Deep Learning on Computational Accelerators
236781 Mini-Project

Spring 2024

- **Submission Due**: 20/10/24, 23:59
- **TA In Charge**: Tamir Shor

1 Wet Assignment

In this project you will experiment with the task of parameter-space classification.
In section 1.1 we supply some basic background about parameter-space classification. You don't have to read it all to do the assignment, however we highly recommend going over this section before starting your work. In section 1.2 we describe what you'll be doing. Section 1.3 elaborates the code we'll be supplying you with for this project. Section 1.4 specifies how we'll evaluate your work.

1.1 Background

1.1.1 Intro. To Implicit Neural Representations

- Implicit Neural Representations (INRs) are the representation of a general signal (single data sample – e.g. image, point cloud, audio sequence) implicitly using a neural network $\varphi(x)$, rather than by explicitly storing all signal values. For example, for an MNIST image $img \in \mathbb{R}^{28\times 28}$, the **explicit** (standard) representation would be a tensor of size 28×28, where the value stored at location $i, j, 0 \leq i, j \leq 28$, is the pixel value for the image at the i, jth location on the grid. the **implicit** representation of the entire image would be a single neural network $varphi(x), x \in \mathbb{R}^2$ taking grid location coordinates i, j and predicting the corresponding pixel value - $img[i, j] = \varphi(i, j)$.

INRs are trained by sampling values from the explicit representation as "ground truth labels" at a given location. and optimizaing a neural network to predict those explicit pixel values. Rather than generalizing to unseen samples, The goal of $\varphi(x)$ is to strictly overfit itself to the input signal to best describe

it. We usually do not have a train-test split, as the downstream goal is not generalization to unseen data.

INRs can be used to represent any signal, not only images. For instance, an INR of a video image could be trained to predict entire time frames given a desired moment in time. An INR of an audio waveform could predict sound given a point in time.

A short, non-exhaustive list of advantages of using INRs is:

1. "Infinite" Resolution Sampling - At least in theory, INRs allow sampling the represented signal at an infinite resolution. In standard, explicit representation, the signal can only be represented using a finite set of coordinates. For example, for a 28×28 MNIST image we only have 784 pixel values. We cannot sample the image at non-integer grid locations, or at locations beyond the grid. INRs, on the other hand, can be queried at any given location. It should be noted that without adequate inductive bias, this may not generalize well.

2. Imposing Constraints - INRs basically treat our data samples as continuous functions. This makes it more easier to mathematically embed various desired constraints. For example, say we want to represent a noisy audio waveform in a close but cleaner manner. Additionally to imposing signal fidelity, we can also penalize the represented signal's derivative to clean noise.

3. Novel View Synthesis - INRs are widely used in the task of novel view synthesis - given several 2D images from a 3D scene, captured at different angles, generalize to how the scene would look like from a novel, unknown angle. A very common example would be NeRFs [1], who are in essence INRs.

1.1.2 Parameter-Space Classifiers

It has been shown [2] that a variety of common downstream tasks, such as generation, regression and classification, can be performed directly over INRs. In this project we will specifically explore classifiers that operate directly over INRs. Namely, given a dataset $\mathcal{X}$ of images annotated for classification over a set of classes $\mathcal{C}$ (say, MNIST annotated with the digit written for each image), we will first create a new dataset $\varphi_x \forall x \in \mathcal{X}$ containing by optimizing an INR network for every image in $\mathcal{X}$. Note INRs are inherently optimized for every data sample separately. Each $x \in \mathcal{X}$ would have its own separate set of parameters. At a second stage, given the dataset of INRs $\varphi_x \forall x \in \mathcal{X}$, we optimize a classifier model ψ. This classifier receives an INR φ_x, operates on it directly (meaning, using the actual set of weights and biases as input) and predicts a class $c \in \mathcal{C}$.

