Assignment 2: Regression with Neural Networks

Sanaz Marefati

October 2, 2025

1 Functions and Deliverables

The functions to approximate are:

1. (1D, discontinuous) Heaviside function on $x \in [-2, 2]$.
2. (1D) $\Gamma(x) = \sqrt{2 - x}$ on [0, 2], Holder continuous with exponent 1/2.
3. (2D, multi-modal) $\Gamma(x, y) = \exp(-k_1(x - 0.5)^2 - (y - 0.5)^2) + \exp(-k_2(x + 0.5)^2 - (y + 0.5)^2)$ on $[-1, 1]^2$, with $k_1 = k_2 = 1$ (optional $k_1 = k_2 = 10$).
4. Rosenbrock function $f(x, y) = 100(y - x^2)^2 + (1 - x)^2$ on [-2, 2] × [-2, 2].
5. (High-dimensions, optional) $\Gamma(x) = \frac{1}{(2\pi)^{d/2}} \exp\left(-\sum_{i=1}^{d} x_i^2\right)$ on $[0, 1]^d$ for $d = 1$ to 8.

Deliverables include plots of functions, NN approximations, absolute/relative errors, code, explanations of choices, and comments on best settings.

2 Methodology

Each target function was approximated using fully connected neural networks trained on quasi–Monte Carlo (QMC) Sobol points. For the 1D problems, 100 points were used, while for the 2D problems 10,000 points were generated, with independent Sobol sequences for training and testing. All inputs were normalized to $[-1, 1]$, and for the Rosenbrock function the outputs were additionally mean–std normalized to improve numerical stability.

Two architectures were employed: a shallow network with one hidden layer of 40 neurons, and a deep network with three hidden layers of 20 neurons each (comparable total neuron count). Both `ReLU` and `tanh` activations were tested, using He uniform and variance scaling initialization respectively. Training was performed with the ADAM optimizer and a polynomially decaying learning rate ($10^{-3} \to 10^{-4}$) over 10,000 epochs. A summary of these settings is provided in Table 1.

Remark. For the Rosenbrock function, the initial error was large due to insufficient output normalization. Applying mean–std scaling to the outputs prior to training, and denormalizing for error computation, significantly improved model performance.

Table 1: Summary of neural network settings and best performers for each function.

Function	**Heaviside**	$\sqrt{2-x}$	**2D Gaussians**	**Rosenbrock**
Training points	100 (QMC Sobol)	100 (QMC Sobol)	10,000 (QMC Sobol)	10,000 (QMC Sobol)
Test points	100	100	10,000	10,000
NN architecture	Shallow (1,40,1)	Deep (1,20,20,20,1)	Deep (2,20,20,20,1)	Deep (2,20,20,20,1)
Activation function	`ReLU`	`tanh`	`tanh`	`ReLU`
Initialization	He_uniform (`ReLU`)	Variance Scaling (`tanh`)	Variance Scaling (`tanh`)	He_uniform (`ReLU`)
Optimizer	ADAM (lr=10^{-3}, decay to 10^{-4})	ADAM (lr=10^{-3}, decay to 10^{-4})	ADAM (lr=10^{-3}, decay to 10^{-4})	ADAM (lr=10^{-3}, decay to 10^{-4})
Epochs	10,000	10,000	10,000	10,000
Best performer	Shallow `ReLU`	Deep `tanh`	Deep `tanh`	Deep `ReLU`

