

Assignment 3

Version: 1.0

Version Release Date: 2023-03-04

Deadline: Friday, Mar. 31, at 11:59pm.

Submission: You must submit two files through MarkUs¹: (1) a PDF file containing your writeup, titled `a3-writeup.pdf`, and (2) your code file `nmt.ipynb`, `bert.ipynb`, `clip.ipynb`. There will be sections in the notebook for you to write your responses. Your writeup must be typed. Make sure that the relevant outputs (e.g. `print_gradients()` outputs, plots, etc.) are included and clearly visible.

See the syllabus on the course website² for detailed policies. You may ask questions about the assignment on Piazza³. *Note that 10% of the assignment mark (worth 2 pts) may be removed for lack of neatness.*

You may notice that some questions are worth 0 pt, which means we will not mark them in this Assignment. Feel free to skip them if you are busy. However, you are expected to see some of them in the midterm. So, we won't release the solution for those questions.

The teaching assistants for this assignment are Sheng Jia, Yongchao Zhou, Ke Zhao, and John Giorgi. Send your email with subject “[CSC413] A3 ...” to `csc413-2023-01-tas@cs.toronto.edu` or post on Piazza with the tag `a1`.

¹<https://markus.teach.cs.toronto.edu/2023-01>

²<https://uoft-csc413.github.io/2023/assets/misc/syllabus.pdf>

³<https://piazza.com/class/lcp8mp3f9dl71p>

Important Instructions

Read the following before attempting the assignment.

Overview and Expectations

You will be completing this assignment with the aid of large language models (LLMs) such as ChatGPT, text-davinci-003, or code-davinci-002. To alleviate the unnecessary steps related to generating results and screenshotting, we have provided the GPT-generated solution with minimum prompting effort in **ChatGPT**: clauses. The goal is to help you (*i*) develop a solid understanding of the course materials, and (*ii*) gain some insight in problem-solving with LLMs. Think of this as analogous to (*i*) understanding the rules of addition and multiplication, and (*ii*) learning how to use a calculator. Note that LLMs may not be a reliable “calculator” (yet) — as you will see, GPT-like models can generate incorrect and contradicting answers. It is, therefore important that you have a good grasp of the lecture materials, so that you can evaluate the correctness of the model output, and also prompt the model toward the correct solution.

Prompt engineering. In this assignment, we ask that you try to (*i*) solve the problems yourself, and (*ii*) use LLMs to solve a selected subset of them. You will “guide” the LLMs toward desired outcomes by typing *text prompts* into the models. There are a number of different ways to prompt an LLM, including direct copy-pasting L^AT_EX strings of a written question, copying function docstrings, or interactively editing the previously generated results. Prompting offers a natural and intuitive interface for humans to interact with and use LLMs. However, LLM-generated solutions depend significantly on the quality of the prompt used to steer the model, and most effective prompts come from a deep understanding of the task. You can decide how much time you want to spend as a university student vs. a prompt engineer, but we’d say it’s probably not a good idea to use more than 25% of your time on prompting LLMs. See Best Practices below for the basics of prompt engineering.

What are LLMs good for? We have divided the assignment problems into the following categories, based on our judgment of how difficult it is to obtain the correct answer using LLMs.

- **[Type 1]** LLMs can produce almost correct answers from rather straightforward prompts, e.g., minor modification of the problem statement.
- **[Type 2]** LLMs can produce partially correct and useful answers, but you may have to use a more sophisticated prompt (e.g., break down the problem into smaller pieces, then ask a sequence of questions), and also generate multiple times and pick the most reasonable output.
- **[Type 3]** LLMs usually do not give the correct answer unless you try hard. This may include problems with involved mathematical reasoning or numerical computation (many GPT models do not have a built-in calculator).
- **[Type 4]** LLMs are not suitable for the problem (e.g., graph/figure-related questions).

Written Assignment

What you have to submit for this part

See the top of this handout for submission directions. Here are the requirements.

- The zero point questions (in black below) will not be graded, but you are more than welcome to include your answers for these as well in the submission.
- For (nonzero-point) questions labeled [Type 1] [Type 2] you need to submit your own solution. Your own solution can be a copy-paste of the LLM output (if you verify that it is correct), but make sure you **cite the model properly**.
- For (nonzero-point) questions in [Type 3] [Type 4] you only need to submit your own written solution, but we encourage you to experiment with LLMs on some of them.

For reference, here is everything you need to hand in for the first half of the PDF report `a3-writeup.pdf`.

- **Problem 1:** 1.1.2[Type 2] , 1.2.2[Type 1]
- **Problem 2:** 2.1.1[Type 1] , 2.1.2[Type 4] , 2.2[Type 4]

1 Robustness and Regularization

Adversarial examples plague many machine learning models, and their existence makes the adoption of ML for high-stakes applications undergo increasingly more regulatory scrutiny. The simplest way to generate an adversarial examples is using the untargeted fast gradient sign method (FGSM) from Goodfellow et al. [2014]:

$$\mathbf{x}' \leftarrow \mathbf{x} + \epsilon \operatorname{sgn}(\nabla_{\mathbf{x}} \mathcal{L}(f(\mathbf{x}; \mathbf{w}), y))$$

where $\mathbf{x} \in \mathbb{R}^d$ is some training example we want to perturb, y is the label for that example, and ϵ is a positive scalar chosen to be small enough such that the ground truth class of \mathbf{x}' is the same as that of \mathbf{x} according to human perception, yet large enough such that our classifier f misclassifies \mathbf{x}' while correctly classifying \mathbf{x} . Read about how the `sgn()` function works here (https://en.wikipedia.org/wiki/Sign_function).

Note that we are taking the gradient of $\mathcal{L}(f(\mathbf{x}; \mathbf{w}), y)$ with respect to the *input* \mathbf{x} instead of the weights \mathbf{w} , and that we are adding this gradient rather than subtracting it since the goal here is to increase the loss on \mathbf{x}' .

For the rest of the question, we assume we are dealing with a binary linear classifier that outputs a scalar logit as follows:

$$f(\mathbf{x}; \mathbf{w}) = \mathbf{w}^\top \mathbf{x},$$

where $\mathbf{w} \in \mathbb{R}^d$ where d is dimension of the input \mathbf{x} , so $f : \mathbb{R}^d \rightarrow \mathbb{R}$. For the remainder of the question, we ignore the loss function, and simply try to reduce the output predicted by the classifier f .

To simplify our analysis, assume that the linear classifier outputs a positive logit on the input \mathbf{x} , $\mathbf{w}^\top \mathbf{x} > 0$. The attack is performed on the outputted logits directly to change the model's prediction from positive to negative. The attack now becomes:

$$\mathbf{x}' \leftarrow \mathbf{x} - \epsilon \operatorname{sgn}(\nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w})),$$

where we are trying to decrease the outputted logit.

1.1 Adversarial Examples

1.1.1 Bounding FGSM [0pt]

To understand why the $\operatorname{sgn}()$ function is used, compute the ℓ^∞ norm of $\delta_1 = \epsilon \operatorname{sgn}(\nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w}))$ and $\delta_2 = \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w})$. Under what conditions does $\|\delta_1\|_\infty = \|\delta_2\|_\infty$? What guarantee does the $\operatorname{sgn}()$ function give us on the ℓ^∞ norm of the perturbation?

ChatGPT:

The ℓ^∞ norm of a vector δ is defined as $\|\delta\|_\infty = \max_i |\delta_i|$, where $[\delta]_i$ denotes the i th entry of the vector. Using this definition, we can compute the norms of δ_1 and δ_2 as follows:

$$\begin{aligned} \|\delta_1\|_\infty &= \max_i |[\epsilon \operatorname{sgn}(\nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w}))]_i| \\ &= \epsilon \max_i |\operatorname{sgn}([\nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w})]_i)| \\ &= \epsilon \\ \|\delta_2\|_\infty &= \max_i |[\epsilon \nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w})]_i| \\ &= \epsilon \max_i |\nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w})_i|. \end{aligned}$$

The conditions under which $\|\delta_1\|_\infty = \|\delta_2\|_\infty$ are when $\nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w})$ has all its entries either positive or negative. In this case, the $\operatorname{sgn}()$ function doesn't change the signs of the entries, and hence the magnitudes of the two vectors are equal. Otherwise, $\|\delta_1\|_\infty$ is strictly less than $\|\delta_2\|_\infty$.

The $\operatorname{sgn}()$ function gives us a guarantee on the ℓ^∞ norm of the perturbation by ensuring that it is bounded by ϵ . In other words, $\|\delta_1\|_\infty \leq \epsilon$, which means that the perturbation applied to the input \mathbf{x} is limited to a small range. This is important for ensuring that the adversarial example is visually similar to the original example and is imperceptible to human eyes.

1.1.2 Prediction under Attack [1pt]

If we remove the $\operatorname{sgn}()$ function from the FGSM, we are left with just the FGM

$$\mathbf{x}' \leftarrow \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w})$$

Let us construct \mathbf{x}' using the FGM. Write down the model output under the adversarial attack $f(\mathbf{x}'; \mathbf{w})$ as a function of $\epsilon, \mathbf{x}, \mathbf{w}$ in a closed form.

ChatGPT: Using the FGM, we perturb \mathbf{x} to obtain \mathbf{x}' . The perturbation is given by $\epsilon \nabla_{\mathbf{x}} \mathcal{L}(f(\mathbf{x}; \mathbf{w}), y)$. In our case, since we are simply trying to reduce the output predicted by the classifier, we can set

y to be the opposite of the true label of \mathbf{x} , i.e., if \mathbf{x} is classified as positive, we set $y = -1$, and if \mathbf{x} is classified as negative, we set $y = 1$. Thus, we have:

$$\begin{aligned}\mathbf{x}' &= \mathbf{x} + \epsilon \nabla_{\mathbf{x}} \mathcal{L}(f(\mathbf{x}; \mathbf{w}), y) \\ &= \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}; \mathbf{w}) y \\ &= \mathbf{x} - \epsilon y \mathbf{w}\end{aligned}$$

where we used the fact that the gradient of a linear function $f(\mathbf{x}; \mathbf{w})$ with respect to \mathbf{x} is simply \mathbf{w} .

Now, to obtain the predicted output of the classifier $f(\mathbf{x}'; \mathbf{w})$, we substitute the expression for \mathbf{x}' :

$$\begin{aligned}f(\mathbf{x}'; \mathbf{w}) &= \mathbf{w}^\top \mathbf{x}' \\ &= \mathbf{w}^\top (\mathbf{x} - \epsilon y \mathbf{w}) \\ &= \mathbf{w}^\top \mathbf{x} - \epsilon y \mathbf{w}^\top \mathbf{w} \\ &= f(\mathbf{x}; \mathbf{w}) - \epsilon y \|\mathbf{w}\|^2\end{aligned}$$

Thus, we obtain the closed-form expression for the predicted output of the classifier under the adversarial attack \mathbf{x}' as:

$$f(\mathbf{x}'; \mathbf{w}) = f(\mathbf{x}; \mathbf{w}) - \epsilon y \|\mathbf{w}\|^2$$

1.2 Gradient Descent and Weight Decay

The most trivial though impractical way of making a classifier robust to adversarial examples is to set $\mathbf{w} = \mathbf{0}$ such that $f(\mathbf{x}; \mathbf{w}) = 0$ for any \mathbf{x} . However, this just computes a constant function, and is not useful. Intuitively, it looks like the smaller the norm of \mathbf{w} , then less the output will change when changing \mathbf{x} . We explore if this is always the case.

