S&DS 689 Homework 2

Jonas Katona

December 21, 2023

1 Problem 1

1.1 Introduction

Universal approximation theorems (and in particular, those with ReLU activation functions) suggest that even discontinuous functions such as L^p functions can be approximated using shallow feedforward neural networks. Considering how the ReLU activation function is only C^0 (i.e., it has discontinuous derivatives), this approximability could be expected heuristically speaking. In particular, a shallow ReLU network should be able to approximate $f(x)$ in this case:

$$f\left(x\right) = \begin{cases} 5 + \sum_{k=1}^{4} \sin\left(kx\right), & x \in \left[-\pi, 0\right) \\ \cos\left(10x\right), & x \in \left[0, \pi\right]. \end{cases} \tag{1}$$

As shown in Figure 1, $f(x)$ has a discontinuity at $x = 0$, although the function is still L^2 since f is defined over a bounded domain and only discontinuous at a single point.

For this problem, we used the scripts `1.py` (Section 1.3.1), `err_process.sh` (Appendix 7.1), and `plt_rel_errors.py` (Appendix 7.2). In the former, we run the shallow ReLU network with `N_data`$=200$ points for training (uniformly distributed on $[-\pi, \pi]$, `N_test`$=$ 689 points for testing (also uniformly distributed on $[-\pi, \pi]$), and 5×10^4 epochs for training. For optimizing the parameters of the network, we use the Adam optimizer with a learning rate of $\lambda = 10^{-3}$, and the implementation of both the neural network architecture and optimization/training was done using PyTorch and GPU acceleration via CUDA. These were done over the following network widths (as stated in the problem): 10, 30, 100, 300, 1000. We did not do any batch training as it was not necessary for the size of the problem.

`1.py` saves the plots in Figure 1 and several plots analogous to Figure 3 except for the hyperparameters in Problem 1 rather than Problem 2; we omit these for Problems 1 and 3 for brevity. `1.py` also returns the final MSE losses (for training) and L^2 relative errors (for training) for each width. For run `i` of the code, these results are saved into `1_run_$i.txt` for `i`$= 1, 2, 3$. Then, using `err_process.sh` (Appendix 7.1), these files are concatenated and then piped into `plt_rel_errors.py` (Appendix 7.2) to compute the means and standard deviations over the three runs and generate the plots shown in Figure 2. Both `err_process.sh` and `plt_rel_errors.py` have a few lines which need to be changed (related to file names) depending on if we are interested in generating plots for Problem 1, 2, or 3.

As for difficulties in the results, I noticed that, with the neural network width, the testing error in the solution was greater for $x > 0$ than for $x < 0$ when comparing against the ground truth. This suggests how, in practice, for problems with discontinuous solutions (and in particular, those with more than just a removable singularity), since the discontinuity could somewhat "interrupt" the properties of the solution, it might be advisable to train separate,

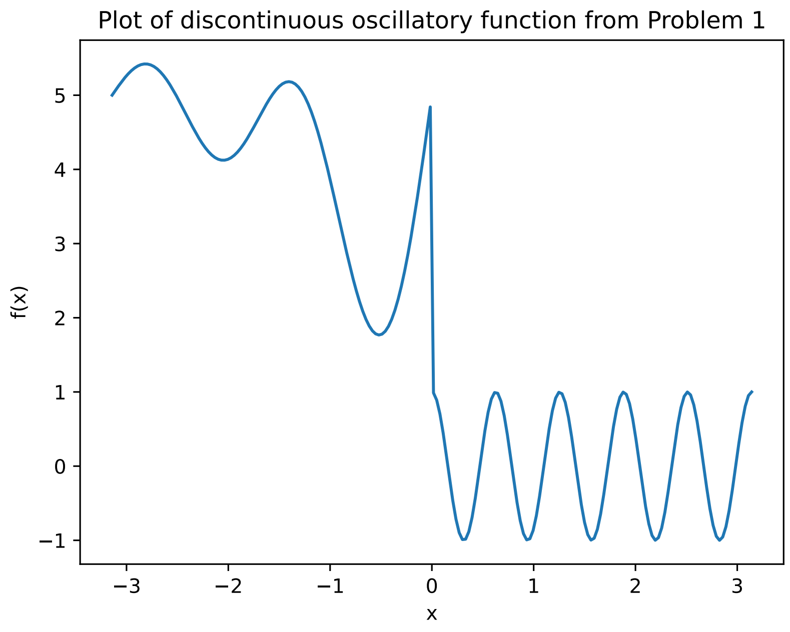


Figure 1: Plot of the discontinuous, oscillatory function $f(x)$ (eq. (1)) from Problem 1.

disconnected networks on each branch of the solution. In particular, in this case, we would want to train two separate feedforward neural networks on $[-\pi, 0)$ and $(0, \pi]$.

1.2 Results

From the three runs of the shallow ReLU network with the hyperparameters described in Figure 1.1, we can generate the statistics and trend shown in Figure 2. Assuming an L^2 error which converges via a power law with network width (which would look linear under log-log scaling), we plot the L^2 relative testing error (averaged across three runs) as a function of network width using both linear-linear and log-log scaling, along with error bars to show the standard deviations ($\pm$ 1 standard deviation). While we can observe roughly power law convergence for network widths tested from 30 to 300 (within some leeway of 1 and 2 standard deviations), the convergence flattens out afterwards as we increase the network width up to 1000. This is probably due to overfitting to the training data, since a network width of 1000 gives us far more network parameters to fit than the original number of input-output pairs, `N_data`= 200, used for training.

1.3 Code

1.3.1 1.py

`1.py` is the primary script used for running the training and testing of the shallow ReLU network from Problem 1. `1.py` was used in tandem with the two scripts found in the Appendix (Section 7) to generate the L^2 relative error plots (Figure 2).

```
import numpy as np

# generate the dataset
N_data = 200 # number of points for data
```

