Homework #6 — MTH 602

Assigned: Wednesday, Nov 5, 2025
Due: Monday, Nov 17, 2025 (print off and submit your report in class if possible)

This is a *hard deadline*! There's an exam on the 19th, so I will need to upload the solutions to this homework on Monday (Nov 17th) promptly at 5pm

Hardcopy reports should be submitted to Scott by Monday, Nov 17, 2025 at 5pm. If you cannot make it to class, you must find some other way to submit the report. **You can work in groups of 1, 2, or 3.**

Report: Put all the figures and answers asked for in the questions into a single, well-organized PDF document (prepared using LaTeX, a Jupyter notebook, or something else). **Please submit your report by uploading it to your git project and printing it off to hand in.** Ensure your submission is *reproducible*: I should be able to clone your repository and generate your reported results by using available code. Paper and pencil questions can be included in the PDF report or turned in separately

I. PAPER & PENCIL WORK (20 POINTS)

1. (10 points) Consider a two-layer network of the form given by Eq. 6.11 in the book.
   - How many learnable parameters are in this model? Your answer should be an expression that depends on the input vector size D as well as the number of hidden units M.
   - Consider $D = 1$, $M = 2$, and a ground-truth function you are approximating $f(x) = 2x$ (i.e., multiplication by 2). Suppose you use a ReLU activation function. Find two neural networks (one hidden layer with $M = 2$ neurons) with the same architecture but different weights that exactly match this target function. From this, you should conclude that (i) even simple operations like multiplication lead to nontrivial networks and (ii) solutions are not unique.
   - Book problem 6.4
2. (10 points) When numerically solving an optimization problem we (i) define a cost function $E(\vec{\omega})$, and (ii) ask the optimizer to find a vector $\vec{\omega}$ satisfying $\nabla_{\vec{\omega}} E(\vec{\omega}) = 0$. In this problem, you will explore when $\nabla_{\vec{\omega}} E(\vec{\omega}) = 0$ is a true minimum.
   - Book problem 7.3
   - Book problem 7.4
   - From the above you should conclude that (i) a minimum is found if and only if the Hessian matrix is positive definite and (ii) for linear regression, the Hessian is positive definite everywhere (even away from the minimum), hence checking $\nabla_{\vec{\omega}} E(\vec{\omega}) = 0$ is sufficient.

II. COMPUTING REQUIREMENTS

The problems below will likely benefit from Unity or Colab. While you won't need a GPU (you are welcome to use one), some of these problems might need to run for a while. Everyone has access to Unity, which is the default/assumed option. The class GitHub page provides instructions for using Unity. And remember: never run your code on the login node!

III. A NOTE ON JAX AND OPTAX

The homework assumes you will be using the Python packages **JAX** with **Optax**; however, you are welcome to use any other Python library or Julia to solve this problem. From their own websites:

- "JAX is a Python library for accelerator-oriented array computation and program transformation, designed for high-performance numerical computing and large-scale machine learning. It is developed by Google with contributions from Nvidia and other community contributors."
- "Optax is a gradient processing and optimization library for JAX. It is designed to facilitate research by providing building blocks that can be recombined in custom ways in order to optimize parametric models such as, but not limited to, deep neural networks."

Again, you are welcome to use any other Python library or Julia for this problem. Whatever library you use, report on whether you are using 32 or 64 bit computations. In JAX, 64 bit (double) is enabled by doing "jax_enable_x64=True"

IV. RIDGE REGRESSION VIA SGD IN JAX (50 POINTS TOTAL)

A. Brief recap of the previous assignment for context

In Homework 4 you fit noisy data $\{(t_i, y_i)\}_{i=1}^N$ with regularized polynomial least squares by adding ℓ_2 (ridge) regularization. Given the Vandermonde design matrix $A \in \mathbb{R}^{N\times(M+1)}$ with columns $[1, t, t^2, \ldots, t^M]$, we define a loss function

