ECON622: Problem Set 4

Jesse Perla

Packages

Add whatever packages you wish here

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
import pandas as pd
import torch
import jax
import jax.numpy as jnp
from jax import grad, hessian
import torch.nn as nn
import torch.optim as optim
from jax import random
import optax
import jax_dataloader as jdl
from jax_dataloader.loaders import DataLoaderJAX
from flax import nnx
from openai import OpenAI
import wandb
import jsonargparse
```

Question 1: W&B Logging and CLI (JAX NNX)

For the linear regression examples with PyTorch we added in linear_regression_pytorch_logging.py logging and a CLI interface — which came for free with PyTorch Lightning.

In this question you will add in some of those features to the linear_regression_jax_nnx.py example.

Question 1.1: Add W&B Logging

Take the `linear_regression_jax_nnx.py` copied below for your convenience and:

1. Setup W&B properly
2. Add in logging of the `train_loss` at every step of the optimizer
3. Remove the other epoch printing, or try to log an epoch specific `||theta - theta_hat||` if you wish
4. Log the end `||theta - theta_hat||` at the end of the training

```
# MODIFY CODE HERE
import jax
import jax.numpy as jnp
from jax import random
import optax
import jax_dataloader as jdl
from jax_dataloader.loaders import DataLoaderJAX
from flax import nnx

N = 500  # samples
M = 2
sigma = 0.001
rngs = nnx.Rngs(42)
theta = random.normal(rngs(), (M,))
X = random.normal(rngs(), (N, M))
Y = X @ theta + sigma * random.normal(rngs(), (N,))  # Adding noise

def residual(model, x, y):
    y_hat = model(x)
    return (y_hat - y) ** 2

def residuals_loss(model, X, Y):
    return jnp.mean(
        jax.vmap(residual, in_axes=(None, 0, 0))(
            model, X, Y
        )
    )

model = nnx.Linear(M, 1, use_bias=False, rngs=rngs)

lr = 0.001
optimizer = nnx.Optimizer(
    model, optax.sgd(lr), wrt=nnx.Param
```

```
)

@nnx.jit
def train_step(model, optimizer, X, Y):
    def loss_fn(model):
        return residuals_loss(model, X, Y)
    loss, grads = nnx.value_and_grad(loss_fn)(model)
    optimizer.update(model, grads)
    return loss


num_epochs = 1000
batch_size = 512
dataset = jdl.ArrayDataset(X, Y)
train_loader = DataLoaderJAX(
    dataset, batch_size=batch_size, shuffle=True
)
for epoch in range(num_epochs):
    for X_batch, Y_batch in train_loader:
        loss = train_step(
            model, optimizer, X_batch, Y_batch
        )

    if epoch % 100 == 0:
        err = jnp.linalg.norm(
            theta - jnp.squeeze(model.kernel)
        )
        print(
            f"Epoch {epoch},"
            f"||theta - theta_hat|| = {err}"
        )

err = jnp.linalg.norm(
    theta - jnp.squeeze(model.kernel)
)
print(f"||theta - theta_hat|| = {err}")
```

```
import jax
import jax.numpy as jnp
from jax import random
import optax
import jax_dataloader as jdl
```

```
from jax_dataloader.loaders import DataLoaderJAX
from flax import nnx
import wandb

N = 500  # samples
M = 2
sigma = 0.001
rngs = nnx.Rngs(42)
theta = random.normal(rngs(), (M,))
X = random.normal(rngs(), (N, M))
Y = X @ theta + sigma * random.normal(rngs(), (N,))  # Adding noise

def residual(model, x, y):
    y_hat = model(x)
    return (y_hat - y) ** 2

def residuals_loss(model, X, Y):
    return jnp.mean(
        jax.vmap(
            residual, in_axes=(None, 0, 0)
        )(model, X, Y)
    )

model = nnx.Linear(
    M, 1, use_bias=False, rngs=rngs
)

lr = 0.001
optimizer = nnx.Optimizer(
    model, optax.sgd(lr), wrt=nnx.Param
)