1.1.3 Functas

Training a parameter-space classifier is difficult both because we need to save an entire set of neural network weights for each data sample, which incurs a high memory burden, and because the data dimensionality is very large - even a small neural network could have thousands of parameters to be processed by the classifier ψ.
One solution is Functas [2] - this work essentially proposes to learn a dataset of SIREN INRs. SIRENs [3] are a specific type of MLP using sine functions as activations. You do not have to concern yourselves with what those are for the sake of the project, however it's a cool paper and we recommend you take a look at it if you find the field interesting.
The main contribution of Functas is how to create the INR dataset - for a given natural signal dataset by first learning a good initialization for all INRs, and then optimize for each sample for a small number of steps for the sample-specific network. Give this initialization, shared between all INRs, the Functa method offers coping with mentioned difficulties of scalability by only optimizing a smaller modulation vector of chosen dimensionality, that would condition the only biases for the original initialization.
If you're interested, you can refer to the original Functa paper [2] for the full details about this modulation vector. However, for the purpose of our project we just need you to understand that every INR ϕ_x (for every data sample) is governed by some vector of a fixed dimension of 512. These modulation vectors would represent the INRs in our INR dataset, and these vectors will be fed into the downstream classifier. These vectors can also be used to directly recreate the full INR ϕ_x, and thus to recreate the original image x. For every image $x \in \mathcal{X}$, we will denote the corresponding modulation vector by $v_x \in \mathbb{R}^{512}$.

1.2 Assignment

1.2.1 Clean Classification

Your assignment in this part will be to design and train a neural network to perform weight-space classification of a dataset of INRs (represented as modulation vectors $v_x \in \mathbb{R}^{512}$, as elaborated in section 1.1). Note that from here on out in this document we will use the terms "INRs" and "Modulation vectors" interchangeably. For this project, distinguishing between both terms is only important for the background section.
The dataset consists of 50000 optimized modulation vectors fitted for images from the Fashion-MNIST [4] dataset, split to training , validation and test sets. You will be supplied with this dataset as well as some code we've implemented for you (section 1.3).

1.2.2 Adversarial Attacks

In this section you will perform adversarial attacks over the classifier you trained in the previous section. Generally speaking, adversarial attacks optimize di-

rectly the input to an already trained model, in order to harm its performance as much as possible. In tutorial 4 we gave an example where we optimized an adversarial perturbation added to an MNIST image, to minimize the classification accuracy of a pre-trained classifier. Please refer to this tutorial to refresh the topic if needed.
As shown in the tutorial, a reasonable adversarial attack must not only fool the pre-trained classifier, but also be bounded somehow (otherwise, to make the MNIST classifier classify an image of the digit 5 as an image of the digit 7, we can just change the pixels so the input image looks like the digit 7, which isn't very impressive or meaningful).
In this section our attack bound will be defined as the $\mathcal{L}_\infty$ norm of the input modulation vector. This means we will make sure all entries in our perturbation vector do not exceed an absolute value of some ε of our choice. We can impose this by simple clipping of vector values (see provided code). Note bounding the modulation vector is not the most realistic constraint (it does not actually impose proximity to the original image), however this would be enough for the sake of this exercise.
As elaborated in section 1.3, some code for implementing the attack would be given to you. You will have to fill-in missing code.

1.2.3 Bonus Section - Adversarial Robustness

The following task is offered as a bonus section - it is purely optional, and those who choose to do it will receive a bonus of up to 10 points (depending on the quality of work).
One field of research that has been generating much interest in recent years is adversarial robustness - these studies aim to propose ways of training models so that they would be less prone to be harmed by adversarial attacks.
In this section you will need to propose, implement and evaluate a method to train your classifier in a more adversarially robust manner. See section 1.4 for more details.

1.3 Codebase

The codebase we provide consists of the following files:

- **classifier.py** - In this file you should implement and train your classifier for section 1.2.1 (you can also implement the classifier at a separate file and import). It contains examples on how to create the data-loaders, how to visualize an image based on a modulation vector and some other relevant comments. The code there is mainly used to give you some examples on how to manipulate the vector INR data - you can change this file as you see fit.
  Use the*data-path* parameter to pass path to functaset data (this is mounted on the Lambda servers for you at */datasets/functaset*, change it if needed).

- **attack.py** - In this file you should implement the adversarial attack from section 1.2.2. We gave you partial code for the attack_classifier function - you should complete the code as instructed and implement the main loop using the attack_classifier function. If you choose to do the bonus, you can implement it in this file as well, or in another file.
  Use the*data-path* parameter here similarly to *classifier.py*. You will also need to pass weights to your trained model (trained in *classifier.py*) via the *model-path* parameter.
  ‘

- **SIREN.py** - This file implements the INR architecture. We only need it for the image visualization function we supplied to you. You don't need to use any functions from this file yourself, or understand how it works.

- **utils.py** - This file contains some helper function we supplied to you - all of them are examplified in the two main scripts. You can also look at the documnetation on the file itself.