$$J(\omega) = \|A\omega - y\|_2^2 + \lambda\|\Gamma\omega\|_2^2, \tag{1}$$

and the solution to the optimization problem, ω_{opt}, is found by solving

$$\min_{\omega\in\mathbb{R}^{M+1}} J(\omega),$$

where Γ is a matrix (typically either $\Gamma = \mathrm{diag}(0, 1, \ldots, 1)$ or $\Gamma = \mathrm{diag}(1, 1, \ldots, 1)$), $\lambda \geq 0$ is the regularization parameter, and $\omega \in \mathbb{R}^{M+1}$ is a vector of model weights. In this case, the solution is given in closed form by the normal equations,

$$\omega_{opt} = (A^\top A + \lambda\Gamma^\top\Gamma)^{-1}A^\top y, \tag{2}$$

which can be compared to Eq. (4.27) from the Bishop book. Homework 4 had you consider the problem's conditioning as well as the test/train error computed over the hyperparameter space (M, λ).

B. Why numerical optimization

As we move into deep learning, closed-form solutions are no longer possible. Instead, the relevant optimization problem must be solved using numerical algorithms implemented in existing code bases.

Before considering a deep learning example, this first problem will have you revisit regularized polynomial least squares but solved using optimization algorithms. The purpose is twofold: (i) gain experience with solving optimization problems numerically with well-tested software and (ii) better understand the behavior of optimizers. Regularized least squares is an ideal setting for this because we can always compare back to the optimal solution.

You will solve the regularized regression problem using *stochastic gradient descent (SGD)*.

C. Dataset and setup

Use the following data generator for this problem:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

def generate_data(scale_noise, N, seed=None):
    rng = np.random.default_rng(seed)
```

```
    t = np.linspace(0.0, 1.0, N)
    s = 1 + t + t*t
    n = rng.normal(loc=0.0, scale=scale_noise, size=N)
    y = s + n
    return t, y, s, n

t_all, y_all, s_all, n_all       = generate_data(0.0, 1000, seed=0)
t_train, t_test, y_train, y_test = train_test_split(t_all,y_all,test_size=0.2,random_state=42)