2.1 Code for the Problems

```
#========== Setup ==========
import jax, jax.numpy as jnp
from jax import random, vmap, value_and_grad, jit, grad, nn,
    jacrev
from jax.nn import relu, tanh, initializers
from jax.nn.initializers import variance_scaling, he_normal,
    he_uniform
import optax
from scipy.stats import qmc
from scipy.interpolate import griddata
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize, LightSource
from mpl_toolkits.mplot3d import Axes3D
import time

#========== Define functions and datasets ==========
def f_heaviside(x):
    return jnp.where((x >= 0.0) & (x <= 2.0), 1.0, 0.0)

def f_gamma(x):
    return jnp.sqrt(jnp.maximum(2.0 - x, 0.0))

def f_gaussians(x1, x2):
    k1, k2 = 1.0, 1.0
    #k1, k2 = 10, 10
    return jnp.exp(-(k1*(x1 - 0.5)**2 + (x2 - 0.5)**2)) + \
           jnp.exp(-(k2*(x1 + 0.5)**2 + (x2 + 0.5)**2))

def f_rosenbrock(x1, x2):
    return 100*(x2 - x1**2)**2 + (1-x1)**2

def f_gaussian_highdim(x):
    d = x.shape[-1]
    prefactor = 1 / (2 * jnp.pi)**(d / 2)
    return prefactor * jnp.exp(-jnp.sum(x**2, axis=-1))

datasets = {
    'heaviside': {
        'dim': 1,
        'domain': [(-2, 2)],
        'func': f_heaviside,
        'name': 'Heaviside'
    },
    'gamma': {
        'dim': 1,
        'domain': [(0, 2)],
        'func': f_gamma,
        'name': 'sqrt(2-x)'
    },
```

```
    'gaussians': {
        'dim': 2,
        'domain': [(-1, 1), (-1, 1)],
        'func': f_gaussians,
        'name': '2D Gaussians'
    },
    'rosenbrock': {
        'dim': 2,
        'domain': [(-2, 2), (-2, 2)],
        'func': f_rosenbrock,
        'name': 'Rosenbrock'
    }
}
#high_dim_ds = lambda d: {'f': f_gaussian_highdim, 'dim': d,
   'domain': [(0, 1)]*d, 'name': f'High-d Gaussian (d={d})'}

def bounds(domain):
    lower = jnp.array([d[0] for d in domain])
    upper = jnp.array([d[1] for d in domain])
    return lower, upper

def generate_qmc_samples(dim, n, domain, seed=0):
    sampler = qmc.Sobol(d=dim, scramble=True, seed=seed)
    samples = sampler.random(n=n)
    lower, upper = bounds(domain)
    samples = lower + samples * (upper - lower)
    return jnp.array(samples)

def normalize(x, domain):
    lower, upper = bounds(domain)
    return 2 * (x - lower) / (upper - lower + 1e-10) - 1

# ========== Neural Network Architecture ==========
def initialize_mlp(sizes, key, activ_name):
    keys = random.split(key, len(sizes))

    def initialize_layer(m, n, k, activ_name):
        w_key, b_key = random.split(k)
        if activ_name == 'relu':
            initializer = he_uniform()  # He initialization for
               relu
            #initializer = he_normal()
        elif activ_name == 'tanh':
            initializer = variance_scaling(1.0, 'fan_avg',
               'uniform')
            #initializer = glorot_uniform()
        else:
            raise ValueError("Unsupported activation.")
        W = initializer(w_key, (m, n))
        b = jnp.zeros((n,))
        return W, b
```

```
96
97      return [initialize_layer(m, n, k, activ_name) for m, n, k in
            zip(sizes[:-1], sizes[1:], keys)]
98
99  # Forward pass
100 def forward(params, acts, activation):
101     for w, b in params[:-1]:
102         acts = activation(jnp.dot(acts, w) + b)
103     w, b = params[-1]
104     return jnp.dot(acts, w) + b
105
106 # Predict function
107 def predict(params, x, activation):
108     pred = vmap(lambda x_i: forward(params, x_i, activation))(x)
109     return jnp.squeeze(pred, axis=-1) # Shape: (n_data,)
110
111 # Loss function
112 def loss(params, x, y, activation):
113     y_pred = predict(params, x, activation)
114     return jnp.mean(jnp.square(y_pred- y))
115
116 # ========== Training with optax (ADAM with scheduler) ==========
117 def train_model(x_train, y_train, sizes, activation,
        epochs=10000, lr=1e-3, batch_size=32, seed=0):
118     key = random.PRNGKey(seed)
119     params = initialize_mlp(sizes, key, activation)
120
121     schedule = optax.polynomial_schedule(init_value=lr,
            end_value=lr * 0.1, power=2.0, transition_steps=1000 )
122     # Set up optimizer in Optax
123     optimizer = optax.adam(learning_rate=schedule)
124     opt_state = optimizer.init(params)
125
126     loss_grad = value_and_grad(loss, argnums=0)
127     loss_history = []
128
129     n_batches = len(x_train) // batch_size
130     for epoch in range(epochs):
131         epoch_loss = 0.0
132         perm = random.permutation(key, len(x_train))
133         x_train = x_train[perm]
134         y_train = y_train[perm]
135         for i in range(n_batches):
136             start = i * batch_size
137             end = start + batch_size
138             x_batch = x_train[start:end]
139             y_batch = y_train[start:end]
140             loss_val, grads = loss_grad(params, x_batch,
                    y_batch, activations[activ_name])
141             updates, opt_state = optimizer.update(grads,
                    opt_state)
```

```
            params = optax.apply_updates(params, updates)
            epoch_loss += loss_val
        loss_history.append(epoch_loss / n_batches)
        if epoch % 1000 == 0:
            print(f'Epoch [{epoch}/{epochs}], Loss:
               {loss_history[-1]:.6f}')
    return params, loss_history

# ========== Main Execution ==========
start_time = time.time() # Start timing

# Choose dataset
#selected_dataset = datasets['heaviside']
#selected_dataset = datasets['gamma']
selected_dataset = datasets['gaussians']
#selected_dataset = datasets['rosenbrock']

dim = selected_dataset['dim']
domain = selected_dataset['domain']
func = selected_dataset['func']
name = selected_dataset['name']

data_points=100 if dim == 1 else 10000
test_points=100 if dim == 1 else 10000

x_data = generate_qmc_samples(dim, data_points, domain, seed=0)
x_test = generate_qmc_samples(dim, test_points, domain, seed=1)
x_test_norm = normalize(x_test, domain)

y_data = func(x_data[:, 0]) if dim == 1 else
   func(*x_data.T[:dim])
y_test = func(x_test[:, 0]) if dim == 1 else
   func(*x_test.T[:dim])

arch_shallow = [dim, 40, 1]          # 60 neurons total in hidden
arch_deep    = [dim, 20, 20, 20, 1] # 20*3 = 60 neurons total
for activ_name in ['relu','tanh']:
    for name, size in [('shallow', arch_shallow), ('deep',
       arch_deep)]:
        params, hist = train_model(x_test_norm, y_test, size,
           activ_name, epochs=10000, lr=1e-3)
        y_pred = predict(params, x_test_norm,
           activations[activ_name])

abs_err = jnp.abs(y_pred - y_test)
rel_err = abs_err / (jnp.abs(y_test) + 1e-10)

print(f"{name} - Mean Absolute Error: {jnp.mean(abs_err):.6f},
   Mean Relative Error: {jnp.mean(rel_err):.6f}")

end_time = time.time()
```

```
print(f"Total Execution time: {end_time - start_time} seconds")

# ========== Visualization ==========
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
axes = axes.flatten()
plt.rcParams.update({'font.size': 10})

if dim == 1:
        axes[0].scatter(x_data[:, 0], y_data, c='b', s=10,
           alpha=0.8, label='True')
        axes[0].set_xlabel('Data Points')
        axes[0].set_ylabel('True Function')
        axes[0].grid(True, alpha=0.3)

        axes[1].scatter(x_test[:, 0], y_pred, c='orange', s=10,
           alpha=0.8, label='NN Approx')
        axes[1].set_xlabel('Data Points')
        axes[1].set_ylabel('NN Approximation')
        axes[1].grid(True, alpha=0.3)

        axes[2].scatter(x_test[:, 0], abs_err, c='r', s=10,
           alpha=0.8)
        axes[2].set_xlabel('Data Points')
        axes[2].set_ylabel('Absolute Error')
        axes[2].grid(True, alpha=0.3)

        axes[3].scatter(x_test[:, 0], rel_err, c='g', s=10,
           alpha=0.8)
        axes[3].set_xlabel('Data Points')
        axes[3].set_ylabel('Relative Error')
        axes[3].grid(True, alpha=0.3)
else:
        x1, x2 = x_data[:, 0], x_data[:, 1]
        x1i, x2i = jnp.linspace(min(x1), max(x1), 100),
           jnp.linspace(min(x2), max(x2), 100)
        xi, xii = jnp.meshgrid(x1i, x2i)
        points = jnp.stack([xi.ravel(), xii.ravel()], axis=1)

        y_true_grid = func(points[:, 0], points[:,
           1]).reshape(xi.shape) #griddata((x1, x2), y_data,
           (xi, xii), method='cubic')
        y_pred_grid = predict(params, points,
           activations[activ_name]).reshape(xi.shape)
           #((x_test[:, 0], x_test[:, 1]), y_pred, (xi, xii),
           method='cubic')

        abs_err_grid = jnp.abs(y_pred_grid - y_true_grid)
        rel_err_grid = abs_err_grid / (jnp.abs(y_true_grid) +
           1e-10)

        cmap_true = 'viridis' if 'Gaussian' in name else 'plasma'
```

```
226
227         axes[0].scatter(x1, x2, c=y_data, cmap='viridis' if name
               != 'Rosenbrock' else 'plasma', s=5)
228         axes[0].set_xlabel('x1')
229         axes[0].set_ylabel('x2')
230         cbar = plt.colorbar(axes[0].collections[0], ax=axes[0])
231         cbar.set_label('True Function', rotation=90)
232
233         cont = axes[1].contourf(xi, xii, y_pred_grid,
               levels=100, cmap='viridis' if name != 'Rosenbrock'
               else 'plasma')
234         axes[1].set_xlabel('x1')
235         axes[1].set_ylabel('x2')
236         cbar = plt.colorbar(cont, ax=axes[1])
237         cbar.set_label('NN Approximation', rotation=90)
238
239         axes[2].remove()
240         axes[2] = fig.add_subplot(2, 2, 3, projection='3d')
241         axes[2].plot_surface(xi, xii, abs_err_grid,
               cmap='Reds',linewidth=0, antialiased=True)
242         axes[2].set_xlabel('x1')
243         axes[2].set_ylabel('x2')
244         axes[2].set_zlabel('Absolute Error')
245         #axes[2].grid(True, alpha=0.3)
246
247         axes[3].remove()
248         axes[3] = fig.add_subplot(2, 2, 4, projection='3d')
249         axes[3].plot_surface(xi, xii, rel_err_grid,
               cmap='Greens', linewidth=0, antialiased=True)
250         axes[3].set_xlabel('x1')
251         axes[3].set_ylabel('x2')
252         axes[3].set_zlabel('Relative Error')
253         axes[3].grid(True, alpha=0.3)
254
255 fig.suptitle(f"{name} | Activation: {choice_act} | Layers NN:
      {choice_arch} | Training pts: {test_points}")
256 plt.tight_layout()
257 plt.savefig(f"{name} | Activation: {choice_act} | Layers NN:
      {choice_arch} | Training pts: {test_points}", dpi=300)
258 plt.show()
```

