on Gradescope [10 points]

It is strongly advised to begin your Programming Section 2 early, as running all experiments (excluding coding and debugging) on Google Colab with a T4 GPU is estimated to take approximately 3 hours. Additionally, it is recommended to code and debug on your personal computer before training on Google Colab to avoid exceeding the daily GPU time limit.

3.5 Low-Rank Adaptation (LoRA)

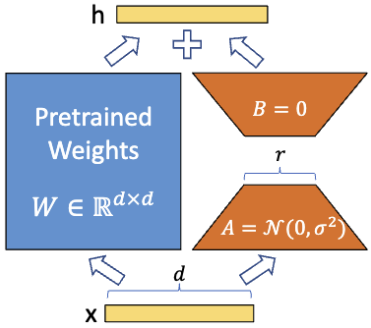


Figure 1: Illustration of Low-Rank Adaptation (LoRA)

In traditional fine-tuning methods, all model parameters are updated, which can be computationally expensive, especially for large models. LoRA addresses this issue by introducing a more efficient approach that reduces the number of parameters updated during fine-tuning. This makes the process more feasible for environments with limited resources.

LoRA restricts the update of the weight matrix $W_0 \in \mathbb{R}^{d \times k}$ using a low-rank decomposition:

$$W_0 + \Delta W = W_0 + BA$$

where $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$, with $r \ll \min(d, k)$. Only the matrices A and B are trainable, significantly reducing the number of updated parameters. The pre-trained weight matrix W_0 remains unchanged during adaptation, which helps preserve the model's learned knowledge. To maintain stability at the beginning of training, the parameters A and B are initialized in a specific way: B is initialized to zero, while A is initialized using Kaiming initialization (also known as He initialization). This ensures that the initial update $\Delta W = BA = 0$, allowing the model to begin training in a stable state without immediately introducing large changes to the pre-trained weights.

To ensure a balanced update, the low-rank modification is scaled by a factor α/r:

$$h = W_0 x + \frac{\alpha}{r} BAx$$

where x is the input to the layer, α is a constant in r. The scaling factor α allows for proper control over the influence of the low-rank update relative to the pre-trained weights.

LoRA is highly effective for adapting large pre-trained models, such as GPT-2 or GPT-3, to specific tasks while minimizing computational costs. It enables fine-tuning for applications such as sentiment analysis or text classification while preserving the model's general-purpose knowledge. By leveraging LoRA, large models can be efficiently adapted to new tasks and domains with limited computational resources or small datasets. This makes it a valuable tool in modern machine learning pipelines, particularly for deploying models in resource-constrained settings.

Before starting this coding experiment, it is highly recommended to review the following resources to gain a deeper understanding of LoRA: Low-Rank Adaptation of Large Language Models (LoRA) - Paper and Introduction to LoRA - YouTube Video.

3.6 Attention in Transformer-based Models

In this homework, you will be experimenting with the "bidirectional encoder representations from transformers" (BERT) model. BERT is a transformer-based language model (LM) that relies heavily on the attention

mechanism as a core component. The model contains numerous matrix multiplication operations, which are suitable points to apply LoRA.

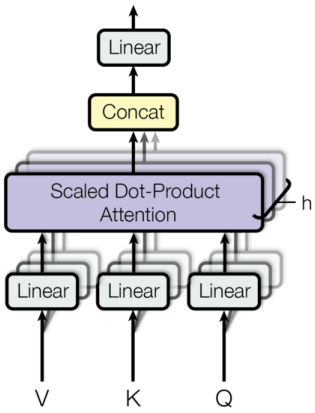


Figure 2: Illustration of the multi-head scaled dot-product attention mechanism.

In the multi-head attention layer of a transformer, a query (Q) and a set of key-value pairs (K and V) are used to compute an output. The output is a weighted sum of the values, where the weights are determined by the compatibility between the query and the keys. This process is mathematically defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

For the purpose of the assignment, it is sufficient to focus on identifying where matrix multiplications occur in the attention mechanism, and apply the LoRA technique to those matrices. However, if you're interested in learning more about transformers and the attention mechanism, you can explore the paper Attention Is All You Need (2017) and the following blogs by an OpenAI researcher: The Transformer Family by Lilian Weng (2020) and Attention Mechanism by Lilian Weng (2018).