Suppose we have a design matrix $X \in \mathbb{R}^{n \times d}$ where n is the number of samples and d is the dimensionality, and a target vector $\mathbf{t} \in \mathbb{R}^n$. We can define the objective of linear regression as

$$\min_{\mathbf{w}} \frac{1}{2n} \|X\mathbf{w} - \mathbf{t}\|_2^2$$

If we penalize the squared ℓ^2 norm of the weights, we end up with ridge regression:

$$\mathbf{w}_{ridge}^* = \arg \min_{\mathbf{w}} \frac{1}{2n} \|X\mathbf{w} - \mathbf{t}\|_2^2 + \lambda \|\mathbf{w}\|_2^2,$$

where λ is the weight decay coefficient, $\lambda > 0$.

1.2.1 Toy Example [0pt]

Consider the following dataset $\mathcal{D} = \{(\underbrace{(1, 1)}_{\mathbf{x}_1}, \underbrace{1}_{t_1}), (\underbrace{(-1, -1)}_{\mathbf{x}_2}, \underbrace{-1}_{t_2})\}$ visualized in Figure 1 for your convenience. Draw the set of optimal solutions \mathbf{w}^* (it's a line) in weight space (which is different from Figure 1 which is in data space) with \mathbf{w}_1^* on the x-axis, and \mathbf{w}_2^* on the y-axis. On the same plot, draw the contours of $\mathbf{w}^\top \mathbf{w}$. Is there a contour value for which the intersection between the set of optimal solutions and $\mathbf{w}^\top \mathbf{w}$ is a single point? If so, provide the coordinate of the point. What does this imply about the uniqueness of ridge regression solution for this dataset?

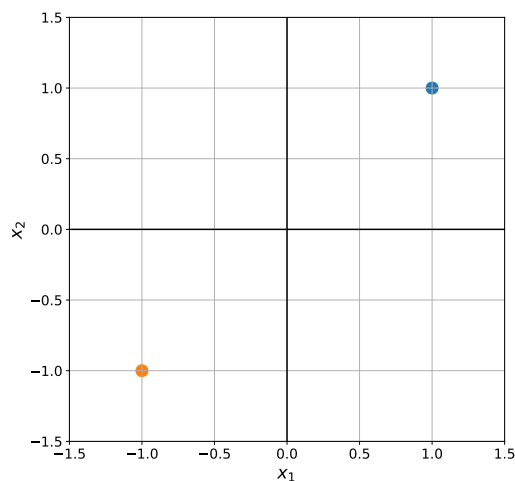


Figure 1: 2D plot of the training data.

1.2.2 Closed Form Ridge Regression Solution [1pt]

Recall the solution to plain regression is $\mathbf{w}^* = (X^\top X)^{-1} X^\top \mathbf{t}$. Write down the closed-form solution to ridge regression in matrix form, \mathbf{w}_{ridge}^* . Show your work.

ChatGPT: The objective function of ridge regression can be written as

$$\begin{aligned} \frac{1}{2n} \|X\mathbf{w} - \mathbf{t}\|_2^2 + \lambda \|\mathbf{w}\|_2^2 &= \frac{1}{2n} (\mathbf{w}^\top X^\top X \mathbf{w} - 2\mathbf{t}^\top X \mathbf{w} + \mathbf{t}^\top \mathbf{t}) + \lambda \mathbf{w}^\top \mathbf{w} \\ &= \frac{1}{2n} \mathbf{w}^\top (X^\top X + n\lambda I) \mathbf{w} - \frac{1}{n} \mathbf{t}^\top X \mathbf{w} + \frac{1}{2n} \mathbf{t}^\top \mathbf{t}. \end{aligned}$$

To find the optimal \mathbf{w} that minimizes the objective function, we set the gradient with respect to \mathbf{w} to zero:

$$\begin{aligned} \nabla_{\mathbf{w}} \left[\frac{1}{2n} \mathbf{w}^\top (X^\top X + n\lambda I) \mathbf{w} - \frac{1}{n} \mathbf{t}^\top X \mathbf{w} \right] &= \frac{1}{n} (X^\top X + n\lambda I) \mathbf{w} - \frac{1}{n} X^\top \mathbf{t} = 0 \\ \Rightarrow \mathbf{w}_{ridge}^* &= (X^\top X + n\lambda I)^{-1} X^\top \mathbf{t}. \end{aligned}$$

Thus, the closed-form solution to ridge regression in matrix form is $\mathbf{w}_{ridge}^* = (X^\top X + n\lambda I)^{-1} X^\top \mathbf{t}$.

1.2.3 Adversarial Attack under Weight Decay [0pt]

Previously, we derived model output under the FGM adversarial attack $f(\mathbf{x}' ; \mathbf{w})$ without the sign function. Here, let us consider attacking the ridge regression solution. For any adversarial attacks, we first need to choose the appropriate amount of adversarial perturbation added to the original inputs. In FGM, the perturbation amount is decided by setting ϵ , larger ϵ corresponds to larger perturbation. So, how much perturbation is necessary to fool the model to output zero, that is $f(\mathbf{x}' ; \mathbf{w}_{ridge}^*) = 0$, with weight decay?

To answer this question concretely, let us consider a 1-D model that takes a scalar input $x \in \mathbb{R}$ and a scalar weight $w_{ridge}^* \in \mathbb{R}$,

$$x' \leftarrow x - \epsilon \nabla_x f(x ; w_{ridge}^*).$$

Derive the analytical closed form of ϵ as a function of the weight decay coefficient λ such that $f(x' ; w_{ridge}^*) = 0$. Show your work. Does weight decay make the model more robust under FGM attack? Why?

(Hint: Substitute your 1.2.2 solution into 1.1.2 final form then set the equation to zero. Simplify.)

ChatGPT:

To derive the closed form of ϵ , we start by writing out the expression for the adversarial example using the update equation:

$$\begin{aligned} x' &= x - \epsilon; \nabla_x f(x ; w_{ridge}^*) \\ &= x - \epsilon; \nabla_x (xw^{ridge}) - \epsilon; \nabla_x \lambda w_{ridge}^2 \\ &= x - \epsilon w^{ridge} - 2\epsilon \lambda w_{ridge}^* \\ &= x - \epsilon(1 + 2\lambda)w_{ridge}^*. \end{aligned}$$

Now, we want to find the value of ϵ that satisfies $f(x' ; w_{ridge}^*) = 0$. Using the expression for x' and the definition of the linear classifier, we have:

$$\begin{aligned} f(x' ; w_{ridge}^*) &= (x - \epsilon(1 + 2\lambda)w^{ridge})w_{ridge}^* \\ &= xw^{ridge} - \epsilon(1 + 2\lambda)w_{ridge}^{*2}. \end{aligned}$$

Setting this expression to zero and solving for ϵ , we get:

$$\begin{aligned} xw^{ridge} &= \epsilon(1 + 2\lambda)w_{ridge}^{*2} \\ \epsilon &= \frac{xw^{ridge}}{(1 + 2\lambda)w_{ridge}^{*2}}. \end{aligned}$$

This is the closed form expression for ϵ as a function of the weight decay coefficient λ that satisfies $f(x' ; w_{ridge}^*) = 0$.

Now, let us consider the impact of weight decay on the adversarial robustness of the model. From the expression for ϵ , we see that the larger the value of λ , the smaller the value of ϵ for a given input x and weight w_{ridge}^* . This means that weight decay is making the model more robust to adversarial attacks under FGM, as it requires a smaller perturbation to fool the model. Intuitively, this is because weight decay encourages the model to have smaller weights, which in turn makes it less sensitive to small changes in the input. However, it is important to note that weight decay is not a silver bullet for adversarial robustness, and other techniques such as adversarial training may also be necessary to achieve robustness in practice.

1.2.4 The Adversary Strikes Back [0pt]

Now consider the 1-D case again under for the Fast Gradient Sign Method (FGSM) by including the sign function in the perturbation:

$$x' \leftarrow x - \epsilon \operatorname{sgn}(\nabla_x f(x ; w_{ridge}^*)).$$

Does weight decay make the model more robust under FGSM attack? Why?

2 Trading off Resources in Neural Net Training

2.1 Effect of batch size

When training neural networks, it is important to select an appropriate batch size. In this question, we will investigate the effect of batch size on some important quantities in neural network training.

2.1.1 Batch size vs. learning rate

Batch size affects the stochasticity in optimization, and therefore affects the choice of learning rate. We demonstrate this via a simple model called the noisy quadratic model (NQM). Despite the simplicity, the NQM captures many essential features in realistic neural network training Zhang et al. [2019].

For simplicity, we only consider the scalar version of the NQM. We have the quadratic loss $\mathcal{L}(w) = \frac{1}{2}aw^2$, where $a > 0$ and $w \in \mathbb{R}$ is the weight that we would like to optimize. Assume that we only have access to a noisy version of the gradient — each time when we make a query for the gradient, we obtain $g(w)$, which is the true gradient $\nabla\mathcal{L}(w)$ with additive Gaussian noise:

$$g(w) = \nabla\mathcal{L}(w) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2).$$

One way to reduce noise in the gradient is to use minibatch training. Let B be the batch size, and denote the minibatch gradient as $g_B(w)$:

$$g_B(w) = \frac{1}{B} \sum_{i=1}^B g_i(w), \quad \text{where } g_i(w) = \nabla\mathcal{L}(w) + \epsilon_i, \quad \epsilon_i \stackrel{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2).$$

- (a) [1pt] As batch size increases, how do you expect the optimal learning rate to change? Briefly explain in 2-3 sentences.

(Hint: Think about how the minibatch gradient noise change with B .)

ChatGPT:

As the batch size increases, the variance of the minibatch gradient decreases, which implies that a larger learning rate can be used without the risk of overshooting the minimum. Therefore, as the batch size increases, the optimal learning rate tends to increase as well. However, the relationship between batch size and learning rate can also be affected by other factors, such as the curvature of the loss function and the specific optimization algorithm used.

2.1.2 Training steps vs. batch size

For most of neural network training in the real-world applications, we often observe the relationship of training steps and batch size for reaching a certain validation loss as illustrated in Figure 2.

- (a) [1pt] For the three points (A, B, C) on Figure 2, which one has the most efficient batch size (in terms of best resource and training time trade-off)? Assume that you have access to scalable (but not free) compute such that minibatches are parallelized efficiently. Briefly explain in 1-2 sentences.
- (b) [1pt] Figure 2 demonstrates that there are often two regimes in neural network training: the noise dominated regime and the curvature dominated regime. In the noise dominated regime, the bottleneck for optimization is that there exists a large amount of gradient noise. In the curvature dominated regime, the bottleneck of optimization is the ill-conditioned loss landscape. For points A and B on Figure 2, which regimes do they belong to, and what would you do to accelerate training? Fill each of the blanks with **one** best suited option.

Point A: Regime: _____. Potential way to accelerate training: _____.

Point B: Regime: _____. Potential way to accelerate training: _____.

Options:

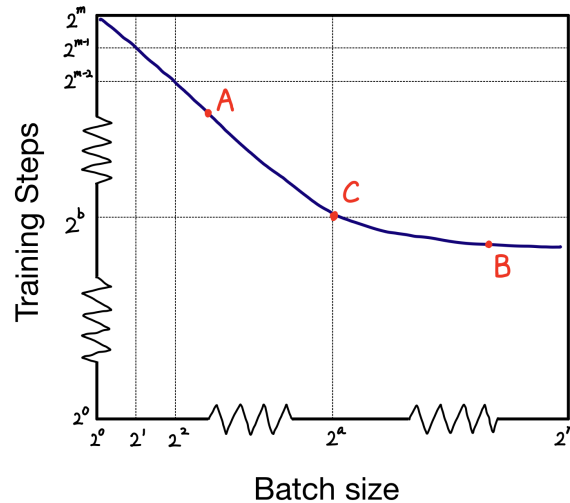


Figure 2: A cartoon illustration of the typical relationship between training steps and the batch size for reaching a certain validation loss (based on Shallue et al. [2018]). Learning rate and other related hyperparameters are tuned for each point on the curve.