3 Results and Comments

Plots of the functions and their neural network approximations are shown in figures. Each figure includes the true function, the NN prediction, and the absolute and relative error. Networks were trained with QMC Sobol points, using the ADAM optimizer with a polynomially decaying learning rate ($10^{-3} \to 10^{-4}$), and both `ReLU` and `tanh` activations. Two architectures were considered: a shallow model with one hidden layer (40 units) and a deep model with three hidden layers (20 units each).

3.1 Function-wise Performance

- **Heaviside (1D)**: Shallow `ReLU` performed best, reflecting the piecewise linear nature of the function.
- $\sqrt{2-x}$ **(1D)**: Deep `tanh` networks achieve lower overall error due to smooth approximation, but at $x = 2$ `ReLU` yields smaller local error, showing its advantage in handling sharp slope changes.
- **2D Gaussians**: Deep `tanh` captured the multi-modal structure most effectively, both for $k = 1$ and $k = 10$.
- **Rosenbrock (2D)**: Deep `ReLU` performed best, handling the narrow curved valley after output normalization.

Summary: Shallow `ReLU` networks are well-suited for discontinuous or piecewise functions, while deep `tanh` networks are superior for smooth, complex, and higher-dimensional problems. In the case of the 2D Gaussians, when $k = 10$ the peaks become narrower and sharper compared to $k = 1$, creating highly localized structures. This makes the approximation problem more challenging and increases the need for deeper architectures and smooth activations (`tanh`). Shallow or `ReLU`-based networks tend to underfit in this setting, whereas deep `tanh` networks achieve the lowest errors.

For the Rosenbrock function, `ReLU` sometimes converges faster under a limited training budget. However, with longer training and improved normalization, `tanh` networks generally catch up and surpass `ReLU` in capturing the nonlinear valley.

3.2 Comments on Performance

- `ReLU` converges quickly but is less accurate for smooth functions.
- `tanh` converges more slowly but achieves lower final errors on continuous problems.
- Deep networks outperform shallow ones for multi-modal or nonlinear landscapes.
- QMC Sobol sampling improves coverage and stability compared to random sampling.

