CS4787/5777 Prelim Exam Solutions

Fall 2023

NAME:	
Net ID:	
Email:	

(Total points possible: 65)

You have ninety (90) minutes to complete the exam. Please make sure to write your NetID on each page of the exam.

No use of notes, computers, or mobile devices is allowed on this exam. Use of a calculator is allowed, but you are not expected to need one—all computations are designed to be doable by hand. This exam is subject to Cornell's Code of Academic Integrity: please sign this document below to indicate that you understand and commit to abide by the Code.

I promise to abide by Cornell's Code of Academic Integrity.

Signature: ______________________________________

1 [??: 11] True/False Questions

Please identify if these statements are either True or False. Please justify your answer **if false**. Correct "True" questions yield 1 point. Correct "False" questions yield 2 points, one for the answer and one for the justification. Note that a justification that merely states the logical negation of the statement will not be considered as a valid justification.

1. **(T/F)** Consider the vector space $V = \{(x, y) \mid x \in \mathbb{R}^p, y \in \mathbb{R}^q\}$ equipped with addition operator $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$ and scalar multiplication operator given by $c \odot (x, y) = (cx, cy)$. This vector space has dimension pq.

   F. It has dimension $p + q$.

2. **(T/F)** Adding ℓ_2 regularization to a convex loss function makes the total loss function strongly convex.

   T.

3. **(T/F)** Automatic differentiation, numeric differentiation, and symbolic differentation are all types of backpropagation.

   F. No, backpropagation is a type of automatic differentiation; the others are different things.

4. **(T/F)** Two tensors with shapes $(6, 6, 5, 1)$ and $(2, 6, 1, 2)$, respectively, can be broadcast together. (That is, if we tried to add them in numpy or torch, it would not cause a shape error.)

   F. They don't match in the 0th index since $6 \neq 2$.

5. **(T/F)** For $p \in \mathbb{N}$ and $d \in \mathbb{N}$, the function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ given by $k(x_1, x_2) = (1 + x_1^T x_2)^p$ is a kernel.

   T.

6. **(T/F)** Some transformers use positional encoders to preserve information about the order of an input sequence.

   T.

7. **(T/F)** We use the term “hyper-hyperparameter” to refer to a hyperparameter copied from a published paper.

> F. A hyper-hyperparameter is a parameter that controls a hyperparameter optimization algorithm, such as the number of points used in random search.

2 [14] Short Answers

1. (4 pts) Some concepts in machine learning have multiple terms that mean the same thing. Among the following list of terms, identify **three (3) pairs** of synonyms—terms which either mean the same thing as each other or which describe two very similar concepts.

learning rate	momentum
RNN	residual neural network
ℓ_1 regularization	ℓ_2 regularization
1-dimensional tensor	2-dimensional tensor
step size	matrix
weights	parameters

> Learning rate—step size. 2d tensor—matrix. Weights—parameters.

2. (2 pts) Suppose that we run minibatch Adam on a linear model with a batch size of $B = 100$ on a training dataset with $n = 5000$ examples, each of which lies in $\mathbb{R}^d$ for $d = 20$. Suppose that the loss function has condition number $\kappa = 10$. How many steps of minibatch Adam would there be in 1 epoch for this task? Be sure to show your work.

> One epoch is one pass through the data, so the number of iterations would be
> $$\frac{n}{B} = \frac{5000}{100} = 50.$$

3. (2 pts) Suppose that you want to run grid search on the learning rate α, momentum β, and regularization λ paramters of a machine learning model. Say that you want to search $\alpha \in \{0.01, 0.03, 0.1, 0.3, 1.0\}$, $\beta \in \{0.5, 0.9, 0.99, 0.999\}$, and $\lambda \in \{0.0001, 0.001, 0.01\}$. How many total training runs will be required to do this search? Be sure to show your work.

> $5 \times 4 \times 3 = 60.$

4. (6 pts) In class, we talked about kernel linear models of the form

$$f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(w^T \phi(x_i); y_i)$$

where n is the number of training examples, $\mathcal{L}$ is some loss function, and $x_i \in \mathbb{R}^d$, and y_i are our training examples and training labels, and $\phi : \mathbb{R}^d \to \mathbb{V}$ is some feature map that corresponds to a kernel $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ where $k(x, y) = \langle \phi(x), \phi(y) \rangle$, and $\mathbb{V}$ is the vector space the features lie in. Let D be the dimension of $\mathbb{V}$, and note that D could be ∞ i.e. ϕ maps to an infinite-dimensional Hilbert space.