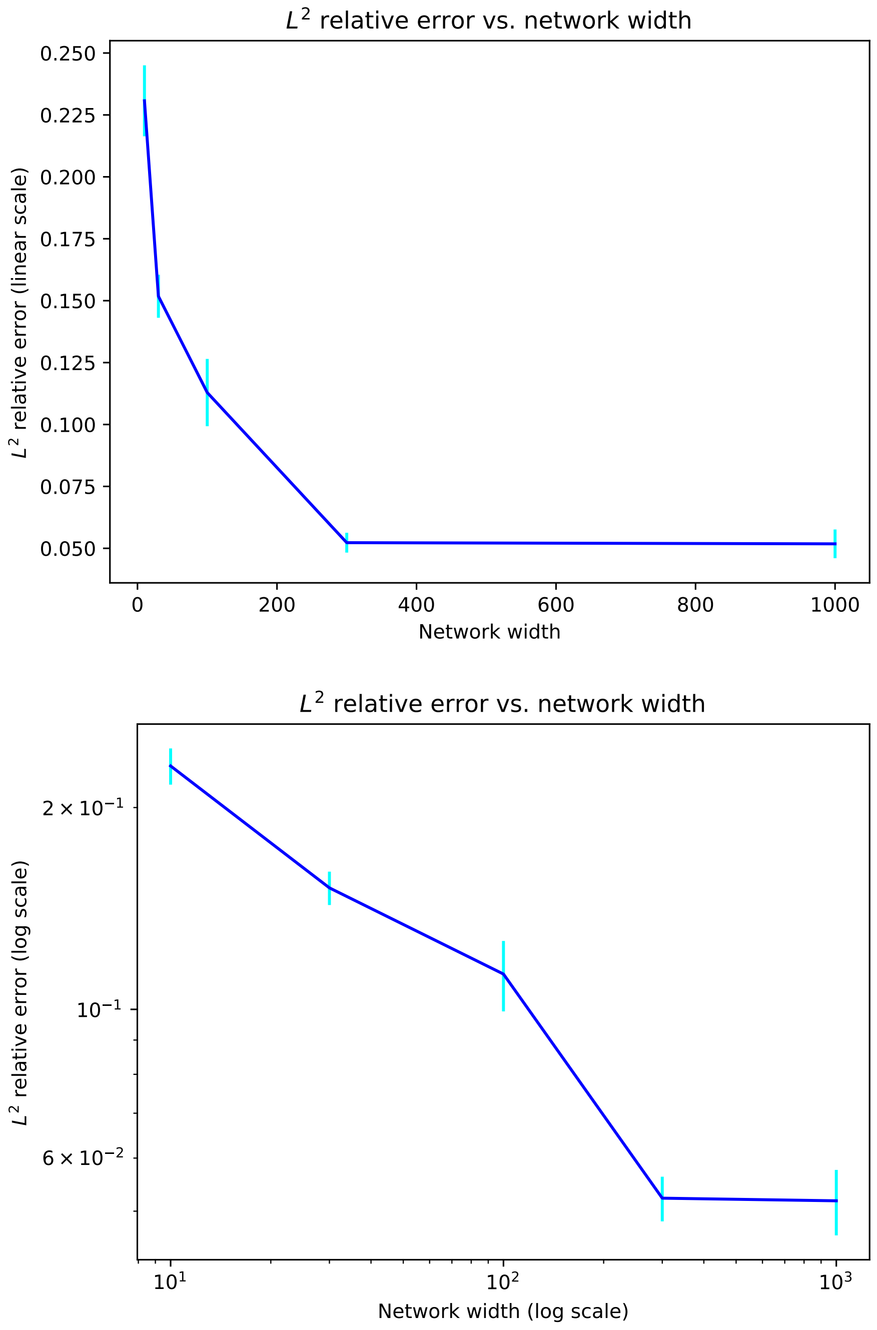


Figure 2: Convergence of L^2 relative error for testing on f (eq. (1)) with network width for the network architecture in Problem 1.

```
N_test = 689 # number of points for testing
k_tot = 4 # number of elements in sinusoidal sum
X = np.linspace(-np.pi,np.pi,N_data) # create grid ofN_data points in x

# define function to evaluate f on grid points in X
def func(xq):
    yq = np.zeros(len(xq))
    # separate X into segments below and above zero
    x_below = xq[xq < 0]
    x_above = xq[xq >= 0]
    yq[xq < 0] = 5
    for k in (np.arange(k_tot) + 1):
        yq[xq < 0] += np.sin(k * x_below)
    yq[xq >= 0] = np.cos(10 * x_above)
    return yq
y = func(X)

# define L2 relative error
def L2_rel_error(approx, actual):
    top = np.sum(np.square(approx - actual))
    bottom = np.sum(np.square(actual))
    return np.sqrt(top / bottom)

# plot function
import matplotlib.pyplot as plt
plt.plot(X, y)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Plot of discontinuous oscillatory function from Problem 1')
plt.savefig('1_plot.pdf')
plt.close()

# train neural network
import torch
import torch.nn as nn
import torch.optim as optim

if torch.cuda.is_available():
  print("CUDA available. Using GPU acceleration.")
  device = "cuda"
else:
  print("CUDA is NOT available. Using CPU for training.")
  device = "cpu"

# convert input and output data to PyTorch tensors
X = torch.tensor(X, dtype=torch.float32, device=device).reshape(-1, 1)
y = torch.tensor(y, dtype=torch.float32, device=device).reshape(-1, 1)

# testing domain
```

```
X_test = torch.tensor(np.linspace(-np.pi,np.pi,N_test), dtype=torch.float32,
                      device=device).reshape(-1, 1)

widths = [10, 30, 100, 300, 1000] # list of widths to test
loss_fn = nn.MSELoss()  # use MSE loss
n_epochs = 5e4 # number of epochs
lr = 0.001 # learning rate for Adam optimizer
for w in widths:
    # define shallow ReLU network architecture with width w
    model = nn.Sequential(
        nn.Linear(1, w),
        nn.ReLU(),
        nn.Linear(w, 1)).to(device)
    # use Adam optimizer with learning rate lr
    optimizer = optim.Adam(model.parameters(), lr=lr)
    for epoch in (np.arange(n_epochs) + 1):
        y_pred = model(X) # forward pass
        loss = loss_fn(y_pred, y) # compute loss
        optimizer.zero_grad() # zero out gradient
        loss.backward() # backpropagation
        optimizer.step() # adjust parameters
    print(f'Finished training with width {w}, latest loss {loss}')
    # testing
    y_pred = torch.flatten(model(X_test)).detach().cpu().numpy()
    X_testq = torch.flatten(X_test).cpu().numpy()
    y_actual = func(X_testq)
    plt.plot(X_testq, y_pred, label='Prediction')
    plt.plot(X_testq, y_actual, label='Actual')
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title('Prediction vs. actual from shallow ReLU network \n '+
            'Network width: '+str(w)+', Number of epochs: '+str(n_epochs))
    plt.savefig('1_testing_'+str(w)+'.pdf')
    plt.close()
    # return L2 relative error for width w
    print(f'L2 relative error for width {w}: {L2_rel_error(y_pred, y_actual)}')
```

2 Problem 2

2.1 Introduction

The setting of the problem is the same as described in Section 1.1 except we replace `1.py` with `2.py` (Section 2.3.1), change a few lines related to file names in `err_process.sh` and `plt_rel_errors.py`, and add an extra layer in our ReLU feedforward network. Along with omitting the generation of the plot shown in 1, `2.py` defines the same neural network as in `1.py` except with an extra hidden layer. Furthermore, unlike in Problem 1, we do not omit the plots of the predicted values of $f(x)$ vs. the ground truth (across 689 uniformly distributed points in $[-\pi, \pi]$) shown in Figure 3.