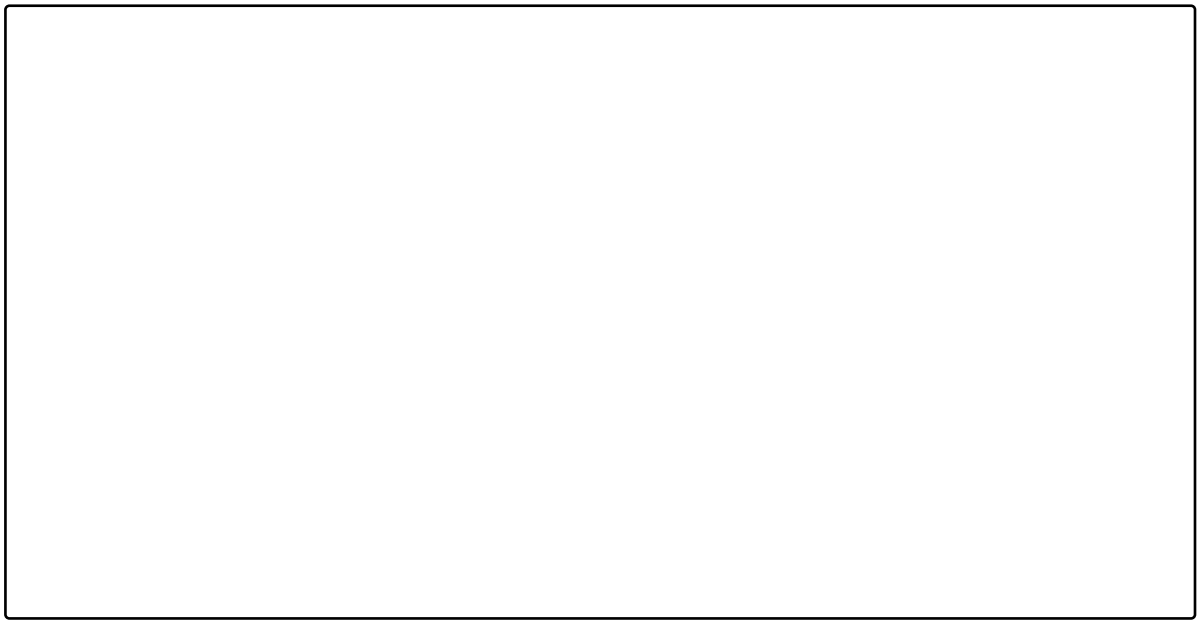
3.7 Questions

Please run full fine-tuning, last layer fine-tuning, and LoRA fine-tuning (add LoRA to all query and value weights) with 20 epochs, learning_rate $= 1 \times 10^{-4}$, $\alpha = 32$, $r = 16$, lora_dropout $= 0.4$, then answer questions (a) - (e).