We discussed six ways to compute SGD for this objective:

1. Transform the features on the fly and compute SGD on w.
2. Pre-compute and cache the transformed features, forming $z_i = \phi(x_i)$, and compute SGD on w.
3. Run a kernelized SGD and compute the kernel values $K(x_j, x_i)$ on the fly as they are needed.
4. Run a kernelized SGD on and pre-compute the kernel values for each pair of examples, forming the Gram matrix, store it in memory, and then use this during training as needed.
5. Pre-compute an approximate feature map ψ (with a smaller dimension than D) and use it to compute approximate features on the fly to compute SGD.
6. Pre-compute an approximate feature map, then also pre-compute and cache the transformed approximate features, forming vectors $z_i = \psi(x_i)$. Then run SGD.

For each of the following scenarios, **fill in** the circle associated with the **best** way of computing SGD in this scenario. Unless otherwise indicated, suppose that we will run SGD for a large number of iterations, such that the extra computational cost of any pre-computation will be effectively amortized across all the iterations of SGD. Also, **cross out** ($\times$) any circles associated with ways that are **computationally infeasible** to run on an ordinary desktop CPU in the described scenario. Briefly explain your answer.

(a) The feature map is finite-dimensional and has $D = 40$, and is given by $\phi(x) = \mathrm{ReLU}(Wx)$ for some fixed matrix $W \in \mathbb{R}^{D \times d}$. The dimension of the examples $d = 20000$ is relatively large, while the number of examples $n = 10^7$ is also quite large. (No approximate map is available.)

1 2 3 4

Since D is small, it would be best to run (2), since transforming the features first allows us to work purely in the D-dimensional feature space without having to process the much larger d-dimensional training examples x_i at each step. $n^2 = 10^{14}$ which would correspond to about 4TB of memory to store the Gram matrix, so (4) is infeasible. Also, since D is small, we won't expect to see much benefit from computing an approximate feature map.

(b) The kernel function is the RBF kernel $K(x, y) = \exp(-\gamma \cdot \|x - y\|^2)$. The dimension of the examples $d = 32$ is relatively small, while the number of examples $n = 10^7$ is relatively large. A good approximate feature map $\psi : \mathbb{R}^d \to \mathbb{R}^{D_{\text{approx}}}$ could be computed with dimension $D_{\text{approx}} = 20000$.

1 2 3 4 5 6

Since $D = \infty$, (1) and (2) are infeasible in this scenario. (4) will also be infeasible because the Gram matrix will be too large. Finally, (6) would be infeasible because we would need to store transformed approximate features of size $10^7 \cdot 2 \cdot 10^4 = 2 \cdot 10^{10} \approx 400$ GB which is infeasible to fit in memory on an ordinary CPU. The best remaining choice is (5).

3 [10] Reasoning About Scaling

1. (10 pts) Many of the techniques we discussed in class are designed to improve over a previous algorithm in the case where n (the training set size), d (the model size), and/or κ (the conditon number) is large—where "improve" is to understood in the sense of having an overall computational cost that has a better asymptotic dependence on these parameters. For each of the following methods, (1) briefly explain what the method does, (2) circle which, **if any**, of these parameters it is designed to address, and (3) briefly explain why it works to address that parameter.

   (a) Stochastic gradient descent (as compared with gradient descent) (n) (d) (κ)

   > (n) Uses only one training example in each iteration as opposed to all n.

   (b) Momentum (as compared with gradient descent) (n) (d) (κ)

   > (κ) Helps with converging faster, at a $1/\sqrt{\kappa}$ rate rather than a $1/\kappa$ rate.

   (c) Preconditioning (n) (d) (κ)

   > (κ) Helps with adjusting the curvature of certain dimensions that make it more difficult to converge.

   (d) Adam (as compared with gradient descent) (n) (d) (κ)

   > (n, κ) Combines the benefits of SGD, preconditioning, and momentum.

   (e) Random projection (n) (d) (κ)

   > (d) Reduces the dimensionality of the problem.