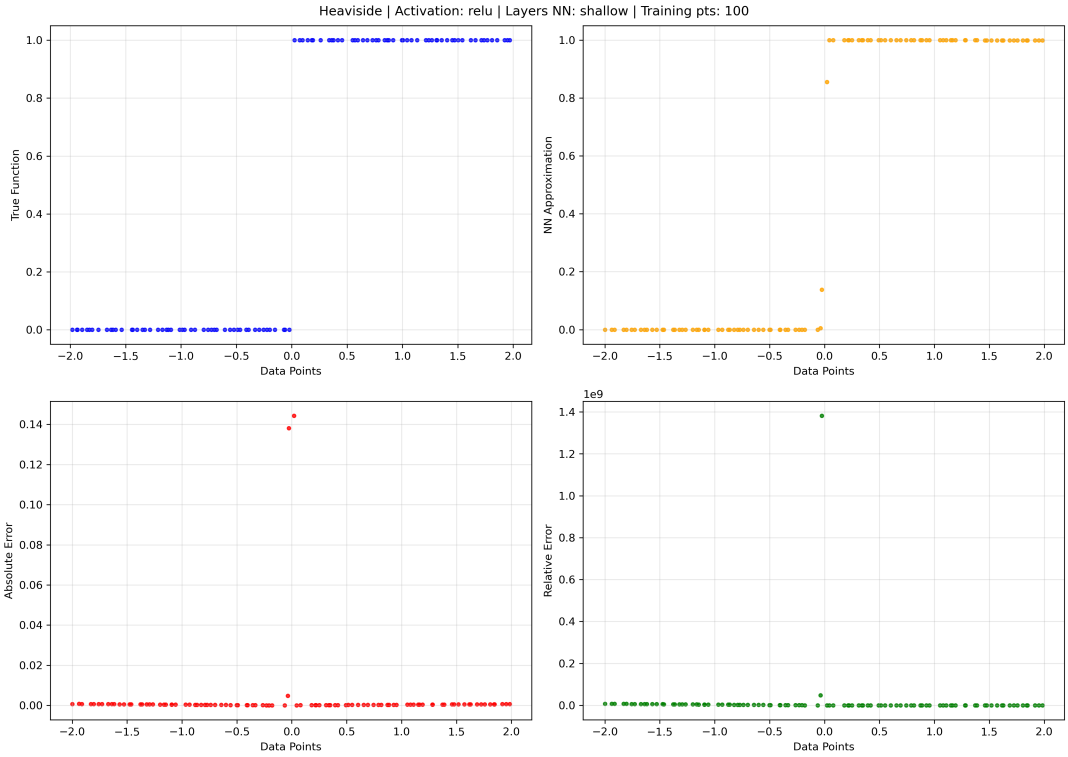

Figure 1: Heaviside step function results for shallow `ReLU` which performs best and reflects its suitability for piecewise functions.

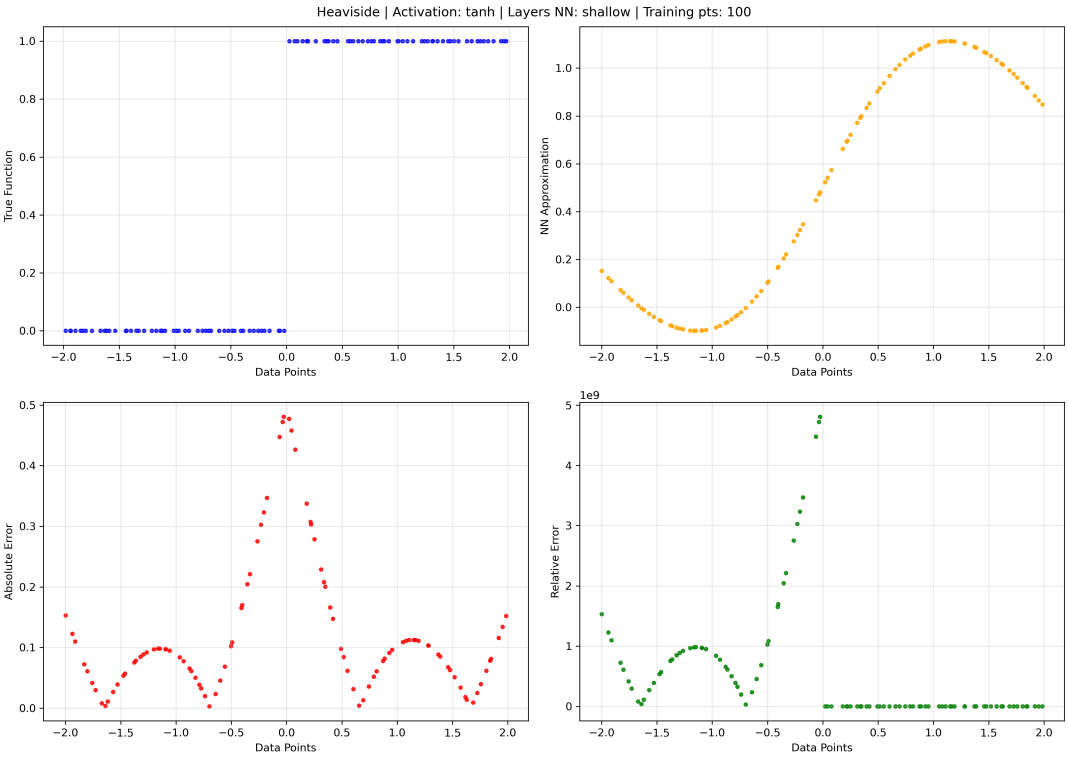

Figure 2: Heaviside step function results for Shallow `tanh`.