For run `i` of the code, these results are saved into `2_run_$i.txt` for `i`$= 1, 2, 3$ to be then used in `err_process.sh`.

2.2 Results

Before discussing the convergence in the relative L^2 testing error as a function of network width, we consider the qualitative characteristics of how the neural network approximation compares with the ground truth as we vary the network width. One can check visually that, at least for widths 10 and 30, the number of piecewise linear segments in the network prediction actually matches the width of the network. I would hypothesize that this is true for greater network widths as well in the case of two hidden layers, so at least in Problem 2. However, this is not true for the networks in Problem 1 or 3. This feature may have to do with the expressivity of the neural network in both the input and output spaces. Namely, even if the output space of the networks in all problems technically utilize the same number of ReLU outputs, adding an extra hidden layer allows the network to sufficiently express the relationship between the input and output spaces.

Relatedly, as Figure 4 demonstrates, for each network width (but higher depth, of course), the architecture from this problem features a lower L^2 relative testing error than for that in Problems 1 or 3. This is because the universal approximation theorems for neural networks suggest that a neural network becomes more expressive and can better approximate functions as its depth increases. This thereby allows the network in Problem 2 to decrease the approximation error vs. Problem 1, but not the estimation error, since the number of points used for training are the same.

Furthermore, as the network width increases in Problem 2, it appears that the overfitting is slightly better mitigated than in Problem 1, since the L^2 relative error still decreases slightly as the network width goes from 100 to 1000. It is not generally true that increasing the number of parameters will increase the test accuracy because of overfitting. However, it has also been observed in deep learning that increasing the number of model parameters into a highly overparameterized regime actually leads to a so-called "double descent" in the test accuracy, where we yet again get a decrease in the test error with model parameters after passing a certain point. I am not sure if that might explain the behavior we are observing here. A more intuitive explanation (in my view) is that a smaller network depth will generally lead to a greater degree of overfitting for the same network width, thereby highlighting the importance of using deep vs. shallow neural networks. After all, a shallow neural network is not too different in mathematical formulation from the classical interpolation regime, in which overfitting and other phenomena such as Runge's phenomenon are well-described and well-known. Nonetheless, these two are likely connected (albeit still not well-understood the literature); double descent is only visible in the training and testing of clearly deep networks such as ResNet-18.

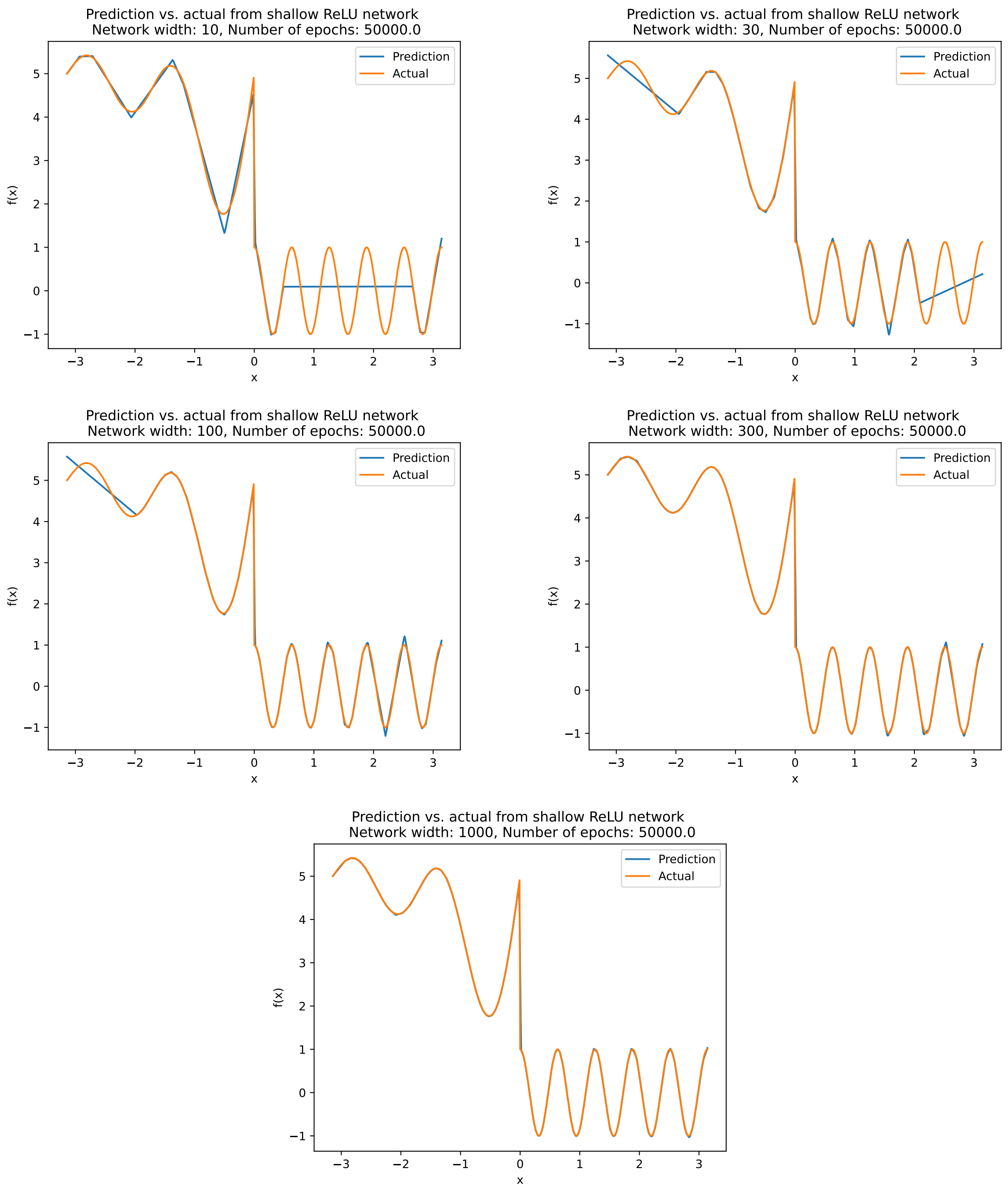


Figure 3: Values of $f(x)$ (eq. (1)) as predicted by the neural network trained in Problem 2 (blue) vs. the ground truth (orange). Evaluated at 689 points uniformly distributed on $[-\pi, \pi]$