1.4 Evaluation

You will submit a report including the following components:

1. An overview of your solution - what classifier architectures did you consider? Which network did you eventually use? Why did you think it was a good choice? Detail your considerations. Any choice is okay as long as you reason about it. Also describe how you performed network parameter choices (e.g. depth, hidden dimension, number of kernels etc) and hyperparameter tuning (learn rates, batch sizes, dropout etc).

2. Report classification accuracy (percentage of data samples classified correctly) for the training, validation and test sets (report separately for each).

3. For each of the train, validation and test sets, plot and attach the confusion matrix for all classes. Each matrix of the three (one for each set) should be a 10×10 matrix, where the i, jth entry contains the number of samples labeled as class i but predicted by your classifier as class j (so a perfect classifier would produce a purely diagonal confusion matrix). Your plot should also include labels as for which class each column and row relates to.

4. Analyze your confusion matrices - what conclusions can you draw from them? If you had the chance to train again/finetune your model, what would you change? Relate to what weaknesses of your classifier the matrix reflects, and how you would combat them. Your answer should consider the roles of the 3 matrices.

5. Implement the *attack_classifier* function in the *attack.py* script (see section 1.3). Attach a screenshot of your fully-implemented function.

6. Run the adversarial attack over your classifier using the test set for 10 iterations per sample. Report classification accuracy for $\mathcal{L}_\infty$ bounds for every $\varepsilon \in \{10^{-6}, 5*10^{-6}, 10^{-5}, 5*10^{-5}, 10^{-4}, 5*10^{-4}, 10^{-3}\}$. You will need to adjust the learn rate to obtain optimal results (meaning achieve maximum reduction of classification accuracy). Plot the final received accuracy for different values of ϵ and add it to your report. Explain the trend - is the graph increasing, decreasing? Why is that the case?

7. Plot the confusion matrix (similar to that from section 3 above) for your classifier over the adversarial test set (i.e. the samples with optimized perturbations). For this evaluation choose a single $\mathcal{L}_\infty$ bound where classification accuracy after attack if between 20% and 80%. You should be able to find such a value within the given set of bounds. If not, try alternative bound values.

8. Analyze the received confusion matrix - What conclusions can be drawn from it? Describe how is it similar and how is it different from the previous case ("clean", non-attacked classifier).

9. Your report should also include some visualization of results - For the same $\mathcal{L}_\infty$ bound you picked in section 7, choose 3 classes, and for every chosen class c of the 3 pick two data samples - one where the attack succeeded (the ground truth label was c but the classifier predicted a different class) and one where it didn't (the classifier still classified the perturbed INR correctly as c). For each case (successful and failed attack, for each class) plot 3 images - The "clean", non-perturbed image (using the original modulation vector, before adding the optimized perturbation), the perturbation itself (generated from the perturbation vector you optimized) and the perturbed image (generated from adding the perturbation vector and original vector). To get an image from a modulation vector, use the provided *vec_to_img* function from the *utils.py* file.
   This means overall you should provide 3 classes $\cdot$ 2 (successful and failed attack) $\cdot$ 3 (clean, perturbation, clean+perturbation) = 18 plots.

10. **Bonus** (only do this section if you chose to do the bonus) -

    - Explain the approach you chose for training your classifier to be more adversarially robust. Specifically, explain how this method works, what's the motivation (why would it encourage robustness) and how did you apply it (did you implement any code, did you use code from another repository). In this section, you don't have to implement any of the solution yourself, however make sure to link relevant sources.
    - Train your classifier with your new approach, and then perform the adversarial attacks like you did for the previous section. Evaluate it under all $\mathcal{L}_\infty$ bound values you used before, and report "clean" (non-attacked) classification accuracy and classification accuracy after attack (for each bound in a graph plot). Also choose one class

and plot the clean image, perturbation and perturbed image for one example where the attack succeeded and one case where it didn't (similar to section 9, only here you only need to have a total of 6 images plotted).

- Compare the adversarial robustness of the classifier from previous sections to the one trained with your method. Did you manage to actually improve robustness (i.e. increase classification accuracy for perturbed modulation vectors)?
- : Note: solutions that are well-motivated and explained will receive partial credit even if they didn't actually achieve robustness improvement.