plt.figure()
plt.plot(t_train, y_train, 'bo', label='training data')
plt.plot(t_test, y_test, 'r+', label='testing data')
plt.legend(); plt.xlabel("t"); plt.ylabel("value"); plt.title("HW4 data + split")
plt.show()
```

D. Write and test the model and loss functions (10 POINTS)

Consider a fixed degree M and penalization parameter $\lambda = 0$. Now define the Vandermonde design matrix A through evaluation of the polynomial basis functions at the requested sample points $\vec{t}$. Write two JAX functions with the following input/output structure:

1. (**Model**). Input: vector of weights ($\vec{\omega}$) and the matrix (A). Output: model evaluation $A\vec{\omega}$.
2. (**Loss**). Input: vector of weights ($\vec{\omega}$), the matrix (A), and the data ($\vec{y}$) on which to compute the loss. Output: the value of the loss according to Eq. (1). Note: please set $\Gamma = \text{diag}(1, 1, \ldots, 1)$ when defining your loss function.
3. (**MSE**). With your loss function from above, you can also compute the mean squared error by setting $\lambda = 0$.

For regression problems, as before, it will be important to distinguish between the MSE (or loss) evaluated on the testing and training sets. For example, the MSE on the training data is:

$$\text{Training error: } \|\vec{y}_{\text{train}} - y_{\text{model}}(\vec{t}_{\text{train}})\|_2^2 = \frac{1}{N_{\text{train}}} \sum_{i=1}^{N_{\text{train}}} \left[y_i^{\text{train}} - y_{\text{model}}(t_i^{\text{train}})\right]^2 \tag{3}$$

And we can use the testing data to monitor the generalization error during the training process by computing:

$$\text{Generalization error: } \|\vec{y}_{\text{test}} - y_{\text{model}}(\vec{t}_{\text{test}})\|_2^2 = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \left[y_i^{\text{test}} - y_{\text{model}}(t_i^{\text{test}})\right]^2 \tag{4}$$

Report on the following: Please provide evidence that your model and loss function are working as expected. That is, you should design and report on a simple test (a sanity test) for this task. The point here is to build confidence in your functions' correctness.

E. Optimizing with stochastic gradient descent (10 POINTS)

In this part, you will use the SGD optimizer without momentum or acceleration. So the only optimizer-level parameters to set are the learning rate (α), the batch size (B), and the weight initialization values (ω_{init}). If you are using Optax, use the `sgd` optimizer. This one specifically:
https://optax.readthedocs.io/en/latest/api/optimizers.html#optax.sgd

Please do the following:

1. Write code that implements a training loop that (per epoch) logs train/test MSE.
   - Verify it's working by doing a simple test case. For example, set $M = 3$ (polynomial degree), $\lambda = 0$ (regularization), $\alpha = .2$ (learning rate), and $B = N_{\text{train}}$ (use the entire training set as the batch).

- Plot the training and testing loss vs. epoch. Use a large number of epochs (I ran up to 100,000)
- Plot the ℓ_2 norm measuring the distance from the optimizer's solution to the true one, $\|\omega_i - \omega_{opt}\|_2$, where ω_{opt} is the optimal value given by the closed-form solution, Eq. (2), and ω_i is the value of the weights (ω_i is a vector!) at the i^{th} epoch.
- Compare the final weights from the SGD optimizer against the closed-form solution.

2. Generalize what you did in part 1 to iterate over batches of the training set for each epoch. For this, it will be helpful for you to consult the Optax documentation: https://optax.readthedocs.io/en/latest/_collections/examples/README.html

F. Batch size study (10 POINTS)

Fix $M = 3$, $\lambda = 0$, and set the learning rate to $\alpha = .1$. Train with $B \in \{1, 16, 64, N_{\text{train}}\}$ for 100,000 epochs. For each different value of B plot the training and testing MSE vs. epoch. Also measure the typical time (using some measurement for timing the code like tic-toc) to a target test MSE. Discuss any interesting observations.

G. Learning-rate study (10 POINTS)

With $B = 32$ and the same (M, λ) as in the previous subsection, compare:

- A constant learning rate.
- Using momentum (e.g., $\beta = 0.9$) and/or Nesterov acceleration.
- Using a learning-rate scheduler like step decay or cosine decay. Note: you may need to tune the *initial* learning rate so the optimizer doesn't diverge over the initial phases of training.

Plot loss vs. epoch for the various cases listed above. Briefly comment on stability and performance. By stability we mean the optimizer doesn't diverge and the loss doesn't have large noisy "spikes" while training. By performance we mean a high-quality solution is obtained (loss is low, the optimizer is close to the true solution ω_{opt}) in as few epoch steps as possible.

H. SGD vs another optimizer (10 POINTS)

Optax provides a variety of optimization methods: https://optax.readthedocs.io/en/latest/api/optimizers.html

Pick another one of your choosing (popular choices are `adam`, `adagrad`, `adabelief`, and L-BFGS). Re-solve the optimization problem and compare stability and performance between this method and the best SGD variant you considered in the previous subsection.

I. Theoretical insights of SGD (Optional)

Reading: Smith et al., *Don't Decay the Learning Rate, Increase the Batch Size* (ICLR 2018); Smith & Le, *A Bayesian Perspective on Generalization and SGD* (2018).

Despite the common conception that the learning rate should decay as training proceeds, these works show that LR decay can often be replaced by increasing the batch size during training, guided by the SGD noise scale $g \approx \epsilon N/B$. Read/skim these papers for the details. If you have extra time (after completing the rest of this assignment!) you can come back to this part and compare (i) using an LR scheduler with (ii) a batch-growth scheduler whereby you fix initial learning rate ϵ_0 and *increase* the batch size each epoch to match the baseline's noise-scale decay. Please read the papers to figure out how to modify the batch size as training proceeds.

V. EMPIRICAL UNIVERSAL APPROXIMATION THEOREM (30 POINTS + BONUS)

A. Problem setup

The Universal Approximation Theorem states that a feed-forward network with a single hidden layer and a ReLU or `tanh` activation can approximate any continuous function on a compact domain arbitrarily well, given sufficient width. Furthermore, as is typical in approximation theory, we would also like to know the rate of convergence to the true function.

In this assignment you will verify the theorem and empirically measure the rate of convergence.

B. Data and target function

Let the target be the smooth function

$$f^\star(t) = e^{-3t}\sin(8\pi t) \qquad t \in [0,1].$$

Generate $N_{\text{train}} = 128$ training points and $N_{\text{test}} = 2048$ test points uniformly on $[0,1]$, unless otherwise stated. Use mean squared error (MSE) for training and for all error reports. Here's some starter code:

```
import numpy as np

def f_star(t):
    return np.exp(-3.0*t) * np.sin(8*np.pi*t)

rng = np.random.default_rng(0)
t_train = rng.uniform(0.0, 1.0, size=128)
y_train = f_star(t_train)
t_test = np.linspace(0.0, 1.0, 2048, endpoint=True)
y_test = f_star(t_test)

# TODO: You should also plot the function
```

C. A two-layer network model

Use a single hidden-layer network with activation ϕ:

$$\hat{f}(t;\theta) = \sum_{j=1}^{m} a_j \phi(w_j t + b_j) + c,$$

where $\theta = \{(a_j, w_j, b_j)_{j=1}^m, c\}$ are the learnable parameters and m is the hidden width. Train by minimizing the MSE on the training set with your optimizer of choice (`sgd`, `adam`, or anything else) from Optax.

Good initialization helps! Draw first-layer biases b_j roughly uniformly over $[-1,1]$ and use small output weights a_j and input weights w_j.

D. Implementation & visualization (10 POINTS)

Pick $m = 16$ and train your model until validation/test error plateaus (you may need as many epochs depending on optimizer/LR). Plot MSE vs. epoch. Also show the true $f^\star(t)$ and your learned $\hat{f}(t;\theta)$ over a dense grid of test points.

E. Empirical rate of convergence with layer width (10 POINTS)

Using a ReLU activation function, train separate models for widths

$$m \in \{2, 4, 8, 16, 32, 64, 128\}.$$

For each m, record the final test errors:

$$E_2(m) = \sqrt{\frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} (\hat{f}(t_i) - f^\star(t_i))^2}, \quad E_\infty(m) = \max_i |\hat{f}(t_i) - f^\star(t_i)|.$$

Given inherent randomness of the optimization, you might consider a few random restarts per m and report the median (or minimum) test errors to reduce optimizer-failure outliers.

For a two-layer ReLU network with m hidden units the error should decay like $E(m) \propto Cm^{-\alpha}$, where α is the rate of convergence. On a log–log plot of error vs. m, you should see a slope near α when optimization succeeds and regularization/noise effects are not problematic.

Make two log–log plots: $E_2(m)$ vs. m and $E_\infty(m)$ vs. m. Estimate the slope α in $E(m) \approx Cm^{-\alpha}$.

F. Effect of the activation function (10 POINTS)

Repeat the width sweep with a smooth activation (e.g., `tanh`). Compare the empirical rates to ReLU. Briefly speculate on why the curves differ (or don't).

Challenge bonus: Building compact networks (+10 POINTS)

Using the same 1D target function as given above, train a fully connected MLP to achieve a test MSE $\leq 10^{-4}$ with as few trainable parameters as possible. This is a short study in hyperparameter optimization (depth, width, activation, optimizer, learning rate/schedule, initialization, batch size, etc.).

Hyperparameters you may consider varying: the number of layers (depth), number of neurons per layer (width), activation function (`ReLU`, `tanh`, SiLU, GELU, etc.), and optimizer (SGD/momentum/Adam), LR scheduler, early stopping, and regularization in the cost function.

What to submit:

1. Clear evidence that the model achieves MSE $\leq 10^{-4}$ on a dense test set. Your figure or number reported should make it clear the MSE is below the threshold on the test set.
2. The number of trainable parameters for your model. If you have a JAX model, this can be obtained as follows:

```python
import jax
import jax.numpy as jnp

def count_params_pytree(params):
  leaves = jax.tree_util.tree_leaves(params) # Collect all leaf arrays from the nested params tree
  sizes = [int(jnp.size(x)) for x in leaves] # Get the size of each array
  total = sum(sizes) # Get the total number of trainable scalars
  return total
```

If you didn't use JAX, your library should still provide some way of getting the size of all model parameters that were learned during training.

Any model that achieves an MSE $\leq 10^{-4}$ qualifies. The winner is the model with the fewest trainable parameters.