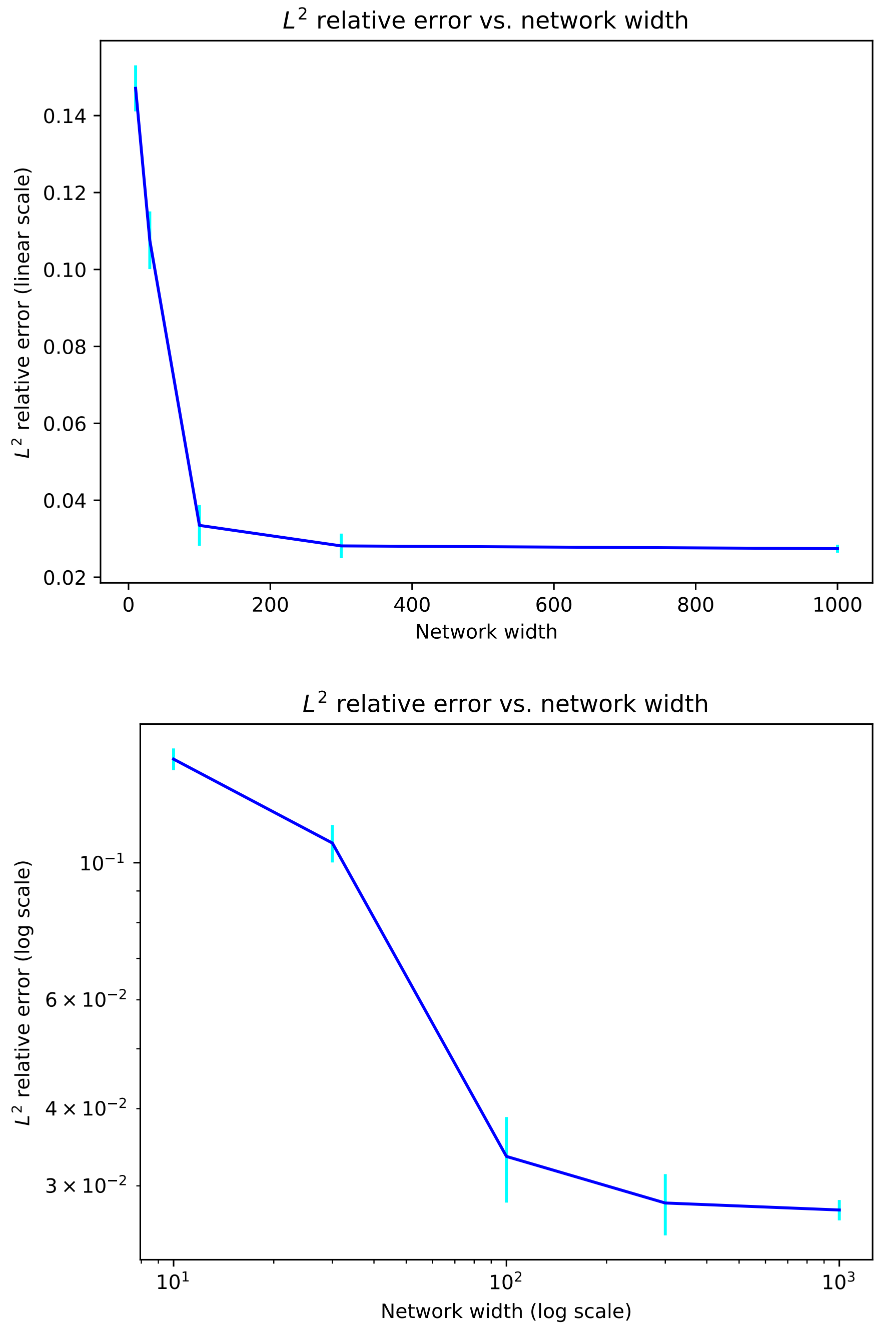


Figure 4: Convergence of L^2 relative error for testing on f (eq. (1)) with network width for the network architecture in Problem 2.

2.3 Code

2.3.1 2.py

2.py is the primary script used for running the training and testing of the ReLU FNN with two hidden layers from Problem 2. 2.py was used in tandem with the two scripts found in the Appendix (Section 7) to generate the L^2 relative error plots (Figure 4).

```
import numpy as np

# generate the dataset
N_data = 200 # number of points for data
N_test = 689 # number of points for testing
k_tot = 4 # number of elements in sinusoidal sum
X = np.linspace(-np.pi,np.pi,N_data) # create grid of N_data points in x

# define function to evaluate f on grid points in X
def func(xq):
    yq = np.zeros(len(xq))
    # separate X into segments below and above zero
    x_below = xq[xq < 0]
    x_above = xq[xq >= 0]
    yq[xq < 0] = 5
    for k in (np.arange(k_tot) + 1):
        yq[xq < 0] += np.sin(k * x_below)
    yq[xq >= 0] = np.cos(10 * x_above)
    return yq
y = func(X)

# define L2 relative error
def L2_rel_error(approx, actual):
    top = np.sum(np.square(approx - actual))
    bottom = np.sum(np.square(actual))
    return np.sqrt(top / bottom)

# train neural network
import torch
import torch.nn as nn
import torch.optim as optim

if torch.cuda.is_available():
  print("CUDA available. Using GPU acceleration.")
  device = "cuda"
else:
  print("CUDA is NOT available. Using CPU for training.")
  device = "cpu"

# convert input and output data to PyTorch tensors
X = torch.tensor(X, dtype=torch.float32, device=device).reshape(-1, 1)
y = torch.tensor(y, dtype=torch.float32, device=device).reshape(-1, 1)
```

```
# testing domain
X_test = torch.tensor(np.linspace(-np.pi,np.pi,N_test), dtype=torch.float32,
                      device=device).reshape(-1, 1)

widths = [10, 30, 100, 300, 1000] # list of widths to test
loss_fn = nn.MSELoss()  # use MSE loss
n_epochs = 5e4 # number of epochs
lr = 0.001 # learning rate for Adam optimizer
import matplotlib.pyplot as plt
for w in widths:
    # define two-layer ReLU network architecture with width w for both hidden layers
    model = nn.Sequential(
        nn.Linear(1, w),
        nn.ReLU(),
        nn.Linear(w, w),
        nn.ReLU(),
        nn.Linear(w, 1)).to(device)
    # use Adam optimizer with learning rate lr
    optimizer = optim.Adam(model.parameters(), lr=lr)
    for epoch in (np.arange(n_epochs) + 1):
        y_pred = model(X)
        loss = loss_fn(y_pred, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f'Finished training with width {w}, latest loss {loss}')
    # testing
    y_pred = torch.flatten(model(X_test)).detach().cpu().numpy()
    X_testq = torch.flatten(X_test).cpu().numpy()
    y_actual = func(X_testq)
    plt.plot(X_testq, y_pred, label='Prediction')
    plt.plot(X_testq, y_actual, label='Actual')
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.title('Prediction vs. actual from shallow ReLU network \n '+
            'Network width: '+str(w)+', Number of epochs: '+str(n_epochs))
    plt.savefig('2_testing_'+str(w)+'.pdf')
    plt.close()
    # return L2 relative error for width w
    print(f'L2 relative error for width {w}: {L2_rel_error(y_pred, y_actual)}')
```