- Regimes: noise dominated / curvature dominated.
- Potential ways to accelerate training: use higher order optimizers / seek parallel compute

2.2 Model size, dataset size and compute

We have seen in the previous section that batch size is an important hyperparameter during training. Besides efficiently minimizing the training loss, we are also interested in the test loss. Recently, researchers have observed an intriguing relationship between the test loss and hyperparameters such as the model size, dataset size and the amount of compute used. We explore this relationship for neural language models in this section. The figures in this question are from Kaplan et al. [2020].

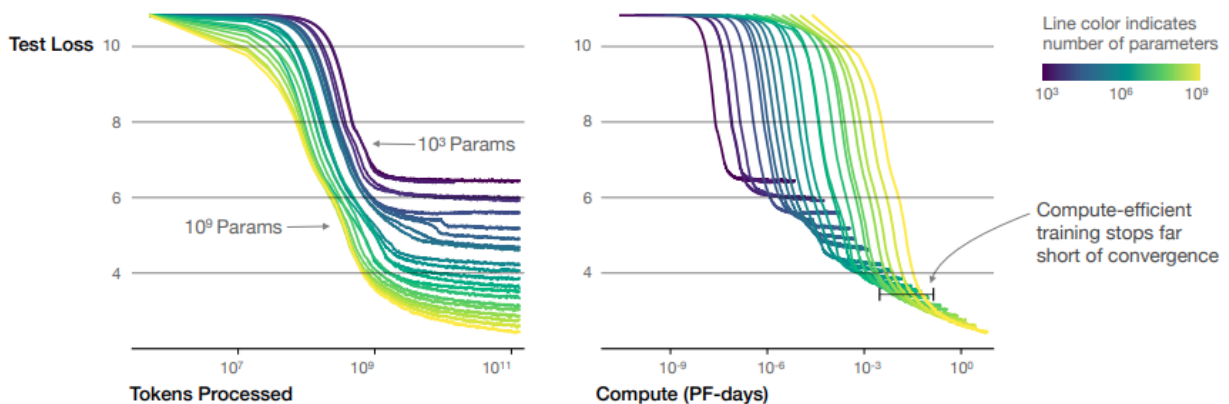


Figure 3: Test loss of language models of different sizes, plotted against the dataset size (tokens processed) and the amount of compute (in petaflop/s-days).

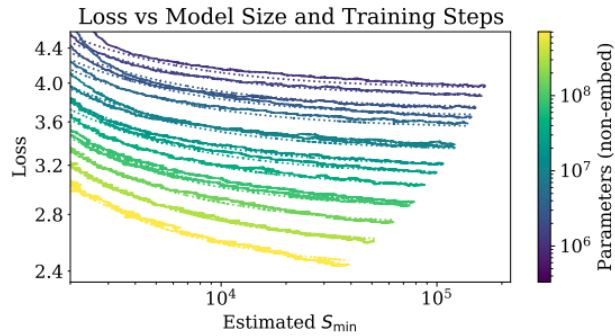


Figure 4: Test loss for different sized models after the initial transient period, plotted against the number of training steps (S_{\min}) when using the critical batch sizes (the batch sizes that separate the two regimes in Question 2.1.2).

- (a) [1pt] Previously, you have trained a neural language model and obtained somewhat adequate performance. You have now secured more compute resources (in PF-days), and want to improve the model test performance (assume you will train from scratch). Which of the following is the best option? Give a brief explanation (2-3 sentences).
- A. Train the same model with the same batch size for more steps.
 - B. Train the same model with a larger batch size (after tuning learning rate), for the same number of steps.
 - C. Increase the model size.

Programming Assignment

What you have to submit for this part

For reference, here is everything you need to hand in:

- This is the second half of your PDF report `a3-writeup.pdf`. Please include the solutions to the following problems. You may choose to export `nmt.ipynb`, `bert.ipynb`, `clip.ipynb` as a PDF and attach it to the first half of `a3-writeup.pdf`.
 - **Question 3:** 3.1[Type 2] , 3.2[Type 2] , 3.3[Type 2]
 - **Question 4:** 4.1[Type 1] , 4.3[Type 4] , 4.4[Type 4] .
 - **Question 5:** 5.2[Type 4]
- Your code file `nmt.ipynb`, `bert.ipynb`, `clip.ipynb`

Introduction

In this assignment, you will explore common tasks and model architectures in Natural Language Processing (NLP). Along the way, you will gain experience with important concepts like *attention* mechanisms (Section 3), *pretrained language models* (Section 4) and *multimodal* vision and language models (Section 5).

Setting Up

We recommend that you use **Colab**(<https://colab.research.google.com/>) for the assignment. To setup the Colab environment, just open the notebooks for each part of the assignment and **make a copy** in your own Google Drive account.

Deliverables

Each section is followed by a checklist of deliverables to add in the assignment writeup. To also give a better sense of our expectations for the answers to the conceptual questions, we've put maximum sentence limits. You will not be graded for any additional sentences.

3 Neural machine translation (NMT) [2pt]

Neural machine translation (NMT) is a subfield of NLP that aims to translate between languages using neural networks. In this section, we will train a NMT model on the toy task of English \rightarrow Pig Latin. Please read the following background section carefully before attempting the questions.

Background

The task

Pig Latin is a simple transformation of English based on the following rules:

1. If the first letter of a word is a *consonant*, then the letter is moved to the end of the word, and the letters “ay” are added to the end: `team` \rightarrow `eamtay`.
2. If the first letter is a *vowel*, then the word is left unchanged and the letters “way” are added to the end: `impress` \rightarrow `impressway`.
3. In addition, some consonant pairs, such as “sh”, are treated as a block and are moved to the end of the string together: `shopping` \rightarrow `oppingshay`.

To translate a sentence from English to Pig-Latin, we apply these rules to each word independently:

`i went shopping` \rightarrow `iway entway oppingshay`

Our goal is to build a NMT model that can learn the rules of Pig-Latin *implicitly* from (English, Pig-Latin) word pairs. Since the translation to Pig Latin involves moving characters around in a string, we will use *character-level* transformer model. Because English and Pig-Latin are similar in structure, the translation task is almost a copy task; the model must remember each character in the input and recall the characters in a specific order to produce the output. This makes it an ideal task for understanding the capacity of NMT models.

The data

The data for this task consists of pairs of words $\{(s^{(i)}, t^{(i)})\}_{i=1}^N$ where the *source* $s^{(i)}$ is an English word, and the *target* $t^{(i)}$ is its translation in Pig-Latin.⁴ The dataset contains 3198 unique (English, Pig-Latin) pairs in total; the first few examples are:

`{ (the, ethay), (family, amilyfay), (of, ofway), ... }`

In this assignment, you will investigate the effect of dataset size on generalization ability. We provide a small and large dataset. The small dataset is composed of a subset of the unique words from the book “Sense and Sensibility” by Jane Austen. The vocabulary consists of 29 tokens: the 26 standard alphabet letters (all lowercase), the dash symbol `-`, and two special tokens `<SOS>` and `<EOS>` that denote the start and end of a sequence, respectively.⁵ The second, larger dataset is obtained from Peter Norvig’s natural language corpus.⁶ It contains the top 20,000 most used English words, which is combined with the previous data set to obtain 22,402 unique words. This dataset contains the same vocabulary as the previous dataset.

⁴In order to simplify the processing of *mini-batches* of words, the word pairs are grouped based on the lengths of the source and target. Thus, in each mini-batch, the source words are all the same length, and the target words are all the same length. This simplifies the code, as we don’t have to worry about batches of variable-length sequences.

⁵Note that for the English-to-Pig-Latin task, the input and output sequences share the same vocabulary; this is not always the case for other translation tasks (i.e., between languages that use different alphabets)

⁶<https://norvig.com/ngrams/>

The model

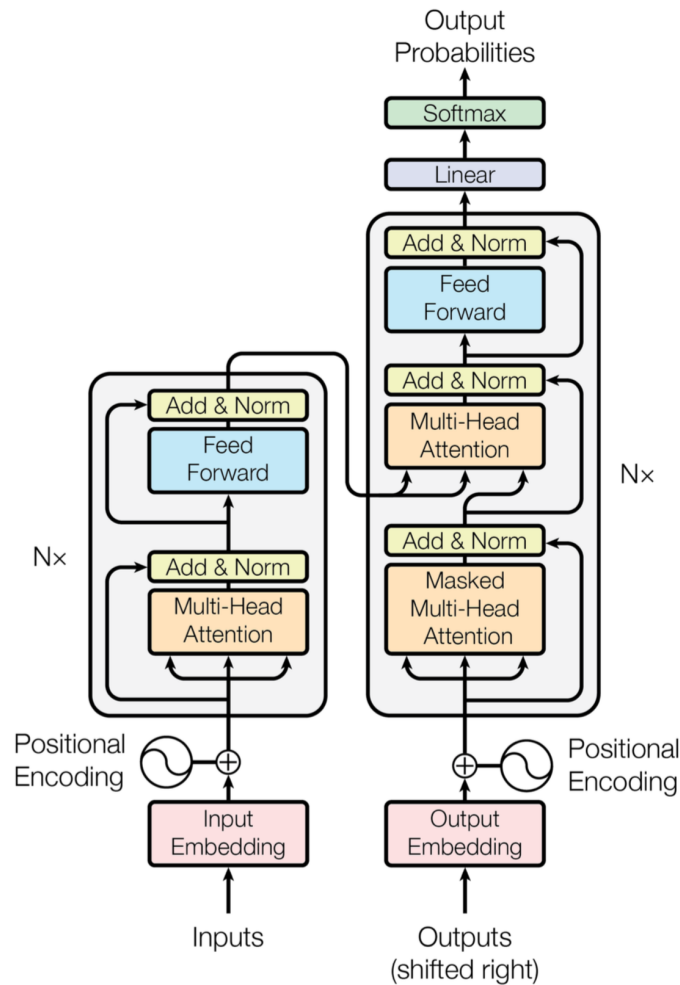


Figure 5: The transformer architecture. Vaswani et al. [2017]

Translation is a *sequence-to-sequence* (seq2seq) problem. The goal is to train a model to transform one sequence into another. A transformer model Vaswani et al. [2017] uses an encoder-decoder architecture and relies entirely on an attention mechanism to draw global dependencies between the input sequence and the output sequence. The encoder processes the input sequence in parallel using stacked self-attention and point-wise fully connected layers, as shown in Figure 5. Given the hidden representations of each input token processed through an encoder, the decoder then generates an output sequence one at a time. The model is auto-regressive when generating the output tokens.

Specifically, input characters are passed through an embedding layer before being fed into an encoder model. If H is the dimension of the encoder hidden state, we learn a $29 \times H$ embedding matrix, where each of the 29 characters in the vocabulary is assigned a H -dimensional embedding. At each time step, the decoder outputs a vector of *unnormalized log probabilities* given by a linear transformation of the decoder hidden state. When these probabilities are normalized (i.e. by passing them through a softmax), they define a distribution over the vocabulary, indicating the most probable characters for that time step. The model is trained via a cross-entropy loss between

the decoder distribution and ground-truth at each time step.