Some Important Notes

:

- You must implement a parameter-space classifier opting over the modulation vectors we gave you. Your solution is not allowed to explicitly transform the input vector into a "standard" image and operate on this image. Doing so would result in disqualifying your solution for this section. While we give you code to transform an INR into an image, you should use this code for visualization purposes only.
- You are encouraged to do some research (papers, blogs) to see how others solved the problem you are trying to solve and gain inspiration. That being said, we will not accept solutions completely copied from existing solutions, without any implementation on your part. You must implement your own model. Failing to do so would lead to disqualifying your solution. We also stress that the knowledge you gathered in this course alone is more than enough to solve this assignment and get perfect grade, without needing any external code or theory.
- We suggest you refrain from using overly large models for your classifier - this is not necessary for good results, and will prolong optimization and evaluation times in both sections. If any of the sections seem computationally heavy to you, reduce model size and use larger batch sizes.
- The set of perturbation $\mathcal{L}_\infty$ bounds we gave you to test your attacks should differentiate attack accuracy. This means that for lower and higher bounds within the set, you should notice a difference in attack success rate. If for some reason, for you model this is not the case (i.e. success rate is always very high or very low), you may also add lower and higher bounds to your set of test cases as needed (in such case, mention added test values in your report).
- All plots *must* include a title and a short explanation of what's in the plot. You must also include axis labels and grids for graph plots. Vague plots will incur grade penalty.

- For the bonus part - if you rely on solutions from the literature - cite relevant papers.
- Your report will be graded based on contents, but also based on presentation. Make sure to display your arguments in a clear, thorough and concise manner.
- Your report must address *all* requirements presented in the report description in this section.

2 Dry Assignment

In this section you will answer general theoretical questions concerning various topics we've seen in the course. Be sure to provide full and elaborate answers to all questions.

1. You want to use a 5-layer MLP to perform some regression task. Your friend claims that according to the Universal Approximation Theorem (UAT) for deep neural networks, any MLP with a single hidden layer can achieve optimal error over any dataset and loss criterion. He therefore concludes that there's never really a reason to use MLPs with more than one hidden layer.
   (a) Briefly explain what the UAT means, and how it supports the claim that you can achieve optimal error over any dataset and loss criteria.
   (b) Explain why his conclusion (never use more than one hidden layer) is wrong.
2. Alice and Bob want to perform image classification over a large, annotated dataset they found online. Alice suggests using an MLP, while Bob thinks it's better to use a CNN architecture.
   (a) Help Bob convince Alice - what is the main advantage of using a CNN over an MLP in this case? What are the main difficulties Alice might encounter of she uses an MLP?
   (b) Alice heard your statements from the previous section, but is still not convinced. She claims that the convolution operation is in any case linear. Therefore, there shouldn't be a difference between using linear layers between activations (MLP) or convolutional operations between activations (CNN). Do you agree with her? Explain.
3. You are trying to solve an optimization problem where the overall loss w.r.t learnable parameters is convex (i.e. there's a single minimum value). Would you expect using momentum would help the optimization process? Explain.

4. Based on what we learned in the automatic differentiation mechanism tutorial, explain why Pytorch demands the loss tensor to be a scalar in order to perfrom backpropagation (i.e. run *loss.backward()*).

5. You are given an image x of shape $1 \times 32 \times 32$. You are using the following convolutional layer to predict a ground-truth label y, which is an image of the same size as x. Given the convolutional layer and loss in the snippet, derive an analytical expression for the gradients of the loss w.r.t each learnable parameter in the model.
   Hint: Notice the loss term computation leverages the broadcasting mechanism.
   Hint 2: You don't have to understand how to strictly differentiate convolutional layers. Think about what the layer actually does in this specific case.

```
model = torch.nn.Conv2d(1,1,32)
for (x,y) in dataloader:
    #x - 1X32X32 shape sample, y - 1X32X32 shape ground truth
    y_hat = model(x)
    loss = ((y_hat-y)**2).sum()
```

6. Your friend wants to process some set of sequential data samples with an RNN model. She learned about positional embeddings in transformers and decided to apply them in RNNs as well.

   (a) Is it possible to add positional embeddings when using an RNN model? Explain.

   (b) If you think it is possible, explain why it would probably not be necessary in the case of RNNs. If you don't think it's possible, offer some adaptation to the RNN model we learned in class so that adding positional embeddings would be applicable.