3 Problem 3

3.1 Introduction

As with Problems 1 and 2, we use the same scripts as described in Section 1.1 except replace `1.py` or `2.py` with `3.py` (Section 3.3.1). `3.py` is the same script as `1.py`—including the same architecture utilizing a shallow ReLU network—except we use 20 equally-spaced points on $[-\pi, \pi]$ for training instead of 200.

For run `i` of the code, these results are saved into `3_run_$i.txt` for `i`$= 1, 2, 3$ to be then used in `err_process.sh`.

3.2 Results

The network training was around three times as fast in Problem 3 ($\approx$ 1 minute/network) vs. Problem 1 ($\approx$ 3 minutes/network) and about twice as fast in Problem 1 vs. Problem 2 ($\approx$ 6 minutes/network). This should be expected, as we had to train 10x less samples—we remind the reader that in all cases, we did not do batch training.

Despite the speed-up in training, naturally, the L^2 relative testing error was noticeably larger than in either of the previous problems, never going below even 10%. And unfortunately, we still had the same issues with apparent overfitting, and maybe even worse since the observed linear trend in the log-log plot from Problem 1 (Figure 2) leading up to a network width of 300 cannot be found since the L^2 relative error already begins saturating too quickly.

In particular, note that neural networks feature both an approximation error and an estimation error. While the approximation error may increase if a model is not sufficiently complex, the estimation error also poses a problem in how if we have less points for training, we only have so much information about what the trained relationship is outside of that training set. Intuitively speaking, overfitting occurs in two general senses: 1) when one tries to force an overparameterized model to fit to information that is irrelevant or just noise or 2) when one makes too many assumptions about the regularity of a model and adds unphysical constraints to generate a unique solution, even though these are not actually physical. The latter is more relevant in this situation; there is only so much information one can get about an ideal model from 20 vs. 200 points.

3.3 Code

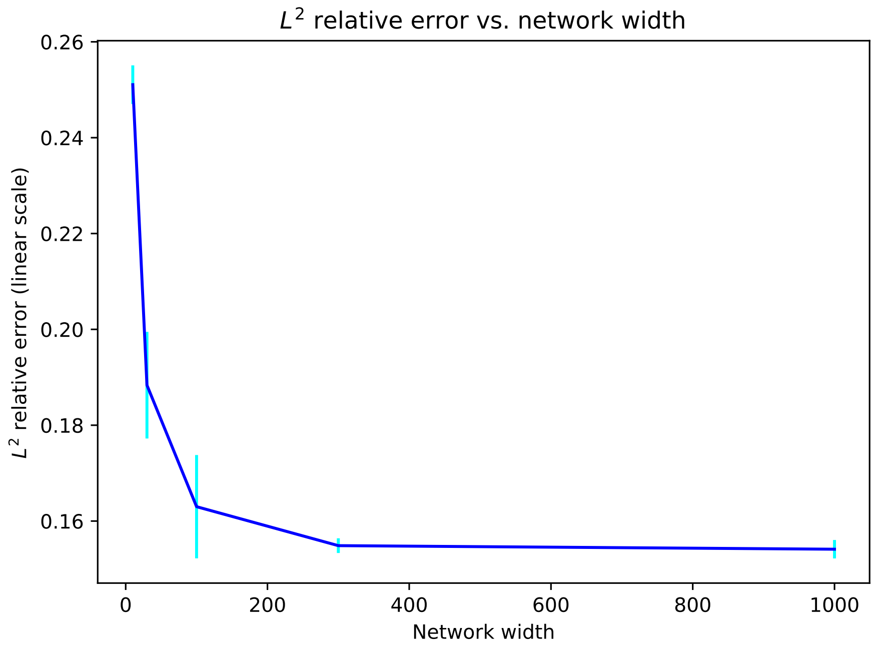
3.3.1 3.py

`3.py` is the primary script used for running the training and testing of the shallow ReLU network from Problem 3. `3.py` was used in tandem with the two scripts found in the Appendix (Section 7) to generate the L^2 relative error plots (Figure 5).

```
import numpy as np

# generate the dataset
N_data = 20 # number of points for data
N_test = 689 # number of points for testing
k_tot = 4 # number of elements in sinusoidal sum
X = np.linspace(-np.pi,np.pi,N_data) # create grid ofN_data points in x

# define function to evaluate f on grid points in X
```