4 [19] Loss Functions and Backpropagation

Your classmate Lydia wants to use the square loss with L1 regularization to train an regressor. Given an input dataset $X \in \mathbb{R}^{n \times d}$ with labels $Y \in \{-1, 1\}^n$, Lydia's loss function is given by

$$\ell(w) = \frac{1}{n} \sum_{i=1}^{n} (y_i - x_i^T w)^2 + \lambda \|w\|_1$$

for $w \in \mathbb{R}^d$ and $\lambda \geq 0$, where y_i denotes the ith entry of Y and x_i denotes the ith example (the ith row of X), and where $\|w\|_1 = \sum_{i=1}^{n} |w_i|$ is the ℓ_1 norm. (You may find it useful to recall that the derivative of the absolute value function is the *sign* function, and it can be computed in PyTorch with `w.sign()`.)

1. (3 pts) Derive an expression for the gradient of Lydia's loss function.

$$\nabla \ell(w) = \frac{2}{n} \sum_{i=1}^{n} x_i (x_i^T w - y_i) + \lambda \operatorname{sign}(w)$$

where the sign function operates elementwise.

2. (3 pts) Lydia wants to test her code on a simple task where $\lambda = 0$ (i.e. no regularization) and where her dataset has size $n = 2$ and $d = 2$ with $x_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $x_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $y_1 = 2$ and $y_2 = -4$. In this simple setting, Lydia's loss function is L-smooth and strongly convex. What is its condition number κ?

The Hessian is just $\frac{2}{n} \sum_{i=1}^{n} x_i x_i^T$ which is I, so all its eigenvalues are 1 and the condition number is 1.

3. (4 pts) Lydia implements her loss function in PyTorch as follows.

```python
# w, X, Y all PyTorch tensors
def lydia_loss(w, X, Y, lamb):
    Xw = X @ w
    Xw_minus_Y = Xw - Y
    Xw_minus_Y_squared = Xw_minus_Y * Xw_minus_Y
    # tensor.mean computes the average of a vector
    square_loss = Xw_minus_Y_squared.mean()
    w_abs = w.abs()
    l1_w = w_abs.sum()
    lambda_l1_w = lamb * l1_w
    loss = square_loss + lambda_l1_w
    return loss
```

Suppose that each basic floating point computation ($+$, $-$, $\times$, $/$), comparison ($=$, $<$), and function computation (ReLU on a scalar) counts as one floating point operation (but assignments and integer operations are not counted). Also suppose that summing a vector of length n takes n additions, and a (m, n) by (n, p) matrix multiply takes $2mnp$ operations. As an expression in d and n, what is the **total** number of floating point operations run by Lydia's loss function? (Show your work.)

We can count them as follows:

- line 3: $n \times d$ matrix multiply with $2nd$ ops
- line 4: elementwise subtract of an n-dimensional vector, n ops
- line 5: elementwise product, n ops
- line 9: mean computation, $n + 1$ ops (n for sum, $+1$ for division by n)
- line 7: elementwise function computation, d ops
- line 11: sum, d ops
- lines 12+13: scalar ops, 1 op each

Total is

$$2nd + n + n + (n + 1) + d + d + 1 + 1 = 2nd + 3n + 2d + 3.$$

4. (9 pts) Now Lydia uses backpropagation to the gradient of her function with respect to w. The following pseudocode illustrates the computations that PyTorch could do behind the scenes. (Really, it illustrates the computations that your PA1 implementation would do.)

```
# inputs: same as before
def lydia_loss_grad(w, X, Y, lamb):
    (n,d) = X.shape
    w_grad = torch.zeros_like(w)
    Xw = X @ w
    Xw_grad = torch.zeros_like(Xw) # zeros of same shape as Xw
    Xw_minus_Y = Xw - Y
    Xw_minus_Y_grad = torch.zeros_like(Xw_minus_Y)
    Xw_minus_Y_squared = Xw_minus_Y * Xw_minus_Y
    Xw_minus_Y_squared_grad = torch.zeros_like(Xw_minus_Y_squared)
    square_loss = Xw_minus_Y_squared.mean()
    square_loss_grad = torch.zeros_like(square_loss)
    w_abs = w.abs()
    w_abs_grad = torch.zeros_like(w_abs)
    l1_w = w_abs.sum()
    l1_w_grad = torch.zeros_like(l1_w)
    lambda_l1_w = lamb * l1_w
    lambda_l1_w_grad = torch.zeros_like(lambda_l1_w)
    loss = square_loss + lambda_l1_w

    # backward pass

    loss_grad = ________________________________

    square_loss_grad += loss_grad

    lambda_l1_w_grad += ________________________________

    l1_w_grad += ________________________________

    w_abs_grad += ________________________________

    w_grad += ________________________________

    Xw_minus_Y_squared_grad += ________________________________

    Xw_minus_Y_grad += ________________________________

    Xw_minus_Y_grad += ________________________________ # hint: same as previous line

    Xw_grad += ________________________________

    w_grad += ________________________________

    return w_grad
```