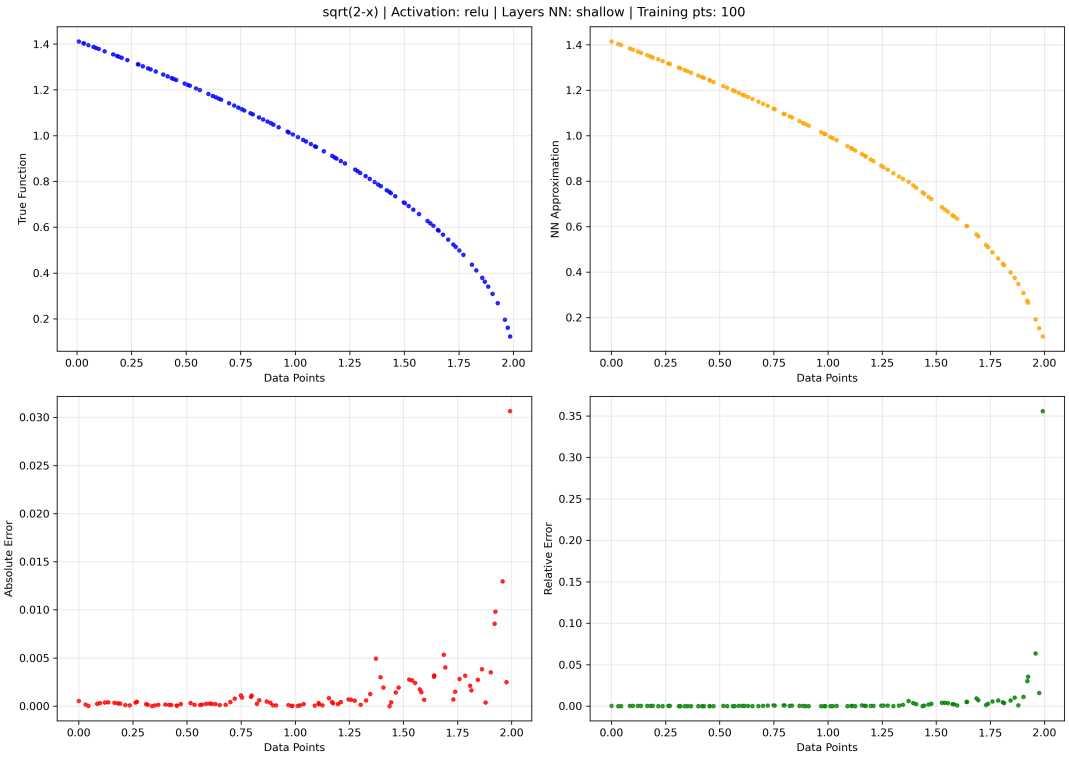

Figure 3: $\sqrt{(2-x)}$ function results for shallow `ReLU`.

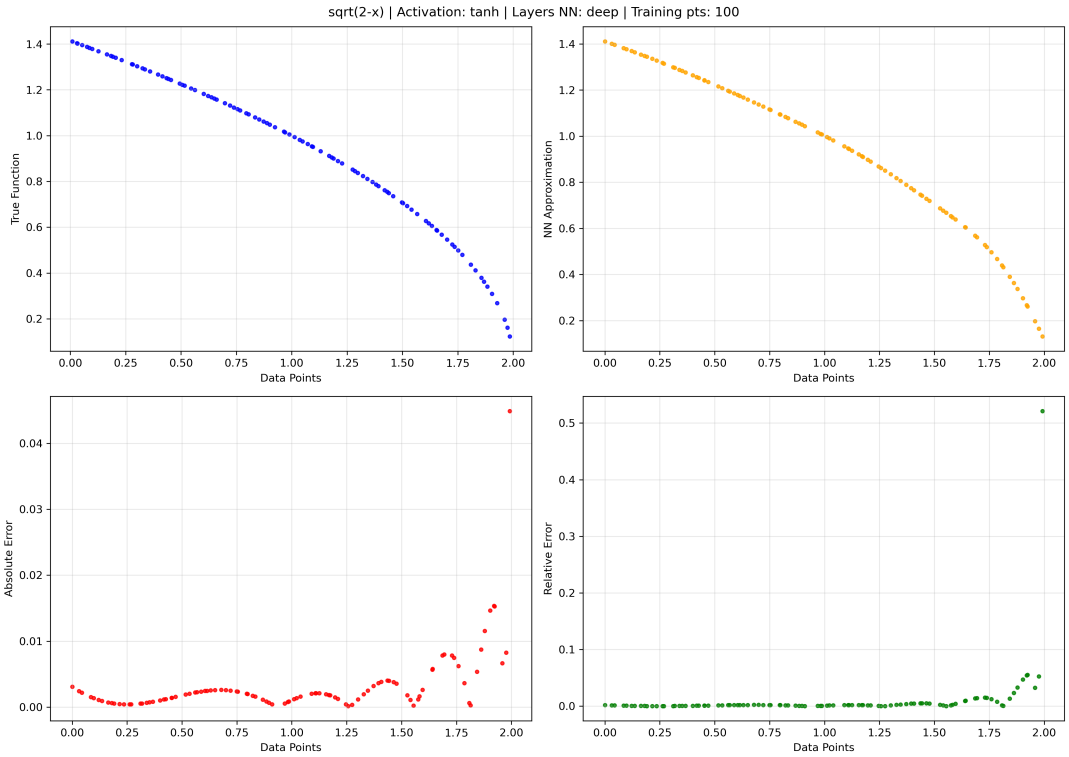

Figure 4: $\sqrt{(2-x)}$ function results for deep `tanh` which performs best and reflects its smoothness for this Holder continuous function.