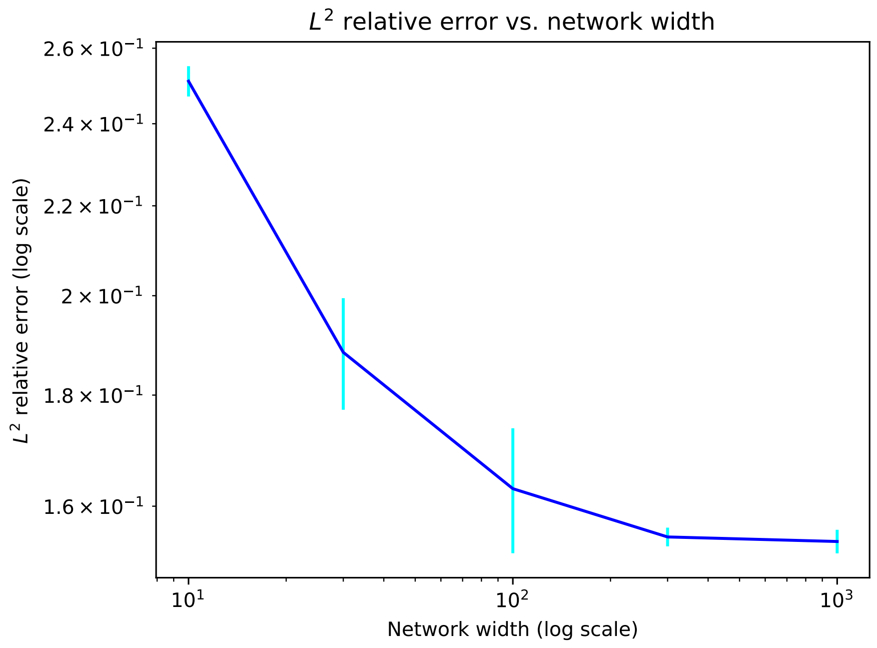


Figure 5: Convergence of L^2 relative error for testing on f (eq. (1)) with network width for the network architecture in Problem 3.

```
def func(xq):
    yq = np.zeros(len(xq))
    # separate X into segments below and above zero
    x_below = xq[xq < 0]
    x_above = xq[xq >= 0]
    yq[xq < 0] = 5
    for k in (np.arange(k_tot) + 1):
        yq[xq < 0] += np.sin(k * x_below)
    yq[xq >= 0] = np.cos(10 * x_above)
    return yq
y = func(X)

# define L2 relative error
def L2_rel_error(approx, actual):
    top = np.sum(np.square(approx - actual))
    bottom = np.sum(np.square(actual))
    return np.sqrt(top / bottom)

# train neural network
import torch
import torch.nn as nn
import torch.optim as optim

if torch.cuda.is_available():
  print("CUDA available. Using GPU acceleration.")
  device = "cuda"
else:
  print("CUDA is NOT available. Using CPU for training.")
  device = "cpu"

# convert input and output data to PyTorch tensors
X = torch.tensor(X, dtype=torch.float32, device=device).reshape(-1, 1)
y = torch.tensor(y, dtype=torch.float32, device=device).reshape(-1, 1)

# testing domain
X_test = torch.tensor(np.linspace(-np.pi,np.pi,N_test), dtype=torch.float32,
                      device=device).reshape(-1, 1)

widths = [10, 30, 100, 300, 1000] # list of widths to test
loss_fn = nn.MSELoss()  # use MSE loss
n_epochs = 5e4 # number of epochs
lr = 0.001 # learning rate for Adam optimizer
import matplotlib.pyplot as plt
for w in widths:
    # define shallow ReLU network architecture with width w
    model = nn.Sequential(
        nn.Linear(1, w),
        nn.ReLU(),
        nn.Linear(w, 1)).to(device)
```

```
# use Adam optimizer with learning rate lr
optimizer = optim.Adam(model.parameters(), lr=lr)
for epoch in (np.arange(n_epochs) + 1):
    y_pred = model(X)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
print(f'Finished training with width {w}, latest loss {loss}')
# testing
y_pred = torch.flatten(model(X_test)).detach().cpu().numpy()
X_testq = torch.flatten(X_test).cpu().numpy()
y_actual = func(X_testq)
plt.plot(X_testq, y_pred, label='Prediction')
plt.plot(X_testq, y_actual, label='Actual')
plt.legend()
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Prediction vs. actual from shallow ReLU network \n '+
        'Network width: '+str(w)+', Number of epochs: '+str(n_epochs))
plt.savefig('3_testing_'+str(w)+'.pdf')
plt.close()
# return L2 relative error for width w
print(f'L2 relative error for width {w}: {L2_rel_error(y_pred, y_actual)}')
```

4 Problem 4

4.1 Introduction

I saw the first video. I appreciate how he sort of adopted 3Blue1Brown's style in combining both auditory and visual cues in his teaching. Even the music was similar.

I have already seen automatic differentiation before, but it was a great review.

4.2 Results

N/A

4.3 Code

N/A

5 Problem 5

5.1 Introduction

The problem is asking us to compute $f(x)$, where

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(x-\sigma)^2}{\sigma^2}\right) \tag{2}$$

is a Gaussian PDF with $\sigma = 1$ and $\mu = 0$, across $N = 10000$ uniformly distributed points on $[-5, 5]$. Since the problem asks that we do this on the CPU, we do it on the CPU. As for the package, we use the JAX implementation of `scipy.stats` with the intention of eventually using `grad` and `vmap` from JAX in Problem 6. But for this problem, doing this allows us to take the JAX version of the `scipy.stats.norm` (normal distribution) class and extract the PDF function they have there. We also implement the grid in x using the JAX implementation of `numpy.linspace` to keep as many things within JAX as possible.

To compute the average runtime, because it only took less than a minute in wallclock time to do this on my CPU (and to keep the average runtime more consistent), I ran my code 5×10^4 times. I also used `timeit.timeit` within Python because this allows us to repeat a block of code a specified number of times quite easily. This is all done within a single Python script, `5.py` (Section 5.3.1).

5.2 Results

The output of `5.py` is as follows after one run on `5.py` (which corresponds to running the code which evaluates (2)5×10^4 times on $N = 10000$ uniformly distributed points on $[-5, 5]$ using JAX):

```
Average time: 0.0008475138339982368 seconds
CPU: Intel64 Family 6 Model 140 Stepping 1, GenuineIntel
```

Hence, using my Intel64 Family 6 Model 140 Stepping 1 CPU, evaluating (2) $N = 10000$ times on uniformly distributed points on $[-5, 5]$ for $\sigma = 1$ and $\mu = 0$ takes an average wallclock runtime of 0.848 milliseconds. Furthermore, after running `wmic cpu get name` from the command line, we have the following output:

```
Name
11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz
```

thereby demonstrating that my CPU, 11th Gen Intel Core i7-11370H, also has a clock rate of 3.30GHz. ("Intel64 Family 6 Model 140 Stepping 1" and "11th Gen Intel Core i7-11370H" are synonymous names for my CPU.)