3.1 Transformers for NMT (Attention Is All You Need) [4pt]

In order to answer the following questions correctly, please make sure that you have **run the code from nmt.ipynb, Part1, Training and evaluation code prior to answering the following questions.**

1. [0.5pt] In lecture, we learnt about Scaled Dot-product Attention used in the transformer models. The function f is a dot product between the linearly transformed query and keys using weight matrices W_q and W_k :

$$\begin{aligned}\tilde{\alpha}_i^{(t)} &= f(Q_t, K_i) = \frac{(W_q Q_t)^T (W_k K_i)}{\sqrt{d}}, \\ \alpha_i^{(t)} &= \text{softmax}(\tilde{\alpha}^{(t)})_i, \\ c_t &= \sum_{i=1}^T \alpha_i^{(t)} W_v V_i,\end{aligned}$$

where, d is the dimension of the query and the W_v denotes weight matrix project the value to produce the final context vectors.

Implement the scaled dot-product attention mechanism. Fill in the `forward` methods of the `ScaledDotAttention` class. Use the PyTorch `torch.bmm` (or `@`) to compute the dot product between the batched queries and the batched keys in the forward pass of the `ScaledDotAttention` class for the unnormalized attention weights.

The following functions are useful in implementing models like this. You might find it useful to get familiar with how they work. (click to jump to the PyTorch documentation):

- `squeeze`
- `unsqueeze`
- `expand_as`
- `cat`
- `view`
- `bmm` (or `@`)

Your forward pass **needs to work** with both 2D query tensor (`batch_size x (1) x hidden_size`) and 3D query tensor (`batch_size x k x hidden_size`).

2. [0.5pt] **Implement the causal scaled dot-product attention mechanism.** Fill in the `forward` method in the `CausalScaledDotAttention` class. It will be mostly the same as the `ScaledDotAttention` class. The additional computation is to mask out the attention to the future time steps. You will need to add `self.neg_inf` to some of the entries in the unnormalized attention weights. You may find `torch.tril` or `torch.triu` handy for this part.
3. [0.5pt] We will now use `ScaledDotAttention` as the building blocks for a simplified transformer Vaswani et al. [2017] encoder.

The encoder looks like the left half of Figure 5. The encoder consists of three components:

- Positional encoding: To encode the position of each word, we add to its embedding a constant vector that depends on its position:

$$\text{pth word embedding} = \text{input embedding} + \text{positional encoding}(p)$$

We follow the same positional encoding methodology described in Vaswani et al. [2017]. That is we use sine and cosine functions:

$$\text{PE}(\text{pos}, 2i) = \sin \frac{\text{pos}}{10000^{2i/d_{\text{model}}}} \quad (3.1)$$

$$\text{PE}(\text{pos}, 2i + 1) = \cos \frac{\text{pos}}{10000^{2i/d_{\text{model}}}} \quad (3.2)$$

Since we always use the same positional encodings throughout the training, we pre-generate all those we'll need while constructing this class (before training) and keep reusing them throughout the training.

- A `ScaledDotAttention` operation.
- A following MLP.

For this question, describe why we need to represent the position of each word through this positional encoding in one or two sentences. Additionally, describe the advantages of using this positional encoding method, as opposed to other positional encoding methods such as a one hot encoding in one or two sentences.

4. [1pt] In the code notebook, we have provided an experimental setup to evaluate the performance of the Transformer as a function of hidden size and data set size. Run the Transformer model using hidden size 32 versus 64, and using the small versus large dataset (in total, 4 runs). We suggest using the provided hyper-parameters for this experiment.

Run these experiments, and report the effects of increasing model capacity via the hidden size, and the effects of increasing dataset size. In particular, report your observations on how loss as a function of gradient descent iterations is affected, and how changing model/dataset size affects the generalization of the model. Are these results what you would expect?

In your report, include the two loss curves output by `save_loss_comparison_by_hidden` and `save_loss_comparison_by_dataset`, the lowest attained validation loss for each run, and your response to the above questions.

Deliverables

Create a section in your report called **Scaled Dot Product Attention**. Add the following:

- Screenshots of your `ScaledDotProduct`, `CausalScaledDotProduct` implementations. Highlight the lines you've added. [1pt]
- Your answer to question 3. [0.5pt]
- The two loss curves plots output by the experimental setup in question 4, and the lowest validation loss for each run. [1pt]
- Your response to the written component of question 4. Your analysis should not exceed **six** sentences. [1pt]

3.2 Decoder Only NMT

In this subsection, we will train a decoder-only NMT model using the **CausalAttention** mechanism. The key difference between this approach and the previous encoder-decoder approach is that we do not encode a hidden state of the input sequence first using an encoder. Instead, we feed both the input sequence and the target sequence to a decoder simultaneously, as in Figure 6. The input sequence and the target sequence will be separated using an end-of-prompt token (**EOP**). The concatenated input to the decoder will have **SOS** token added at the beginning, and the concatenated target will have **EOS** token added at the end. In our provided notebook, the decoder will process this concatenated input using *causal attention*, but we compute the cross-entropy loss by using the output tokens from the output of **<EOP>** only.

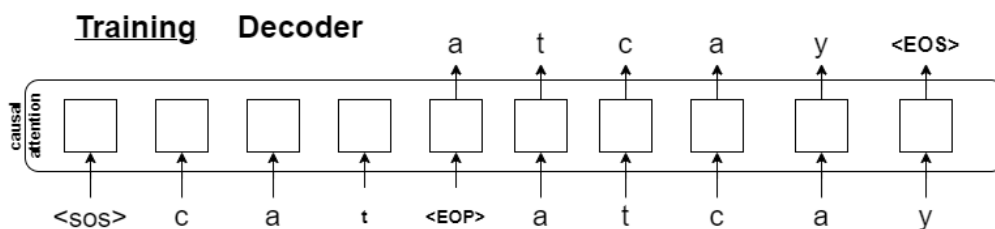


Figure 6: Training the decoder-only NMT model.

For test-time translations, we first feed the input sequence to a trained decoder, enclosed by a **SOS** token and a **EOP** token, as shown in Figure 7. We obtain the first translated token **a** in this case and concatenate the input sequence with the generated token. Then we feed the concatenated sequence to the decoder and obtain two tokens **a** and **t**. This procedure is repeated until reaching the maximum target length or generating a **<EOS>** token.

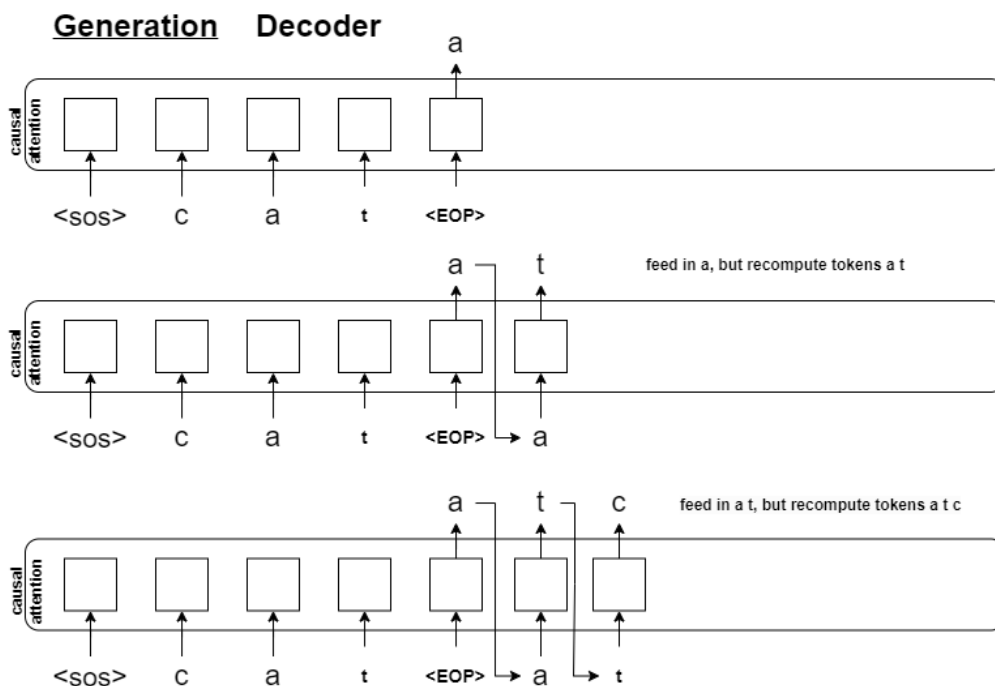


Figure 7: Translating a text using the decoder-only NMT model.

In order to answer the following questions correctly, please make sure that you have **run the code from nmt.ipynb, Part2, Training and evaluation code prior to answering the following questions.**

1. [1pt] *Construct the input tensors and the target tensors for training a decoder.* For this question, we ask you to implement the function `generate_tensors_for_training_decoder_nmt` that takes in an input sequence plus an end-of-prompt token and an output sequence plus an end-of-sentence token and returns two concatenated sequences. One has the form

`<SOS> input sequence <EOP> output sequence`

, as in the input to the decoder shown in Figure 6, and the other has the form

`input sequence <EOP> output sequence <EOS>`

2. [1pt] *Implement the forward function in `DecoderOnlyTransformer`.*
3. [1pt] *Train the model.* Now, run the training and testing code block to see the generated translation using a decoder-only model. Comment on the pros and cons of the decoder-only approach. How is the quality of your generated results compared to the ones using the encoder-decoder model?

Deliverables

Create a section in your report called **Decoder Only NMT**. Add the following:

- Your answer to question 1. (Screenshots of your implementations) [1.0pt]
- Your answer to question 2. (Screenshots of your implementations) [1.0pt]
- Your written response to the question 3. [1.0pt]

3.3 Scaling Law and IsoFLOP Profiles

This section will give you hands-on experience charting scaling law curves to forecast neural network performance. Scaling law is a fundamental concept that describes how the performance of a neural network changes with its size. Specifically, it relates the number of parameters or computations required by a neural network to achieve a certain level of performance, such as accuracy or loss. The scaling law provides a useful tool for predicting the performance of neural networks as they are scaled up or down.

IsoFLOP is a method proposed in the "Training Compute-Optimal Large Language Models" paper [Hoffmann et al., 2022] to study the scaling law of large language models. The authors of the paper used IsoFLOP to study the effect of model size on the performance of large language models and to determine the optimal model size that maximizes performance for a given computational budget.

The motivation for using IsoFLOP to forecast neural network performance is twofold. Firstly, it provides a more accurate and efficient way to explore the scaling law of large language models than traditional methods, which involve training multiple models at different sizes. Secondly, IsoFLOP allows for a better understanding of the trade-off between model size and training cost, which is crucial for designing large-scale neural network architectures that are both efficient and effective. By

leveraging IsoFLOP, researchers can gain insights into the scaling properties of neural networks, such as their accuracy and computational efficiency, and optimize their performance for specific applications and computational resources.

In this question, we will plot the scaling law curve for the decoder-only translation models from the previous section. The notebook provided trains six translation models with different model sizes and varies the FLOP counts by training for different numbers of epochs. You are asked to complete the functions to make the final IsoFLOP curve consisting of models ranging from 0.08 TFLOPs to 1.28 TFLOPs.