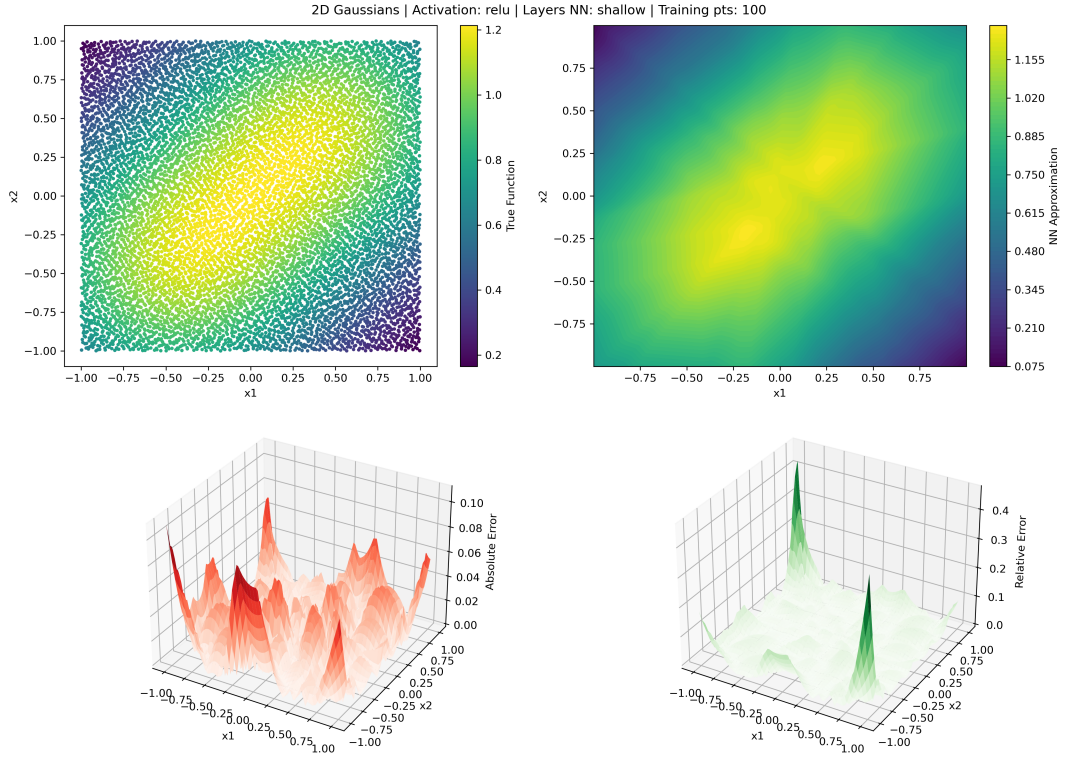

Figure 5: 2D Gaussians ($k = 1$) function results for shallow `ReLU`.

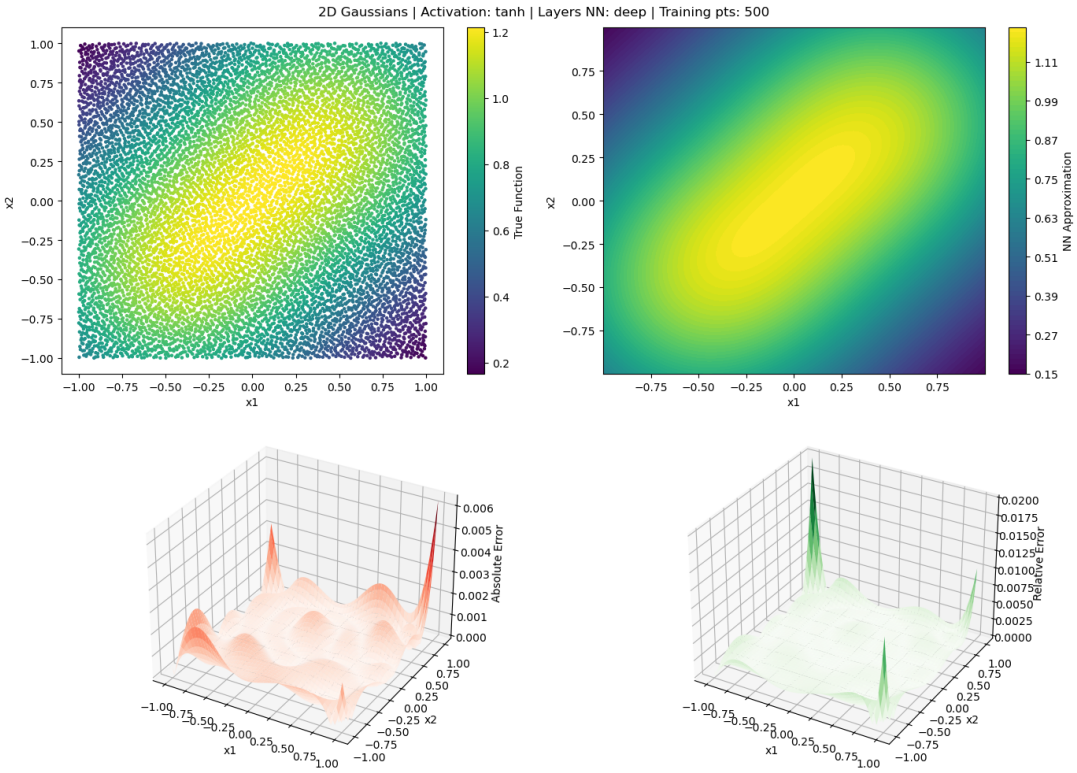

Figure 6: 2D Gaussians ($k = 1$) function results for deep `tanh` which achieves the lowest mean absolute error and benefits from its ability to capture multi-modal smooth structures.

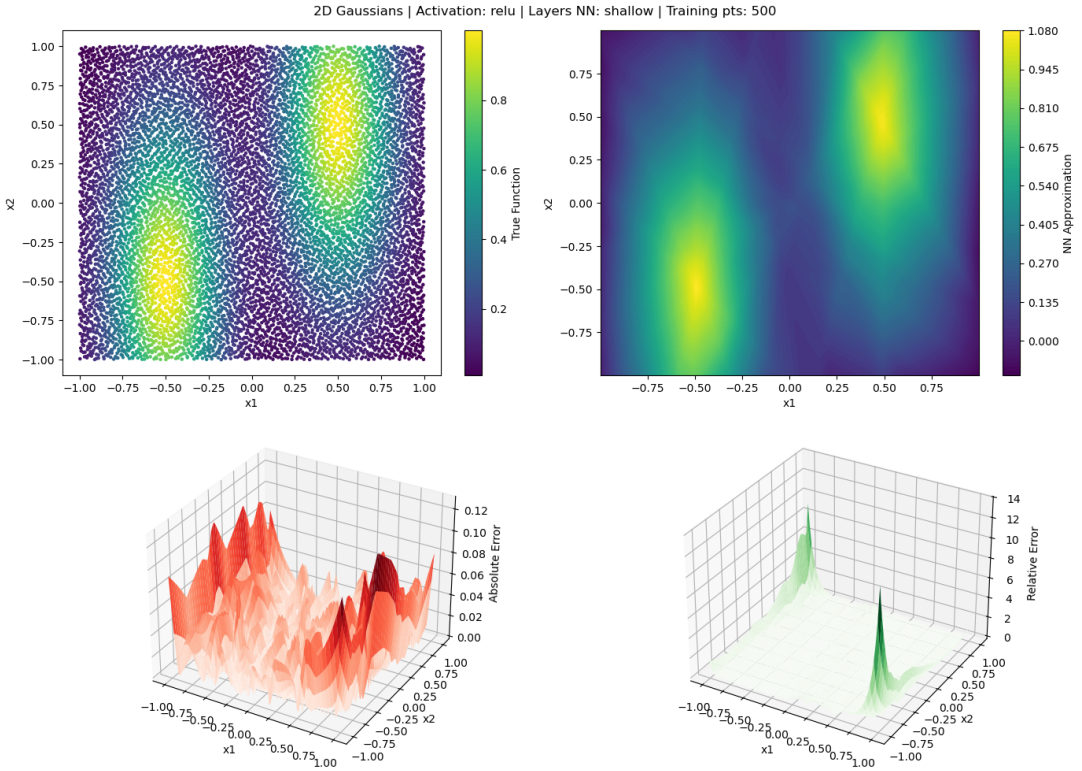

Figure 7: 2D Gaussians ($k = 10$) function results for shallow `ReLU`.