5.3 Code

5.3.1 5.py

`5.py` evaluates f (eq. (2)) on a uniform grid of 10000 points from $[-5, 5]$ using the JAX implementations of `scipy.stats.norm` and `numpy`. The code repeats this operation 5×10^4 times, computes the average wallclock time, and returns the CPU used for the operations.

```
# define setup code
setup = '''
# define parameters
N = 10000 # batch size
sigma = 1
mu = 0
a = -5 # start value for domain
b = 5 # end value for domain

# download normal distribution PDF and np.linspace from JAX
from jax.scipy.stats.norm import pdf
from jax.numpy import linspace

# define function f(x) from Problem 5
def gaussian_f(N, mu, sigma, a, b):
    # define grid on which to evaluate normal distribution, reshape to (N,1)
    X = linspace(a, b, N).reshape((N, 1))
    return pdf(X, loc=mu, scale=sigma)  # evaluate f(x) at points in X
'''

# define test code
test = "gaussian_f(N, mu, sigma, a, b)"

# test timing
n = int(5e4) # number of trials
import timeit # get timer to time n runs of gaussian_f
total_time = timeit.timeit(setup=setup, stmt=test, number=n)
print(f'Average time: {total_time / n} seconds')
import platform # get CPU information
print(f'CPU: {platform.processor()}')
```

6 Problem 6

6.1 Introduction

All timing results and the plots for this problem (Figure 6) were processed using `6.py`. As in Problem 5, we again used JAX to define the grid in x (via the JAX implementation of `numpy.linspace`) and define the Gaussian PDF $f(x)$ (2) (via the JAX implementation of the `norm` module in `scipy.stats`). To implement the autodifferentiation (AD) of f, we first applied `jax.grad` to define the AD of f then `jax.vmap` to vectorize this operation across all points on which f is to be evaluated. To test the time it took to autodifferentiate f, as in Problem 5, we use `timeit.timeit` to run the AD procedure 5×10^3 times. We use less runs of the script than in Problem 5 because, as one would expect, using AD to compute $f'(x)$ still takes longer than just computing $f(x)$.

After timing the computation of $f'(x)$ in JAX, we use the script for testing, execute it, save the result, and compare it with the exact values of $f'(x)$ at the same $N = 10000$ uniformly distributed points on $[-5, 5]$. f' can be computed exactly via repeated application of chain rule on paper, which is ultimately the basis for what AD does as well in this instance. In particular, $f'(x) = -(x - \mu) f(x) / \sigma^2$.

Note that AD only works on a function f if 1) we know the form of f and 2) the implementation of AD can recognize how f is expressed via operations of addition, multiplication, and composition of functions with known derivatives. However, by keeping all of our libraries within JAX, we ensure that the implementation of AD is compatible with our implementation of f.

6.2 Results

After running AD 5×10^3 times on f to compute f', the average wallclock runtime is 18.1 milliseconds, which is over an order of magnitude more time than merely computing f. Using AD to compute f' also takes about an order of magnitude more runtime than computing f' via direct implementation of the exact formula, as I was able to confirm by changing a few lines in `6.py` to time the operation `pdf(X, loc=mu, scale=sigma) * (mu - X) / (sigma ** 2)` instead. Nonetheless, considering the computational complexity involved with AD and the fact that applying AD takes only two orders of magnitude less than a second, this result is quite impressive. Furthermore, as Figure 6 confirms, AD is able to compute f' from f within machine epsilon. There is no error within machine epsilon between f' as computed via AD vs. direct implementation of the exact formula.

6.3 Code

6.3.1 6.py

`6.py` evaluates f' (the derivative of eq. (2)) on a uniform grid of 10000 points from $[-5, 5]$ using autodifferentiation as implemented in JAX in tandem with the JAX implementations of `scipy.stats.norm` and `numpy`. The code repeats this operation 5×10^3 times, computes the average wallclock time, and returns the CPU used for the operations. After doing so, the code runs this operation once more and saves the result, computes f' using the exact formula (also using JAX operations), and then produces/saves the plots found in Figure 6.

```
# define setup code
setup = '''
# define parameters
```