1. [0.5pt] Train six decoder-only translation models using the code provided and plot the validation loss as the function of FLOPs. Comment on any interesting thing you observe. Does larger model always have a smaller validation loss? (Hint: See Question 2.2)
2. [1pt] IsoFLOP Profiles. For a given FLOPs, fit a quadratic function to the validation loss and number of parameters in the log space. Find the optimal number of parameters using the quadratic function. Specifically, you need to fill the “find_optimal_params” function.
3. [1pt] Complete the Compute Optimal Model plot by fitting a linear line to the target FLOPs and the optimal model parameters. Based on the plot, estimate the optimal number of parameters when we have a compute budget of $1e15$.
4. [1pt] Plot Compute Optimal Token using the code provided. Now, given the Compute Optimal Model plot and Compute Optimal Token plot, is the training setup in Section 3.2.3 compute optimal? If not, how should we change it?

Deliverables

Create a section in your report called **Scaling Law and IsoFLOP Profiles**. Add the following:

- Your written response to the question 1. Your answer should not exceed 3 sentences. [0.5pt]
- Your answer to question 2. (Screenshots of your implementations) [1.0pt]
- Your answer to question 3. (Screenshots of your implementations). The optimal number of parameters given $1e15$ FLOPs and the process of how you estimate it. [1.0pt]
- Your written response to the question 4. Your answer should not exceed 3 sentences. [1.0pt]

4 Fine-tuning Pretrained Language Models (LMs) [2pt]

The previous sections had you train models *from scratch*. However, similar to computer vision (CV), it is now very common in natural language processing (NLP) to *fine-tune* pretrained models. Indeed, this has been described as “NLP’s ImageNet moment.”⁷ In this section, we will learn how to fine-tune pretrained *language models* (LMs) on a new task. We will use a simple classification task, where the goal is to determine whether a verbal numerical expression is *negative* (label 0), *zero* (label 1), or *positive* (label 2). For example, “eight minus ten” is negative, so our classifier should output label index 0. As our pretrained LM, we will use the popular BERT model, which uses a transformer encoder architecture. More specifically, we will explore two versions of BERT: **MathBERT** [Shen et al., 2021], which has been pretrained on a large mathematical corpus ranging from pre-kindergarten to college graduate level mathematical content and **BERTweet** [Nguyen et al., 2020], which has been pretrained on 100s of millions of tweets.

Most of the code is given to you in the notebook <https://colab.research.google.com/github/uoft-csc413/2022/blob/master/assets/assignments/bert.ipynb>. The starter code uses the *HuggingFace Transformers* library⁸, which has more than 50k stars on GitHub due to its ease of use, and will be very useful for your NLP research or projects in the future. Your task is to adapt BERT so that it can be fine-tuned on our downstream task. Before starting this section, please carefully review the background for BERT and the verbal arithmetic dataset (below).

Background

BERT

Bidirectional **E**ncoder **R**epresentations from **T**ransformers (BERT) [Devlin et al., 2019] is a LM based on the Transformer [Vaswani et al., 2017] encoder architecture that has been pretrained on a large dataset of unlabeled sentences from Wikipedia and BookCorpus [Zhu et al., 2015]. Given a sequence of tokens, BERT outputs a “contextualized representation” vector for each token. Because BERT is pretrained on a large amount of text, these contextualized representations encode useful properties of the syntax and semantics of language.

BERT has 2 pretraining objectives: (1) Masked Language Modeling (MLM), and (2) Next Sentence Prediction (NSP). The input to the model is a sequence of tokens of the form:

[CLS] Sentence A [SEP] Sentence B

where [CLS] (“class”) and [SEP] (“separator”) are special tokens. In MLM, some percentage of the input tokens are randomly “masked” by replacing them with the [MASK] token, and the objective is to use the final layer representation for that masked token to predict the correct word that was masked out⁹. In NSP, the task is to use the contextualized representation of the [CLS] token to predict whether sentence A and sentence B are consecutive sentences in the unlabeled dataset. See Figure 8 for the conceptual picture of BERT pretraining and fine-tuning.

Once pretrained, we can fine-tune BERT on a downstream task of interest, such as sentiment analysis or question-answering, benefiting from its learned contextual representations. Typically, this is done by adding a simple classifier, which maps BERT’s outputs to the class labels for our downstream task. Often, this classifier is a single linear layer + softmax. We can choose to train

⁷<https://ruder.io/nlp-imagenet/>

⁸<https://huggingface.co/docs/transformers>

⁹The actual training setup is slightly more complicated but conceptually similar. Notice, this is similar to one of the models in Programming Assignment 1!

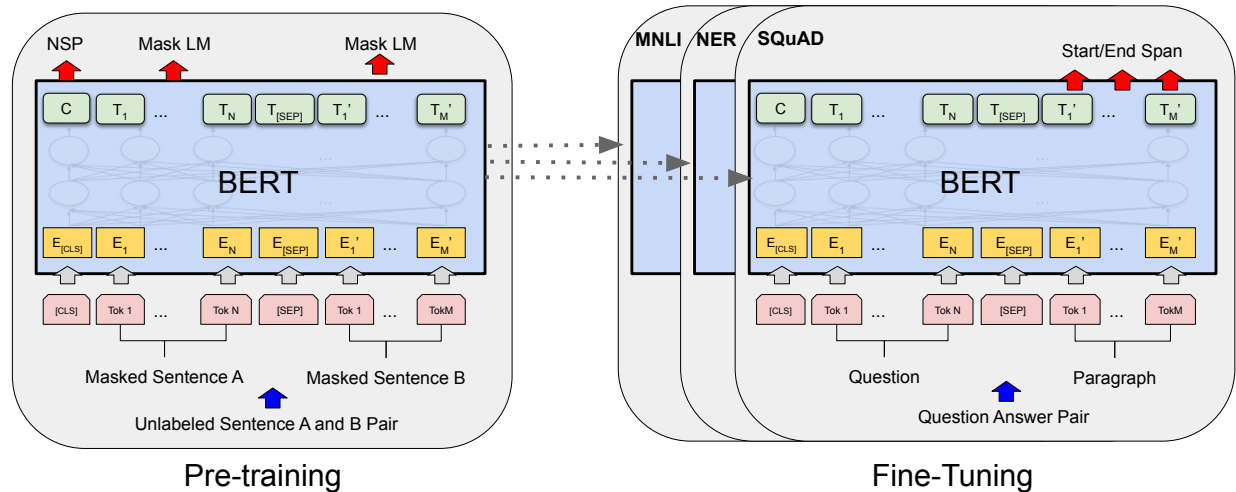


Figure 8: Overall pretraining and fine-tuning for BERT. Reproduced from BERT paper [Devlin et al., 2019]

only the parameters of the classifier, or we can fine-tune both the classifier and BERT model jointly. Because BERT has been pretrained on a large amount of data, we can get good performance by fine-tuning for a few epochs with only a small amount of labelled data.

In this assignment, you will **fine-tune BERT** on a **single sentence classification task**. Figure 9 illustrates the basic setup for fine-tuning BERT on this task. We prepend the tokenized sentence with the [CLS] token, then feed the sequence into BERT. We then take the contextualized [CLS] token representation at the last layer of BERT as input to a simple classifier, which will learn to predict the probabilities for each of the possible output classes of our task. We will use the pretrained weights of MathBERT, which uses the same architecture as BERT, but has been pretrained on a large mathematical corpus, which more closely matches our task data (see below).

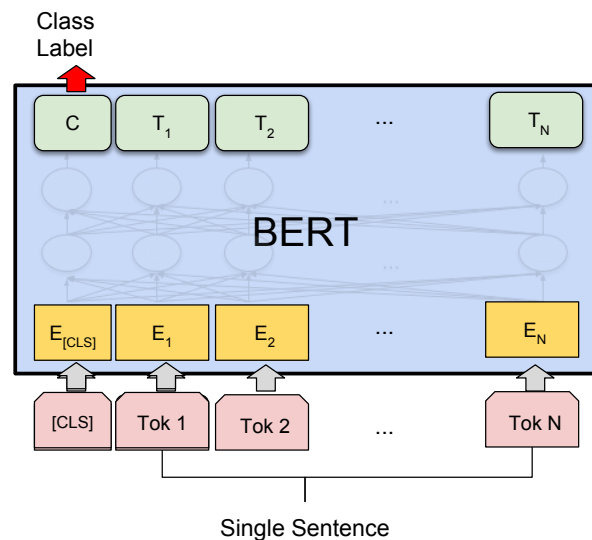


Figure 9: Fine-tuning BERT for single sentence classification by adding a layer on top of the contextualized [CLS] token representation. Reproduced from BERT paper [Devlin et al., 2019]

Verbal Arithmetic Dataset

The verbal arithmetic dataset contains pairs of input sentences and labels. The input sentences express a simple addition or subtraction. Each input is labelled as 0, 1, or 2 if it evaluates to *negative*, *zero*, or *positive*, respectively. There are 640 examples in the train set and 160 in the test set. All inputs have only **three tokens** similar to the examples shown below:

Input expression	Label	Label meaning
four minus ten	0	“negative”
eighteen minus eighteen	1	“zero”
four plus seven	2	“positive”

Questions:

- [1pt] *Add a classifier to BERT.* Open the notebook <https://colab.research.google.com/github/uoft-csc413/2022/blob/master/assets/assignments/bert.ipynb> and complete Question 1 by filling in the missing lines of code in `BertForSentenceClassification`.
- [0pt] *Fine-tune BERT.* Open the notebook and run the cells under Question 2 to fine-tune the BERT model on the verbal arithmetic dataset. If question 1 was completed correctly, the model should train, and a plot of train loss and validation accuracy will be displayed.
- [0.5pt] *Freezing the pretrained weights.* Open the notebook and run the cells under Question 3 to fine-tune only the classifiers weights, leaving BERTs weights frozen. After training, answer the following questions (no more than **four** sentences total)
 - Compared to fine-tuning (see Question 2), what is the effect on train time when BERTs weights are frozen? Why? (1-2 sentences)
 - Compared to fine-tuning (see Question 2), what is the effect on performance (i.e. validation accuracy) when BERTs weights are frozen? Why? (1-2 sentences)
- [0.5pt] *Effect of pretraining data.* Open the notebook and run the cells under Question 4 in order to repeat the fine-tuning process using the pretrained weights of BERTweet. After training, answer the following questions (no more than **three** sentences total).
 - Compared to fine-tuning BERT with the pretrained weights from MathBERT (see Question 2), what is the effect on performance (i.e. validation accuracy) when we fine-tune BERT with the pretrained weights from BERTweet? Why might this be the case? (2-3 sentences)
- [0pt] *Inspect models predictions.* Open the notebook and run the cells under Question 5. We have provided a function that allows you to inspect a models predictions for a given input. Can you find examples where one model clearly outperforms the others? Can you find examples where all models perform poorly?

Deliverables:

- The completed `BertForSentenceClassification`. Either the code or a screenshot of the code. Make sure both the `__init__` and `forward` methods are clearly visible. [1pt]
- Answer to question 3. Your answer should not exceed **4 sentences**. [0.5pt]
- Answer to question 4. Your answer should not exceed **3 sentences**. [0.5pt]

5 Connecting Text and Images with CLIP [1pt]

Throughout this course, we have seen powerful image models and expressive language models. In this section, we will connect the two modalities by exploring CLIP, a model trained to predict an image's caption to learn better image representations.