7. Consider the following attention map, received during inference of a well-trained model performing English-French machine translation:

   Explain the received attention map:

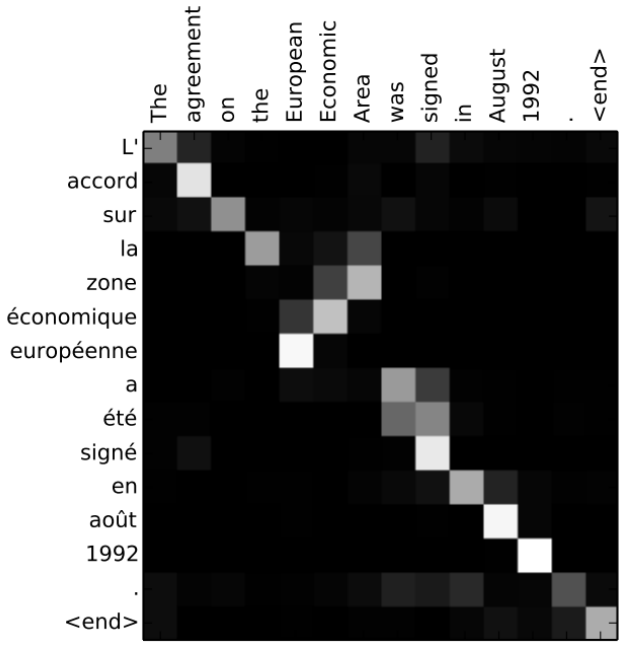
   (a) What does each pixel in the map represent?

   (b) What is the meaning of having rows with only one non-zero pixel?

   (c) What is the meaning of rows that have several non-zero pixels?

   (d) Explain why only rows with one non-zero pixel have a white pixels, while the rest have only gray pixels.

8. Explain what is the difference between the basic GAN we first presented in the GAN tutorial, and WGAN (Wasserstein-GAN) presented later on. What are the advantages of using WGAN over the basic GAN.



9. In the tutorial we've formulated:

$$\log p_\theta(x_0) = \mathbb{E}_q[\log(\frac{q(x_{1:T}|x_0)}{p_\theta(x_{1:T}|x_0)} \cdot \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)})] = D_{KL}(q(x_{1:T}|x_0)||p_\theta(x_{1:T}|x_0)) + \mathbb{E}_q[\log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}]$$

.
In training we drop the KL-divergence term, and only optimize the ELBO term $\mathbb{E}_q[\log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}]$.

   (a) What is the mathematical basis we rely on when we ignore the KL-divergence term?

   (b) Why can't we compute the KL-divergence term?

   (c) In the tutorial we further develop the ELBO term to

$$\mathcal{L}(\theta; x_0) = -D_{KL}(q(x_T|x_0)||p_\theta(x_T)) - \sum_{t>1} \mathbb{E}_q[D_{KL}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}] + \mathbb{E}_q[\log p_\theta(x_0|x_1)]$$

   . In training we ignore the term $-D_{KL}(q(x_T|x_0)||p_\theta(x_T))$. Explain why.

10. (a) Explain the terms "vanishing gradients" and "exploding gradients".

    (b) Provide a numerical example demonstrating each.

    (c) For each of the following architectures - MLP, CNN, RNN, offer one **distinct** way to combat vanishing or exploding gradients (do not use the same technique for two different architectures). Briefly explain how your technique helps reducing these phenomena.

3 Submission

Your submission must include a single zip file named $< id1 > - < id2 > .zip$, where id1,id2 are ids of you and your partner. This folder must include the following files:

- Your wet part's classifier.py and attack.py files. Also add your classifier's implementation if you did not implement it in classifier.py.
- If you chose to do the bonus, also submit your implementation of the robust training mechanism.
- A PDF file named *wet.pdf* containing your wet part report.
- A PDF file named *dry.pdf* containing your dry part report.

Good Luck!

References

[1] Ben Mildenhall et al. "Nerf: Representing scenes as neural radiance fields for view synthesis". In: *Communications of the ACM* 65.1 (2021), pp. 99–106.

[2] Emilien Dupont et al. "From data to functa: Your data point is a function and you can treat it like one". In: *arXiv preprint arXiv:2201.12204* (2022).

[3] Vincent Sitzmann et al. "Implicit neural representations with periodic activation functions". In: *Advances in neural information processing systems* 33 (2020), pp. 7462–7473.

[4] Han Xiao, Kashif Rasul, and Roland Vollgraf. "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms". In: *arXiv preprint arXiv:1708.07747* (2017).