Fill in the missing blanks in Lydia's code to show what backprop would compute. Note that some of the steps are already filled for you.

```python
# inputs: same as before
def lydia_loss_grad(w, X, Y, lamb):
    (n,d) = X.shape
    w_grad = torch.zeros_like(w)
    Xw = X @ w
    Xw_grad = torch.zeros_like(Xw) # zeros of same shape as Xw
    Xw_minus_Y = Xw - Y
    Xw_minus_Y_grad = torch.zeros_like(Xw_minus_Y)
    Xw_minus_Y_squared = Xw_minus_Y * Xw_minus_Y
    Xw_minus_Y_squared_grad = torch.zeros_like(Xw_minus_Y_squared)
    square_loss = Xw_minus_Y_squared.mean()
    square_loss_grad = torch.zeros_like(square_loss)
    w_abs = w.abs()
    w_abs_grad = torch.zeros_like(w_abs)
    l1_w = w_abs.sum()
    l1_w_grad = torch.zeros_like(l1_w)
    lambda_l1_w = lamb * l1_w
    lambda_l1_w_grad = torch.zeros_like(lambda_l1_w)
    loss = square_loss + lambda_l1_w

    # backward pass
    loss_grad = 1
    square_loss_grad += loss_grad
    lambda_l1_w_grad += loss_grad
    l1_w_grad += lamb * lambda_l1_w_grad
    w_abs_grad += l1_w_grad # broadcast
    w_grad += w_abs_grad * w.sign()
    Xw_minus_Y_squared_grad += square_loss_grad / n # broadcast
    Xw_minus_Y_grad += Xw_minus_Y_squared_grad * Xw_minus_Y
    Xw_minus_Y_grad += Xw_minus_Y * Xw_minus_Y_squared_grad # hint: same as previous line
    Xw_grad += Xw_minus_Y_grad
    w_grad += X.T @ Xw_grad

    return w_grad
```

5 [11] Deep Learning and Transformers

1. (3 pts) Your friend Alan uses PyTorch to construct a simple multilayer perceptron for a 10-class classification problem as follows:

```
model = torch.nn.Sequential(
    torch.nn.Linear(in_features=100,out_features=200,bias=False),
    torch.nn.ReLU(),
    torch.nn.Linear(in_features=200,out_features=400,bias=False),
    torch.nn.ReLU(),
    torch.nn.Linear(in_features=400,out_features=10,bias=False))
```

Alan uses 32-bit floating point numbers to store the weights of his network. How much memory in bytes is needed to store all the weights?

> 416000

2. (2 pts) Alan's roommate Lysanderoth suggests that Alan can reduce the number of floating point operations used in his neural network by removing the ReLU layers, thereby significantly improving its speed. Assuming that the number of floating point operations run is a good proxy for the network's speed, is Lysanderoth correct that this would greatly improve the speed of the network? Explain.

> No, the number of FLOps used in the ReLU, which are linear in layer width, is negligible compared with the matrix multiplies. which are quadratic in layer width.

3. (3 pts) Alan tries Lysanderoth's suggestion, but after retraining his model with the new architecture, Alan observes a significant decrease in both the training and validation accuracy of his classifier. Explain why this happened.

> Removing the ReLU layer makes the whole network linear and reduces it to a linear model. So, it likely does not have the expressive capacity to learn Alan's dataset.

4. (3 pts) Alan now tries to switch to a Transformer architecture. He proposes to implement the attention function as follows: if d is the hidden dimension (same for keys and values, i.e. $d_k = d_v$), and n is the number of tokens, then for input matrices $Q \in \mathbb{R}^{n\times d}$, $K \in \mathbb{R}^{n\times d}$ and $V \in \mathbb{R}^{n\times d}$,

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

Unfortunately, Alan observes strange behavior out of his transformer. What is wrong with Alan's attention layer? How should he fix it?

> Alan is multiplying the value matrix on the *right* (like a regular linear layer) when he should be multiplying it on the left (to mix among tokens) To fix, it should do
>
> $$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V.$$