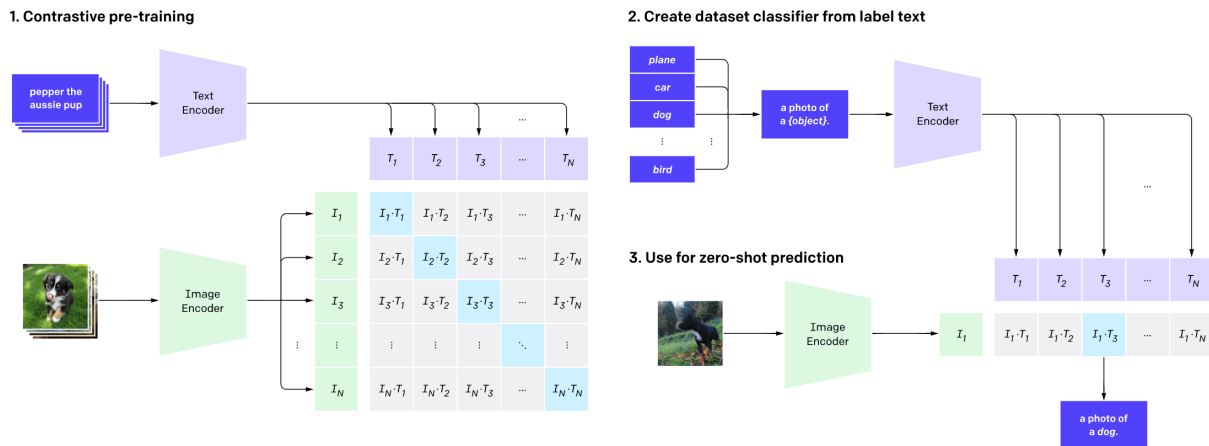


Figure 10: 1. Contrastive pre-training task that predicts the caption that corresponds to an image out of many possible captions. 2. At test time, each class is converted to a caption. This is used with 3. as a zero-shot classifier for a new image that predicts the best (image, caption) pair. Figure taken from [Radford et al., 2021a]

Background for CLIP:

The motivation behind **C**ontrastive **L**anguage-**I**mage **P**re-training (CLIP) [Radford et al., 2021b] was to leverage information from natural language to improve zero-shot classification of images. The model is pre-trained on 400 million (image, caption) pairs collected from the internet on the following task: given the image, predict which caption was paired with it out of 32,768 randomly sampled captions (Figure 10). This is done by first computing the feature embedding of the image and feature embeddings of possible captions. The cosine similarity of the embeddings is computed and converted into a probability distribution. The outcome is that the network learns many visual concepts and associates them with a name.

At test time, the model is turned into a zero-shot classifier: all possible classes are converted to a caption such as "a photo of a (class)" and CLIP estimates the best (image, caption) pair for a new image. Overall, CLIP offers many significant advantages: it does not require expensive hand-labelling while achieving competitive results and offers greater flexibility and generalizability over existing ImageNet models.

Questions:

1. [0pt] *Interacting with CLIP*. Open the notebook <https://colab.research.google.com/github/uoft-csc413/2022/blob/master/assets/assignments/clip.ipynb>. Read through Section I and run the code cells to get familiar with CLIP.

2. [1pt] *Prompting CLIP*. Complete Section II. Come up with a caption that will “prompt” CLIP to select the following target image:



Figure 11: Image that should be selected by CLIP.

Comment on the process of finding the caption: was it easy, or were there any difficulties? (no more than **one** sentence)

Deliverables:

- The caption you wrote that causes CLIP to select the image in Figure 11, as well as a brief (1 sentence) comment on the search process. [1pt]

What you need to submit

- The completed notebook files: `nmt.ipynb`, `bert.ipynb`, `clip.ipynb`.
- A PDF document titled `a3-writeup.pdf` containing your answers to the conceptual questions. You may directly append the PDF exports of the notebooks into the final `a3-writeup.pdf`.

References

- Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
- Guodong Zhang, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George E Dahl, Christopher J Shallue, and Roger Grosse. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model. *arXiv preprint arXiv:1907.04164*, 2019.
- Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.
- Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.
- Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.
- Jia Tracy Shen, Michiharu Yamashita, Ethan Prihar, Neil Heffernan, Xintao Wu, Ben Graff, and Dongwon Lee. Mathbert: A pre-trained language model for general nlp tasks in mathematics education. *arXiv preprint arXiv:2106.07340*, 2021.
- Dat Quoc Nguyen, Thanh Vu, and Anh Tuan Nguyen. Bertweet: A pre-trained language model for english tweets. *arXiv preprint arXiv:2005.10200*, 2020.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL <https://www.aclweb.org/anthology/N19-1423>.
- Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.
- Alec Radford, Ilya Sutskever, Jong Wook Kim, Gretchen Krueger, and Sandhini Agarwal. Clip: Connecting text and images, Jan 2021a. URL <https://openai.com/blog/clip/>.
- Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021b.


```

\section{Robustness and Regularization}
Adversarial examples plague many machine learning models, and their existence makes
the adoption of ML for high-stakes applications undergo increasingly more regulatory
scrutiny. The simplest way to generate an adversarial examples is using the untargeted
fast gradient sign method (FGSM) from-\cite{goodfellow2014explaining}:
\begin{align*}
\mathbf{x}' \leftarrow \mathbf{x} + \epsilon \operatorname{sign}(\operatorname{grad}_x \operatorname{loss}_{\operatorname{classifier}}(\mathbf{x}; \mathbf{b}_w))
\end{align*}

\noindent where  $\mathbf{x} \in \mathbb{R}^d$  is some training example we want to perturb,  $y$ 
is the label for that example, and  $\epsilon$  is a positive scalar chosen to be small enough
such that the ground truth class of  $\mathbf{x}'$  is the same as that of  $\mathbf{x}$  according to
human perception, yet large enough such that our classifier  $f$  misclassifies  $\mathbf{x}'$  while
correctly classifying  $\mathbf{x}$ . Read about how the  $\operatorname{sign}()$  function works
\href{https://en.wikipedia.org/wiki/Sign_function}{here}
(\url{https://en.wikipedia.org/wiki/Sign_function}).

\noindent Note that we are taking the gradient of  $\operatorname{loss}_{\operatorname{classifier}}(\mathbf{x}; \mathbf{b}_w)$  with
respect to the  $\operatorname{input} \mathbf{x}$  instead of the weights  $\mathbf{b}_w$ , and that we are adding
this gradient rather than subtracting it since the goal here is to increase the loss on
 $\mathbf{x}'$ .

\noindent For the rest of the question, we assume we are dealing with a binary linear
classifier that outputs a scalar logit as follows:
\begin{align*}
\operatorname{classifier}(\mathbf{x}; \mathbf{b}_w) = \mathbf{b}_w^{\top} \mathbf{x},
\end{align*}
\noindent where  $\mathbf{b}_w \in \mathbb{R}^d$  where  $d$  is dimension of the input  $\mathbf{x}$ , so
 $\mathbb{R}^d \rightarrow \mathbb{R}$ . For the remainder of the question, we ignore
the loss function, and simply try to reduce the output predicted by the classifier  $f$ .

\noindent To simplify our analysis, assume that the linear classifier outputs a positive logit
on the input  $\mathbf{x}$ ,  $\mathbf{b}_w^{\top} \mathbf{x} > 0$ . The attack is performed on the outputted logits
directly to change the model's prediction from positive to negative. The attack now
becomes:
\begin{align*}
\mathbf{x}' \leftarrow \mathbf{x} - \epsilon \operatorname{sign}(\operatorname{grad}_x \operatorname{classifier}(\mathbf{x}; \mathbf{b}_w)),
\end{align*}
\noindent where we are trying to decrease the outputted logit.

\subsection{Adversarial Examples}
\subsection{Bounding FGSM}
To understand why the  $\operatorname{sign}()$  function is used, compute the  $\ell^{\infty}$  norm of
 $\|\delta_1\|_{\infty} = \epsilon \operatorname{sign}(\operatorname{grad}_x \operatorname{classifier}(\mathbf{x}; \mathbf{b}_w))$  and  $\|\delta_2\|_{\infty} = \operatorname{grad}_x \operatorname{classifier}(\mathbf{x}; \mathbf{b}_w)$ . Under what conditions does  $\|\delta_1\|_{\infty} = \|\delta_2\|_{\infty}$ ? What guarantee does the  $\operatorname{sign}()$  function give us on the  $\ell^{\infty}$  norm of the perturbation?

\textbf{Solution:}

```

The ℓ^{∞} norm of a vector δ is defined as $\|\delta\|_{\infty} = \max_i |\delta_i|$, where δ_i denotes the i th entry of the vector. Using this definition, we can compute the norms of δ_1 and δ_2 as follows:

$$\begin{aligned}
\|\delta_1\|_{\infty} &= \max_i |\epsilon \operatorname{sign}(\operatorname{grad}_x \operatorname{classifier}(\mathbf{x}; \mathbf{b}_w))_i| \\
&= \epsilon \max_i |\operatorname{sign}(\operatorname{grad}_x \operatorname{classifier}(\mathbf{x}; \mathbf{b}_w))_i| \\
&= \epsilon \\
\|\delta_2\|_{\infty} &= \max_i |\operatorname{grad}_x \operatorname{classifier}(\mathbf{x}; \mathbf{b}_w)_i| \\
&= \epsilon \max_i |\operatorname{grad}_x \operatorname{classifier}(\mathbf{x}; \mathbf{b}_w)_i|.
\end{aligned}$$

The conditions under which $\|\delta_1\|_{\infty} = \|\delta_2\|_{\infty}$ are when $\operatorname{grad}_x \operatorname{classifier}(\mathbf{x}; \mathbf{b}_w)$ has all its entries either positive or negative. In this case, the $\operatorname{sign}()$ function doesn't change the signs of the entries, and hence the magnitudes of the two vectors are equal. Otherwise, $\|\delta_1\|_{\infty}$ is strictly less than $\|\delta_2\|_{\infty}$.

The $\operatorname{sign}()$ function gives us a guarantee on the ℓ^{∞} norm of the perturbation by ensuring that it is bounded by ϵ . In other words, $\|\delta_1\|_{\infty} \leq \epsilon$, which means that the perturbation applied to the input \mathbf{x} is limited to a small range. This is important for ensuring that the adversarial example is visually similar to the original example and is imperceptible to human eyes.

Regenerate response

Figure 12: Question 1.1.1

YO \section{Robustness and Regularization}

Adversarial examples plague many machine learning models, and their existence makes the adoption of ML for high-stakes applications undergo increasingly more regulatory scrutiny. The simplest way to generate an adversarial examples is using the untargeted fast gradient method (FGM):

$$\mathbf{x}' \leftarrow \mathbf{x} + \epsilon \operatorname{grad}_{\mathbf{w}} \ell(\operatorname{classifier}(\mathbf{x}; \mathbf{w}); y)$$

where $\mathbf{x} \in \mathbb{R}^d$ is some training example we want to perturb, y is the label for that example, and ϵ is a positive scalar chosen to be small enough such that the ground truth class of \mathbf{x}' is the same as that of \mathbf{x} according to human perception, yet large enough such that our classifier f misclassifies \mathbf{x}' while correctly classifying \mathbf{x} .

Note that we are taking the gradient of $\ell(\operatorname{classifier}(\mathbf{x}; \mathbf{w}); y)$ with respect to the input \mathbf{x} instead of the weights \mathbf{w} , and that we are adding this gradient rather than subtracting it since the goal here is to increase the loss on \mathbf{x}' .