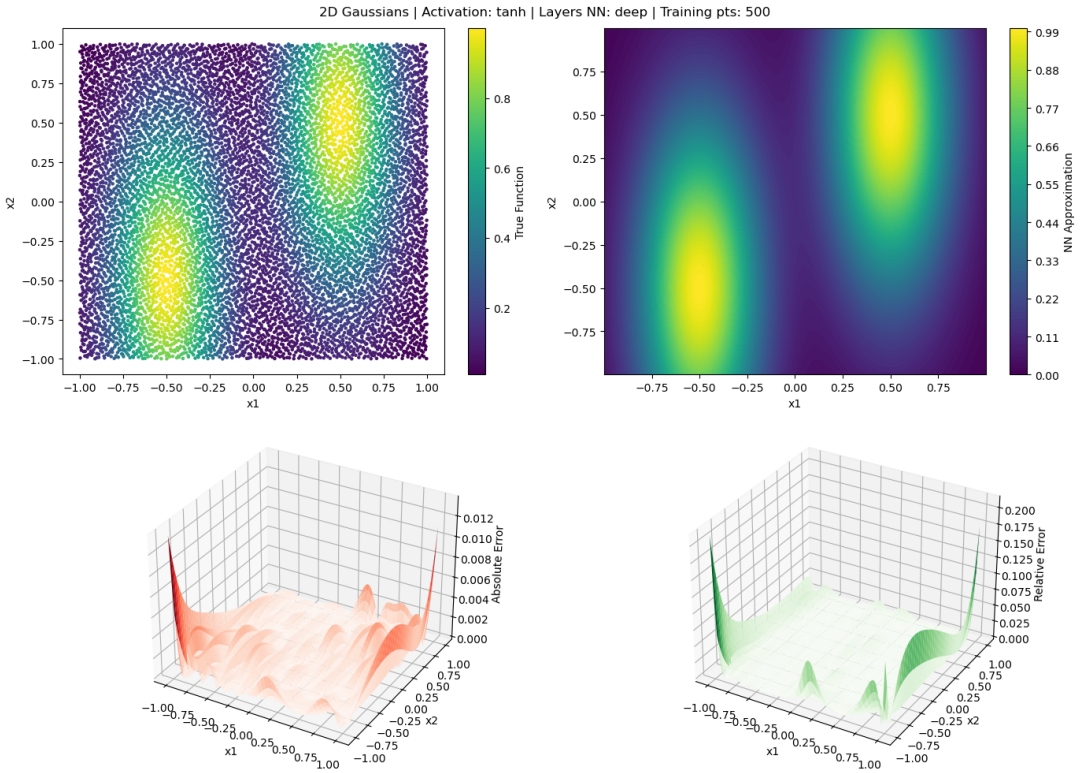

Figure 8: 2D Gaussians ($k = 10$) function results for deep `tanh`.

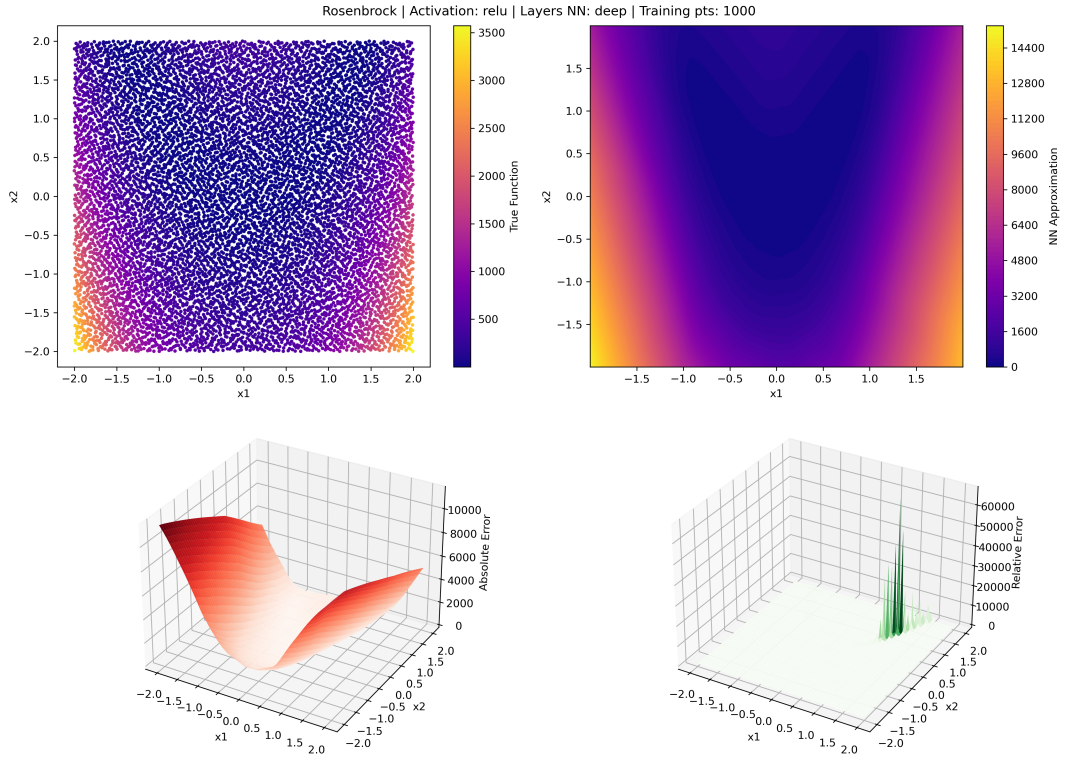

Figure 9: Rosenbrock function results for shallow `ReLU` which performs best and hand the complex non-linear valley effectively after y-normalization.