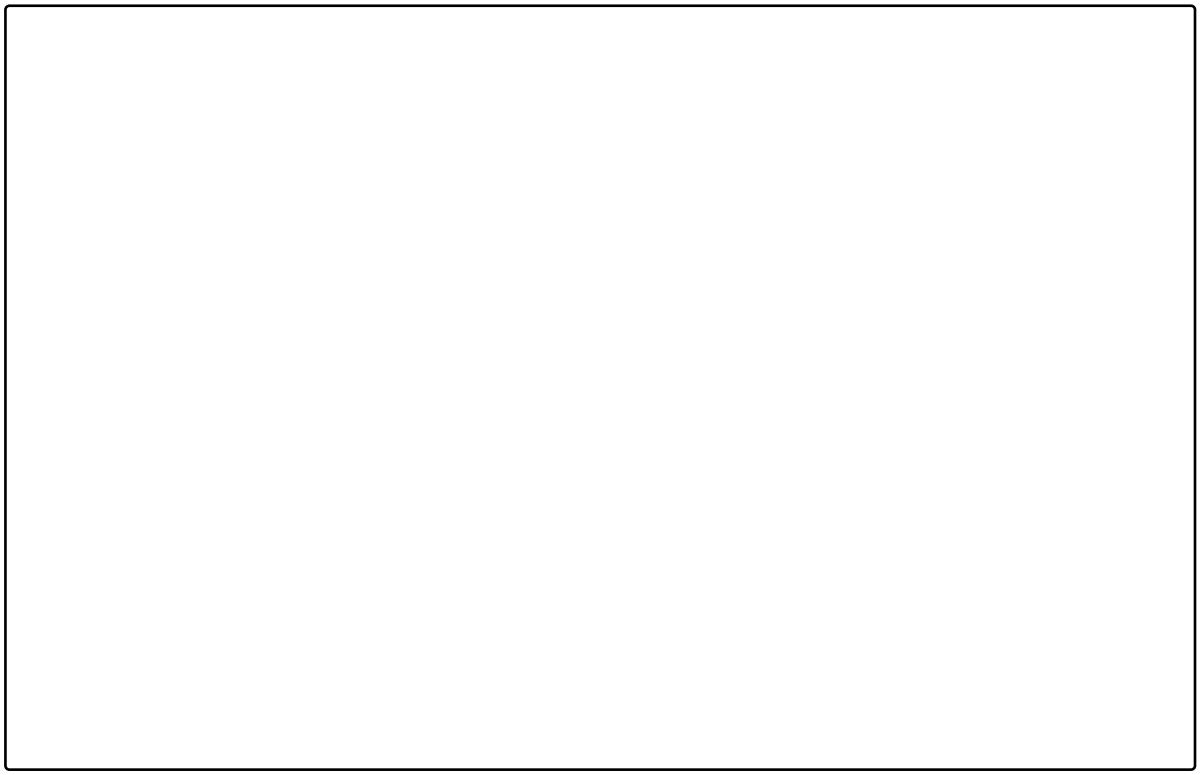
(a) *[3 points]* Plot three pie charts to illustrate the distribution of trainable parameters compared to the total parameters for the three fine-tuning methods.

(b) *[3 points]* Plot a line graph illustrating the training loss against epochs for the three fine-tuning methods.

(c) *[3 points]* Plot a line graph illustrating the val/dev accuracy against epochs for the three fine-tuning methods.



(d) *[3 points]* Plot a bar chart comparing validation accuracy and F1 score for the three fine-tuning methods.



(e) *[4 points]* Based on your earlier visualizations, how does the performance of fine-tuning with a LoRA model compare to full fine-tuning and last layer fine-tuning? Please provide a brief discussion, including comments on the convergence analysis of validation loss.

(f) *[6 points]* Please run full fine-tuning, last layer fine-tuning, and LoRA fine-tuning (add LoRA to all query and value weights) with 20 epochs, learning_rate $= 1 \times 10^{-4}$, $\alpha = 32$, lora_dropout $= 0.4$, and ranks of {1, 16, 256}, then report the best validation accuracies (NOT the last round validation accuracies) for each rank.

Rank (r)	**Best Test Accuracy**
1	
16	
256	

(g) *[4 points]* How does increasing the rank r, while keeping the scaling factor α constant, affect the model's performance? What might be the reason for this behavior? Based on the results, what trade-offs might be involved in choosing different ranks for fine-tuning a model using LoRA?

(h) *[4 points]* In your implementation, does LoRA lead to an increase in forward propagation time at training stage compared to full fine-tuning? If so, can it be eliminated? If not, please explain the reason. How about during inference time?

Hint: $h = W_0x + \frac{\alpha}{r}BAx = (W_0x + \frac{\alpha}{r}BA)x$.

4 Collaboration Questions

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

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

2. (a) Did you give any help whatsoever to anyone in solving this assignment?

   (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?

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