For the rest of the question, we assume we are dealing with a binary linear classifier that outputs a scalar logit as follows:

$$\operatorname{classifier}(\mathbf{x}; \mathbf{w}) = \mathbf{w}^{\top} \mathbf{x}$$

where $\mathbf{w} \in \mathbb{R}^d$ where d is dimension of the input \mathbf{x} , so $f: \mathbb{R}^d \rightarrow \mathbb{R}$. For the remainder of the question, we ignore the loss function, and simply try to reduce the output predicted by the classifier f .

Let us construct \mathbf{x}' using the FGM. Write down the model output under the adversarial attack $\operatorname{classifier}(\mathbf{x}'; \mathbf{w})$ as a function of ϵ , \mathbf{x} , \mathbf{w} in a closed form.

Solution:

Using the FGM, we perturb \mathbf{x} to obtain \mathbf{x}' . The perturbation is given by $\epsilon \operatorname{grad}_{\mathbf{x}} \ell(\operatorname{classifier}(\mathbf{x}; \mathbf{w}); y)$. In our case, since we are simply trying to reduce the output predicted by the classifier, we can set y to be the opposite of the true label of \mathbf{x} , i.e., if \mathbf{x} is classified as positive, we set $y = -1$, and if \mathbf{x} is classified as negative, we set $y = 1$. Thus, we have:

$$\begin{aligned} \mathbf{x}' &= \mathbf{x} + \epsilon \operatorname{grad}_{\mathbf{x}} \ell(\operatorname{classifier}(\mathbf{x}; \mathbf{w}); y) \\ &= \mathbf{x} - \epsilon \operatorname{grad}_{\mathbf{x}} \operatorname{classifier}(\mathbf{x}; \mathbf{w}) y \\ &= \mathbf{x} - \epsilon y \mathbf{w} \end{aligned}$$

where we used the fact that the gradient of a linear function $\operatorname{classifier}(\mathbf{x}; \mathbf{w})$ with respect to \mathbf{x} is simply \mathbf{w} .

Now, to obtain the predicted output of the classifier $\operatorname{classifier}(\mathbf{x}'; \mathbf{w})$, we substitute the expression for \mathbf{x}' :

$$\begin{aligned} \operatorname{classifier}(\mathbf{x}'; \mathbf{w}) &= \mathbf{w}^{\top} \mathbf{x}' \\ &= \mathbf{w}^{\top} (\mathbf{x} - \epsilon y \mathbf{w}) \\ &= \mathbf{w}^{\top} \mathbf{x} - \epsilon y \mathbf{w}^{\top} \mathbf{w} \\ &= \operatorname{classifier}(\mathbf{x}; \mathbf{w}) - \epsilon y \|\mathbf{w}\|^2 \end{aligned}$$

Thus, we obtain the closed-form expression for the predicted output of the classifier under the adversarial attack \mathbf{x}' as:

$$\operatorname{classifier}(\mathbf{x}'; \mathbf{w}) = \operatorname{classifier}(\mathbf{x}; \mathbf{w}) - \epsilon y \|\mathbf{w}\|^2$$

Figure 13: Question 1.1.2

YO `\subsection{Gradient Descent and Weight Decay}`
 The most trivial though impractical way of making a classifier robust to adversarial examples is to set $\mathbf{bw} = \mathbf{0}$ such that $\text{classifier}(\mathbf{bx}|\mathbf{bw}) = 0$ for any \mathbf{bx} . However, this just computes a constant function, and is not useful. Intuitively, it looks like the smaller the norm of \mathbf{bw} , then less the output will change when changing \mathbf{bx} . We explore if this is always the case.

Suppose we have a design matrix $X \in \mathbb{R}^{n \times d}$ where n is the number of samples and d is the dimensionality, and a target vector $\mathbf{bt} \in \mathbb{R}^n$. We can define the objective of linear regression as

$$\min_{\mathbf{bw}} \frac{1}{2n} \|X\mathbf{bw} - \mathbf{bt}\|_2^2$$

If we penalize the squared ℓ^2 norm of the weights, we end up with ridge regression:

$$\mathbf{bw}^*_{\text{ridge}} = \arg\min_{\mathbf{bw}} \frac{1}{2n} \|X\mathbf{bw} - \mathbf{bt}\|_2^2 + \lambda \|\mathbf{bw}\|_2^2,$$

where λ is the weight decay coefficient, $\lambda > 0$.

`\subsection{Closed Form Ridge Regression Solution}` Recall the solution to plain regression is $\mathbf{bw}^* = (X^T X)^{-1} X^T \mathbf{bt}$. Write down the closed-form solution to ridge regression in matrix form, $\mathbf{bw}^*_{\text{ridge}}$. Show your work.

`\textbf{solution:}`

The objective function of ridge regression can be written as

$$\frac{1}{2n} \|X\mathbf{bw} - \mathbf{bt}\|_2^2 + \lambda \|\mathbf{bw}\|_2^2 = \frac{1}{2n} (\mathbf{bw}^T X^T X \mathbf{bw} - 2\mathbf{bt}^T X \mathbf{bw} + \mathbf{bt}^T \mathbf{bt}) + \lambda \mathbf{bw}^T \mathbf{bw}$$

$$= \frac{1}{2n} \mathbf{bw}^T (X^T X + n\lambda I) \mathbf{bw} - \mathbf{bt}^T X \mathbf{bw} + \frac{1}{2n} \mathbf{bt}^T \mathbf{bt}.$$

To find the optimal \mathbf{bw} that minimizes the objective function, we set the gradient with respect to \mathbf{bw} to zero:

$$\nabla_{\mathbf{bw}} \left[\frac{1}{2n} \mathbf{bw}^T (X^T X + n\lambda I) \mathbf{bw} - \mathbf{bt}^T X \mathbf{bw} \right] = \frac{1}{n} (X^T X + n\lambda I) \mathbf{bw} - \mathbf{bt}^T X = 0$$

$$\Rightarrow \mathbf{bw}^*_{\text{ridge}} = (X^T X + n\lambda I)^{-1} X^T \mathbf{bt}.$$

Thus, the closed-form solution to ridge regression in matrix form is $\mathbf{bw}^*_{\text{ridge}} = (X^T X + n\lambda I)^{-1} X^T \mathbf{bt}$.

Figure 14: Question 1.2.2

yo ✍

`\subsection{Gradient Descent and Weight Decay}`

The most trivial though impractical way of making a classifier robust to adversarial examples is to set $\|w\| = 0$ such that $\text{classifier}(x) = 0$ for any x . However, this just computes a constant function, and is not useful. Intuitively, it looks like the smaller the norm of w , then less the output will change when changing x . We explore if this is always the case.

Suppose we have a design matrix $X \in \mathbb{R}^{(n \times d)}$ where n is the number of samples and d is the dimensionality, and a target vector $t \in \mathbb{R}^n$. We can define the objective of linear regression as

$$\min_w \frac{1}{2n} \|Xw - t\|_2^2$$

If we penalize the squared ℓ^2 norm of the weights, we end up with ridge regression:

$$w_{\text{ridge}} = \arg \min_w \frac{1}{2n} \|Xw - t\|_2^2 + \lambda \|w\|_2^2,$$

where λ is the weight decay coefficient, $\lambda > 0$.

`\subsubsection{Adversarial Attack under Weight Decay} [1pt]`

Previously, we derived model output under the FGM adversarial attack $\text{classifier}(x')$ without the sign function. Here, let us consider attacking the ridge regression solution. For any adversarial attacks, we first need to choose the appropriate amount of adversarial perturbation added to the original inputs. In FGM, the perturbation amount is decided by setting ϵ , larger ϵ corresponds to larger perturbation. So, how much perturbation is necessary to fool the model to output zero, that is $\text{classifier}(x') = 0$, with weight decay?

To answer this question concretely, let us consider a 1-D model that takes a scalar input $x \in \mathbb{R}$ and a scalar weight $w_{\text{ridge}} \in \mathbb{R}$,

$$x' \leftarrow x - \epsilon \cdot \text{grad}_x \text{classifier}(x; w_{\text{ridge}}).$$

Derive the analytical closed form of ϵ as a function of the weight decay coefficient λ such that $\text{classifier}(x') = 0$. Show your work. Does weight decay make the model more robust under FGM attack? Why?

`\textbf{solution: }`

🔒 👍 🗨

To derive the closed form of ϵ , we start by writing out the expression for the adversarial example using the update equation:

$$\begin{aligned} x' &= x - \epsilon \cdot \text{grad}_x \text{classifier}(x; w_{\text{ridge}}) \\ &= x - \epsilon \cdot \text{grad}_x (xw_{\text{ridge}} - |xw_{\text{ridge}}| + \lambda w_{\text{ridge}}^2) \\ &= x - \epsilon (w_{\text{ridge}} - 2\lambda w_{\text{ridge}}) \\ &= x - \epsilon (1 + 2\lambda) w_{\text{ridge}}. \end{aligned}$$

Now, we want to find the value of ϵ that satisfies $\text{classifier}(x') = 0$. Using the expression for x' and the definition of the linear classifier, we have:

$$\begin{aligned} \text{classifier}(x') &= (x - \epsilon (1 + 2\lambda) w_{\text{ridge}}) w_{\text{ridge}} \\ &= x w_{\text{ridge}} - \epsilon (1 + 2\lambda) w_{\text{ridge}}^2. \end{aligned}$$


Setting this expression to zero and solving for ϵ , we get:

$$\begin{aligned} x w_{\text{ridge}} - \epsilon (1 + 2\lambda) w_{\text{ridge}}^2 &= 0 \\ \epsilon &= \frac{x w_{\text{ridge}}}{(1 + 2\lambda) w_{\text{ridge}}^2}. \end{aligned}$$

This is the closed form expression for ϵ as a function of the weight decay coefficient λ that satisfies $\text{classifier}(x') = 0$.

Now, let us consider the impact of weight decay on the adversarial robustness of the model. From the expression for ϵ , we see that the larger the value of λ , the smaller the value of ϵ for a given input x and weight w_{ridge} . This means that weight decay is making the model more robust to adversarial attacks under FGM, as it requires a smaller perturbation to fool the model. Intuitively, this is because weight decay encourages the model to have smaller weights, which in turn makes it less sensitive to small changes in the input. However, it is important to note that weight decay is not a silver bullet for adversarial robustness, and other techniques such as adversarial training may also be necessary to achieve robustness in practice.

Figure 15: Question 1.2.3

YO **Batch size vs. learning rate** 

Batch size affects the stochasticity in optimization, and therefore affects the choice of learning rate. We demonstrate this via a simple model called the noisy quadratic model (NQM). Despite the simplicity, the NQM captures many essential features in realistic neural network training—[cite\(zhang2019algorithmic\)](#).

For simplicity, we only consider the scalar version of the NQM. We have the quadratic loss $\mathcal{L}(w) = \frac{1}{2} a w^2$, where $a > 0$ and $w \in \mathbb{R}$ is the weight that we would like to optimize. Assume that we only have access to a noisy version of the gradient --- each time when we make a query for the gradient, we obtain $g(w)$, which is the true gradient $\nabla \mathcal{L}(w)$ with additive Gaussian noise:

$$g(w) = \nabla \mathcal{L}(w) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2).$$

One way to reduce noise in the gradient is to use minibatch training. Let B be the batch size, and denote the minibatch gradient as $g_B(w)$:

$$g_B(w) = \frac{1}{B} \sum_{i=1}^B g_i(w), \quad \text{where } g_i(w) = \nabla \mathcal{L}_i(w) + \epsilon_i, \quad \epsilon_i \stackrel{\text{i.i.d.}}{\sim} \mathcal{N}(0, \sigma^2).$$

As batch size increases, how do you expect the optimal learning rate to change? Briefly explain in 2-3 sentences.