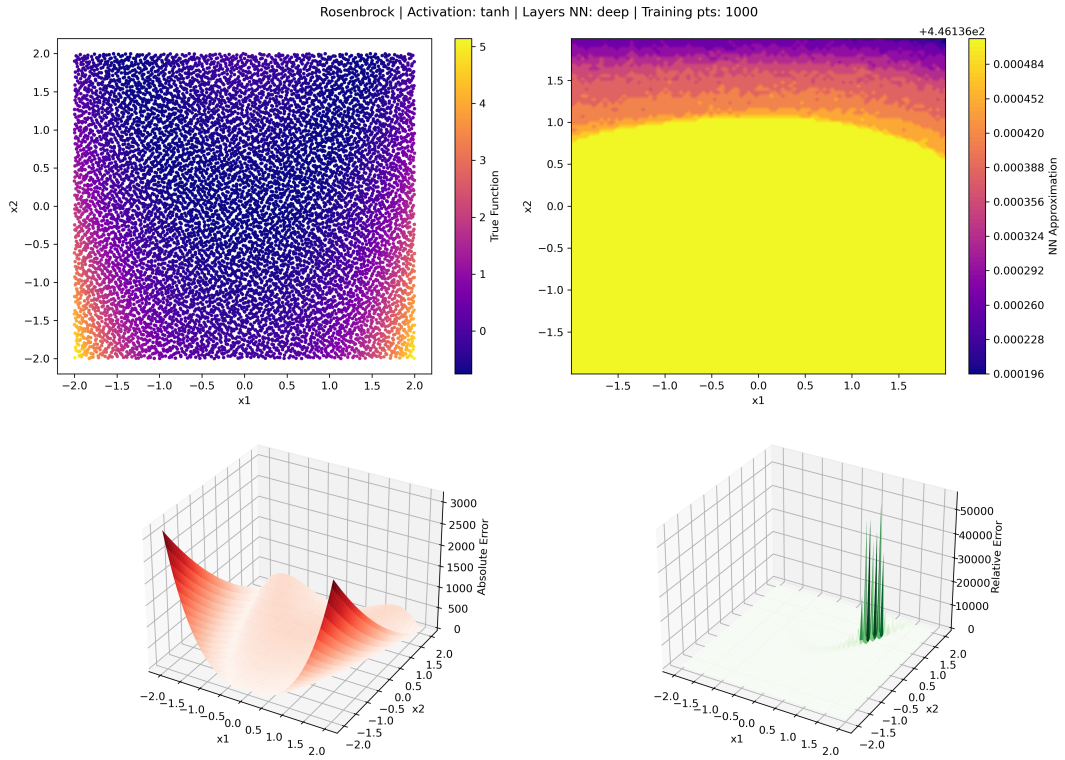

Figure 10: Rosenbrock function results for deep `tanh`.

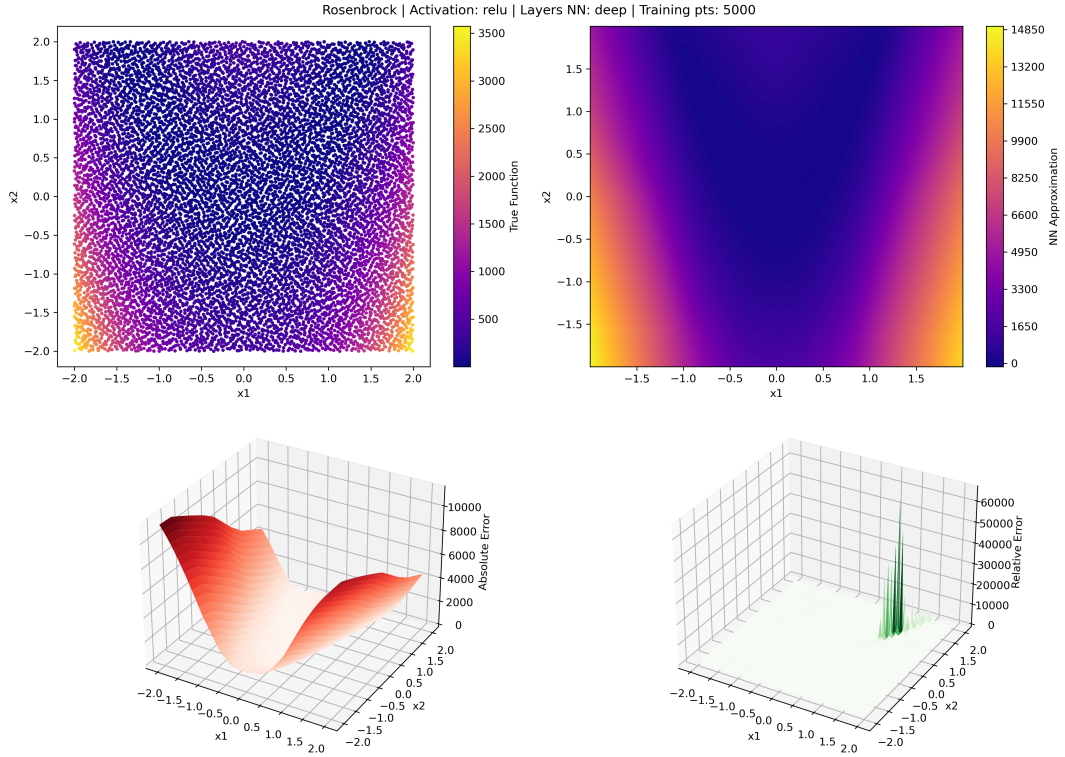

Figure 11: Rosenbrock function results for deep `ReLU`.