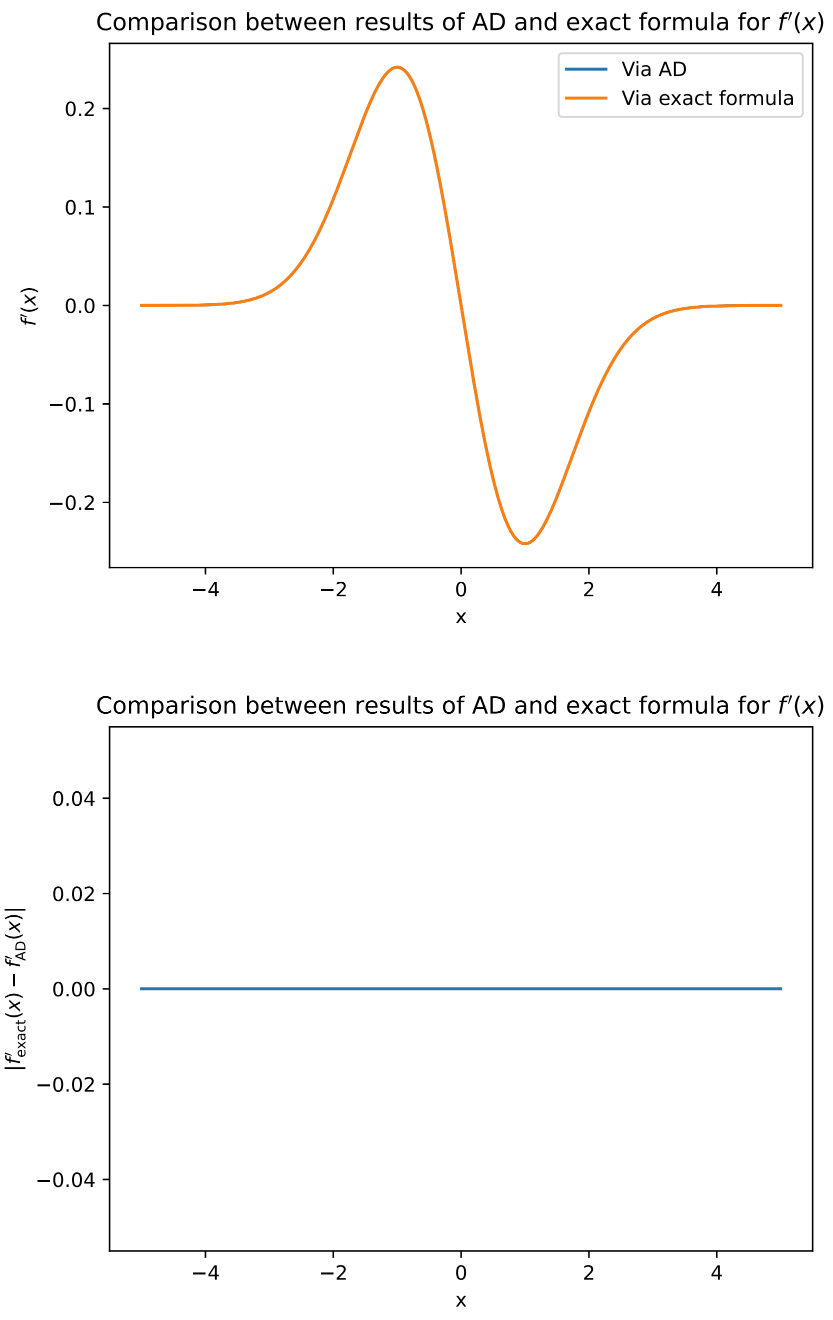


Figure 6: Plots comparing the results of applying AD on $f(x)$ (eq. (2)) to compute $f'(x)$ in JAX, $f'_{\mathrm{AD}}(x)$, vs. directly computing $f'(x)$ using the known formula, $f'_{\mathrm{exact}}(x)$, for x uniformly distributed in 10000 points on $[-5, 5]$. The top plot shows overlapping plots of $f'_{\mathrm{AD}}(x)$ (blue) and $f'_{\mathrm{exact}}(x)$ (orange) vs. x and the bottom plot shows $|f'_{\mathrm{exact}}(x) - f'_{\mathrm{AD}}(x)|$ vs. x (blue).

```
N = 10000 # batch size
sigma = 1
mu = 0
a = -5 # start value for domain
b = 5 # end value for domain

# download normal distribution PDF from JAX
from jax.scipy.stats.norm import pdf

# define function f from Problem 5, evaluated at points in x
gaussian_f = lambda x: pdf(x, loc=mu, scale=sigma)[0]  # evaluate f at points in x

# define first-derivative f' of f from Problem 5, evaluated at the parameters
# define differentiation via jax.grad, do over multiple arguments using vmap
from jax import grad, vmap
gaussian_f_prime = vmap(grad(gaussian_f))

from jax.numpy import linspace
# use gaussian_f_prime to predict f'(x) for N uniform points in [a, b]
X = linspace(a, b, N).reshape((N, 1))
'''

# define test code
test = "gaussian_f_prime(X)"

# test timing
n = int(5e3) # number of trials
import timeit # get timer to time n runs of gaussian_f
total_time = timeit.timeit(setup=setup, stmt=test, number=n)
print(f'Average time: {total_time / n} seconds')

exec(setup) # execute setup code now within script
f_prime_jax = gaussian_f_prime(X)
# compute f'(x) using the exact formula
f_prime_exact = pdf(X, loc=mu, scale=sigma) * (mu - X) / (sigma ** 2)

# plot f_prime_jax and f_prime_exact
import matplotlib.pyplot as plt
plt.plot(X, f_prime_jax, label='Via AD')
plt.plot(X, f_prime_exact, label='Via exact formula')
plt.title("Comparison between results of AD and exact formula for $f'(x)$")
plt.xlabel('x')
plt.ylabel("$f'(x)$")
plt.legend()
plt.savefig('6_1.pdf')
plt.close()

from jax.numpy import abs
plt.plot(X, abs(f_prime_exact-f_prime_jax))
plt.title("Comparison between results of AD and exact formula for $f'(x)$")
plt.ylabel(r"$\left|f_{\mathrm{exact}}'(x)-f_{\mathrm{AD}}'(x)\right|$")
plt.xlabel('x')
plt.savefig('6_2.pdf')
plt.close()
```

7 Appendix: Multi-problem codes

7.1 err_process.sh

`err_process.sh` processes multiple runs of `1.py`, `2.py`, and `3.py`—which are the L^2 relative errors for testing from the network architectures in Problems 1, 2, and 3, respectively—for input into `plt_rel_errors.py`. The script must be changed manually each time depending on which network architecture is being considered (1, 2, or 3).

```
# bash script to process the outputs of 1.py, 2.py, and 3.py,
# then input into plt_rel_errors.py
# these have been piped into scripts of the forms
# 1_run*.txt, 2_run*.txt, and 3_run*.txt, respectively
cd /c/Users/jonas/Documents/SciML_class/HW2 # change to correct local folder

rm tmp0.txt
rm tmp.txt
# get widths tested (assume same for all cases), save in tmp.txt
grep -i 'L2' 1_run_1.txt | awk '{print $6}' | sed s/://g > tmp0.txt
# load samples for each sample
cp tmp0.txt tmp.txt
# loop through all errors
for file in 3_run_*.txt # change number depending on the problem (1, 2, or 3)
do
    # add column with new L2 relative error samples
paste tmp0.txt <(grep -i 'L2' $file | awk '{print $NF}') > tmp.txt
    cat tmp.txt > tmp0.txt
done
cat tmp.txt | python3 plt_rel_errors.py # pipe errors into plt_rel_errors.py
```

7.2 plt_rel_errors.py

With the help of `err_process.sh` to process the outputs of `1.py`, `2.py`, and `3.py`, `plt_rel_errors.py` plots the relative L^2 errors for testing the network architectures from Problems 1, 2, and 3, respectively, as functions of the network width.

```
# script for plotting the relative errors
# input is text file where each row is a network width
# each column corresponds to a new run,
# except the first which contains the list of widths

import sys
import numpy as np

err_data = np.loadtxt(sys.stdin, dtype=np.float64) # load error data
# preallocate where to store widths, error means, and error standard deviations
err = np.zeros((len(err_data), 3))
for i, w in enumerate(err_data):
    errs = w[1:]
    err[i] = np.array([w[0], np.mean(errs), np.std(errs)])
```

```
# save two plots, first in linear scaling and second with log scaling for both axes
import matplotlib.pyplot as plt
plt.errorbar(x=err[:,0], y=err[:,1], yerr=err[:,2], fmt='b-', ecolor='cyan')
plt.title('$L^{2}$ relative error vs. network width')
plt.xlabel('Network width')
plt.ylabel('$L^{2}$ relative error (linear scale)')
plt.tight_layout()
plt.savefig('3_lin_scaling.pdf') # change file name here
plt.close()

plt.errorbar(x=err[:,0], y=err[:,1], yerr=err[:,2], fmt='b-', ecolor='cyan')
plt.title('$L^{2}$ relative error vs. network width')
plt.xlabel('Network width (log scale)')
plt.ylabel('$L^{2}$ relative error (log scale)')
plt.xscale('log')
plt.yscale('log')
plt.tight_layout()
plt.savefig('3_log_scaling.pdf') # change file name here
plt.close()
```