 As the batch size increases, the variance of the minibatch gradient decreases, which implies that a larger learning rate can be used without the risk of overshooting the minimum. Therefore, as the batch size increases, the optimal learning rate tends to increase as well. However, the relationship between batch size and learning rate can also be affected by other factors, such as the curvature of the loss function and the specific optimization algorithm used.  

Figure 16: Question 2.1.1


YO \section{Dropout as Gaussian noise} 

In this question, we show that dropout can be equivalently viewed as applying Gaussian noise. As a warm-up, consider linear regression with input dropout of probability $1 - p$ (the input is retained with probability p).

$$\tilde{y}_m = \frac{1}{p} \sum_j m_j w_j x_j$$
 (where $m_j \sim \text{Ber}(p)$)

Derive the bias-variance decomposition. Show your work.

Solution:

 To derive the bias-variance decomposition for linear regression with input dropout, we first compute the expected prediction $\mathbb{E}[\tilde{y}_m]$:

$$\begin{aligned} \mathbb{E}[\tilde{y}_m] &= \mathbb{E}\left[\frac{1}{p} \sum_j m_j w_j x_j\right] \\ &= \frac{1}{p} \sum_j w_j x_j \mathbb{E}[m_j] \\ &= \frac{1}{p} \sum_j w_j x_j p \end{aligned}$$

Thus, the expected prediction is the same as in the standard linear regression without dropout.

Next, we compute the variance of the prediction:

$$\begin{aligned} \text{Var}[\tilde{y}_m] &= \text{Var}\left[\frac{1}{p} \sum_j m_j w_j x_j\right] \\ &= \frac{1}{p^2} \sum_j \text{Var}[m_j w_j x_j] \\ &= \frac{1}{p^2} \sum_j w_j^2 x_j^2 \text{Var}[m_j] \\ &= \frac{1-p}{p} \sum_j w_j^2 x_j^2 \end{aligned}$$

Finally, we can compute the bias by taking the difference between the expected prediction and the true output:

$$\begin{aligned} \text{Bias}^2 &= \left(\mathbb{E}[\tilde{y}_m] - y\right)^2 \\ &= \left(\sum_j w_j x_j - y\right)^2 \end{aligned}$$

Therefore, the bias-variance decomposition for linear regression with input dropout is:

$$\begin{aligned} \text{MSE} &= \text{Bias}^2 + \text{Var}[\tilde{y}_m] \\ &= \left(\sum_j w_j x_j - y\right)^2 + \frac{1-p}{p} \sum_j w_j^2 x_j^2 \end{aligned}$$

Figure 17: Question 3.1

Q3.2 `\section{Dropout as Gaussian noise}`

In this question, we show that dropout can be equivalently viewed as applying Gaussian noise. Instead of dropout, we apply the multiplicative Gaussian noise as follows:

$$y_{\mathit{math}(drop)^{(i)}} = \sum_{\mathit{math}(j)} (1 + \mathit{math}(p_{\mathit{math}(j)})) w_{\mathit{math}(j)} x_{\mathit{math}(j)}^{(i)}$$

where $\mathit{math}(p_{\mathit{math}(j)}) \stackrel{\text{i.i.d.}}{\sim} \text{Bernoulli}(1-p)$ are independent Bernoulli random variables that indicate whether the j -th input activation is dropped or not. At test time, the full network is used, and the input activations are scaled by $1-p$ to ensure that the expected input to each neuron is the same as during training.

To show that dropout can be viewed as applying Gaussian noise, we will compare the expected value and variance of the two operations. Let us start with the expected value. We have:

$$\mathit{mathbb{E}}[y_{\mathit{math}(drop)^{(i)}}] = \sum_{\mathit{math}(j)} (1 + \mathit{mathbb{E}}[\mathit{math}(p_{\mathit{math}(j)})]) w_{\mathit{math}(j)} x_{\mathit{math}(j)}^{(i)}$$

Here, we have used the fact that $\mathit{mathbb{E}}[\mathit{math}(p_{\mathit{math}(j)})] = 0$ for all j .

Next, let us compute the variance of $y_{\mathit{math}(drop)^{(i)}}$. We have:

$$\mathit{math}(Var)[y_{\mathit{math}(drop)^{(i)}}] = \mathit{math}(Var)[\sum_{\mathit{math}(j)} (1 + \mathit{math}(p_{\mathit{math}(j)}) w_{\mathit{math}(j)} x_{\mathit{math}(j)}^{(i)}]$$

$$= \sum_{\mathit{math}(j)} w_{\mathit{math}(j)}^2 x_{\mathit{math}(j)}^{(i)2} \mathit{math}(Var)[1 + \mathit{math}(p_{\mathit{math}(j)})]$$

$$= \sum_{\mathit{math}(j)} w_{\mathit{math}(j)}^2 x_{\mathit{math}(j)}^{(i)2} (\mathit{math}(Var)[\mathit{math}(p_{\mathit{math}(j)})] + \mathit{math}(Var)[1])$$

$$= \mathit{math}(sigma^2) \sum_{\mathit{math}(j)} w_{\mathit{math}(j)}^2 x_{\mathit{math}(j)}^{(i)2}$$

Here, we have used the fact that $\mathit{math}(p_{\mathit{math}(j)})$ and 1 are independent, and that $\mathit{math}(Var)[1] = 0$.

Now, let us compare these results to those obtained with dropout. The expected value of $y_{\mathit{math}(drop)^{(i)}}$ is:

$$\mathit{mathbb{E}}[y_{\mathit{math}(drop)^{(i)}}] = \sum_{\mathit{math}(j)} \mathit{mathbb{E}}[\mathit{math}(delta_{\mathit{math}(j)})] w_{\mathit{math}(j)} x_{\mathit{math}(j)}^{(i)}$$

Here, we have used the fact that $\mathit{mathbb{E}}[\mathit{math}(delta_{\mathit{math}(j)})] = 1-p$.

The variance of $y_{\mathit{math}(drop)^{(i)}}$ is slightly more complicated to compute, since the dropped activations contribute no variance. We have:

$$\mathit{math}(Var)[y_{\mathit{math}(drop)^{(i)}}] = \sum_{\mathit{math}(j)} w_{\mathit{math}(j)}^2 x_{\mathit{math}(j)}^{(i)2} \mathit{math}(Var)[\mathit{math}(delta_{\mathit{math}(j)})]$$

Let us first recall the formula for dropout in neural networks. During training, dropout randomly sets a fraction p of input activations to zero, so that the forward pass becomes:

$$y_{\mathit{math}(drop)^{(i)}} = \sum_{\mathit{math}(j)} \mathit{math}(delta_{\mathit{math}(j)}) w_{\mathit{math}(j)} x_{\mathit{math}(j)}^{(i)}$$

where $\mathit{math}(delta_{\mathit{math}(j)}) \stackrel{\text{i.i.d.}}{\sim} \text{Bernoulli}(1-p)$ are independent Bernoulli random variables that indicate whether the j -th input activation is dropped or not. At test time, the full network is used, and the input activations are scaled by $1-p$ to ensure that the expected input to each neuron is the same as during training.

To show that dropout can be viewed as applying Gaussian noise, we will compare the expected value and variance of the two operations. Let us start with the expected value. We have:

$$\mathit{mathbb{E}}[y_{\mathit{math}(drop)^{(i)}}] = \sum_{\mathit{math}(j)} (1 + \mathit{mathbb{E}}[\mathit{math}(p_{\mathit{math}(j)})]) w_{\mathit{math}(j)} x_{\mathit{math}(j)}^{(i)}$$

Here, we have used the fact that $\mathit{mathbb{E}}[\mathit{math}(p_{\mathit{math}(j)})] = 0$ for all j .

Next, let us compute the variance of $y_{\mathit{math}(drop)^{(i)}}$. We have:

$$\mathit{math}(Var)[y_{\mathit{math}(drop)^{(i)}}] = \mathit{math}(Var)[\sum_{\mathit{math}(j)} (1 + \mathit{math}(p_{\mathit{math}(j)}) w_{\mathit{math}(j)} x_{\mathit{math}(j)}^{(i)}]$$

$$= \sum_{\mathit{math}(j)} w_{\mathit{math}(j)}^2 x_{\mathit{math}(j)}^{(i)2} \mathit{math}(Var)[1 + \mathit{math}(p_{\mathit{math}(j)})]$$

$$= \sum_{\mathit{math}(j)} w_{\mathit{math}(j)}^2 x_{\mathit{math}(j)}^{(i)2} (\mathit{math}(Var)[\mathit{math}(p_{\mathit{math}(j)})] + \mathit{math}(Var)[1])$$

$$= \mathit{math}(sigma^2) \sum_{\mathit{math}(j)} w_{\mathit{math}(j)}^2 x_{\mathit{math}(j)}^{(i)2}$$

Here, we have used the fact that $\mathit{math}(p_{\mathit{math}(j)})$ and 1 are independent, and that $\mathit{math}(Var)[1] = 0$.

Now, let us compare these results to those obtained with dropout. The expected value of $y_{\mathit{math}(drop)^{(i)}}$ is:

$$\mathit{mathbb{E}}[y_{\mathit{math}(drop)^{(i)}}] = \sum_{\mathit{math}(j)} \mathit{mathbb{E}}[\mathit{math}(delta_{\mathit{math}(j)})] w_{\mathit{math}(j)} x_{\mathit{math}(j)}^{(i)}$$

Here, we have used the fact that $\mathit{mathbb{E}}[\mathit{math}(delta_{\mathit{math}(j)})] = 1-p$.

The variance of $y_{\mathit{math}(drop)^{(i)}}$ is slightly more complicated to compute, since the dropped activations contribute no variance. We have:

$$\mathit{math}(Var)[y_{\mathit{math}(drop)^{(i)}}] = \sum_{\mathit{math}(j)} w_{\mathit{math}(j)}^2 x_{\mathit{math}(j)}^{(i)2} \mathit{math}(Var)[\mathit{math}(delta_{\mathit{math}(j)})]$$

Q3.2 continue

To complete the computation of the variance of $y_{\mathit{math}(drop)^{(i)}}$, we need to compute the variance of the Bernoulli distribution. We have:

$$\mathit{math}(Var)[\mathit{math}(delta_{\mathit{math}(j)})] = \mathit{mathbb{E}}[\mathit{math}(delta_{\mathit{math}(j)})] - \mathit{mathbb{E}}[\mathit{math}(delta_{\mathit{math}(j)})]^2$$

$$= (1-p) - (1-p)^2$$

$$= p(1-p)$$

Substituting this result back into the variance expression, we obtain:

$$\mathit{math}(Var)[y_{\mathit{math}(drop)^{(i)}}] = \sum_{\mathit{math}(j)} w_{\mathit{math}(j)}^2 x_{\mathit{math}(j)}^{(i)2} \mathit{math}(Var)[\mathit{math}(delta_{\mathit{math}(j)})]$$

$$= p(1-p) \sum_{\mathit{math}(j)} w_{\mathit{math}(j)}^2 x_{\mathit{math}(j)}^{(i)2}$$

Comparing this result to the variance obtained with multiplicative Gaussian noise, we see that they match if we choose:

$$\mathit{math}(sigma^2) = \frac{p}{1-p}$$

Thus, applying input dropout with probability $1-p$ is equivalent to applying multiplicative Gaussian noise with variance $\mathit{math}(sigma^2) = p/(1-p)$. This relationship provides a different perspective on the effects of dropout: it can be viewed as adding noise to the inputs to prevent overfitting.

Figure 18: Question 3.2