@nnx.jit
def train_step(model, optimizer, X, Y):
    def loss_fn(model):
        return residuals_loss(model, X, Y)
    loss, grads = nnx.value_and_grad(loss_fn)(model)
    optimizer.update(model, grads)
    return loss

wandb.init(
    project="econ622_ps4", mode="offline"
```

```
)

num_epochs = 1000
batch_size = 512
dataset = jdl.ArrayDataset(X, Y)
train_loader = DataLoaderJAX(
    dataset, batch_size=batch_size, shuffle=True
)
for epoch in range(num_epochs):
    for X_batch, Y_batch in train_loader:
        loss = train_step(
            model, optimizer, X_batch, Y_batch
        )
        wandb.log({"train_loss": float(loss)})

theta_error = float(
    jnp.linalg.norm(
        theta - jnp.squeeze(model.kernel)
    )
)
wandb.log({"final_theta_error": theta_error})
print(f"||theta - theta_hat|| = {theta_error}")
wandb.finish()
```

Question 1.2: Add CLI Interface

Now, take the above code and copy it into a file named `linear_regression_jax_cli.py`.

We want to make it CLI ready:

- A package with many features, most of which you wouldn't use directly, is jsonargparse. Besides the more advanced features like configuration files and the instantiation of classes/etc. as arguments, the main difference will be that it checks the types of arguments and converts them for you using python typehints.
- Alternatively, you can use the builtin Argparse or any other CLI framework

In that case, you can adapt the following code for your `linear_regression_jax_cli.py`:

```
import jsonargparse
def main_fn(lr: float = 0.001, N: int = 100):
    print(f"lr = {lr}, N = {N}")
```

```
if __name__ == "__main__":
     jsonargparse.CLI(main_fn)
```

Using your CLI

In either case, at that point you should be able to call this with `python linear_regression_jax_cli.py` and have it use all of the default values, `python linear_regression_jax_cli.py --N=200` to change them, etc.

Either submit the file as part of the assignment or just paste the code into the notebook.

The key idea is to wrap the training code in a function with typed arguments, then use `jsonargparse.CLI()` to expose those arguments. See mlp_regression_jax_nnx_logging.py for a full working example.

```python
# Save as linear_regression_jax_cli.py
import jax
import jax.numpy as jnp
from jax import random
import optax
import jax_dataloader as jdl
from jax_dataloader.loaders import DataLoaderJAX
from flax import nnx
import wandb
import jsonargparse

def fit_model(
    N: int = 500,
    M: int = 2,
    sigma: float = 0.001,
    lr: float = 0.001,
    num_epochs: int = 1000,
    batch_size: int = 512,
    seed: int = 42,
    wandb_mode: str = "offline",
):
    if wandb_mode != "disabled":
        wandb.init(project="econ622_ps4", mode=wandb_mode)

    rngs = nnx.Rngs(seed)
    theta = random.normal(rngs(), (M,))
    X = random.normal(rngs(), (N, M))
    Y = X @ theta + sigma * random.normal(rngs(), (N,))
```

```
    def residual(model, x, y):
        y_hat = model(x)
        return (y_hat - y) ** 2

    def residuals_loss(model, X, Y):
        return jnp.mean(
            jax.vmap(
                residual, in_axes=(None, 0, 0)
            )(model, X, Y)
        )

    model = nnx.Linear(
        M, 1, use_bias=False, rngs=rngs
    )
    optimizer = nnx.Optimizer(
        model, optax.sgd(lr), wrt=nnx.Param
    )

    @nnx.jit
    def train_step(model, optimizer, X, Y):
        def loss_fn(model):
            return residuals_loss(model, X, Y)
        loss, grads = nnx.value_and_grad(
            loss_fn
        )(model)
        optimizer.update(model, grads)
        return loss

    dataset = jdl.ArrayDataset(X, Y)
    train_loader = DataLoaderJAX(
        dataset,
        batch_size=batch_size,
        shuffle=True,
    )
    for epoch in range(num_epochs):
        for X_batch, Y_batch in train_loader:
            loss = train_step(
                model, optimizer,
                X_batch, Y_batch,
            )
            if wandb_mode != "disabled":
                wandb.log(
```

```
                    {"train_loss": float(loss)}
                )

    theta_error = float(
        jnp.linalg.norm(
            theta - jnp.squeeze(model.kernel)
        )
    )
    print(f"||theta - theta_hat|| = {theta_error}")
    if wandb_mode != "disabled":
        wandb.log({"final_theta_error": theta_error})
        wandb.finish()

if __name__ == "__main__":
    jsonargparse.CLI(fit_model)
```

Run with: `python linear_regression_jax_cli.py --lr=0.01 --N=200 --wandb_mode=disabled`

Question 1.3 (BONUS): W&B Sweep

Given the CLI you can now run a hyperparameter search. For this bonus problem, do a hyperparameter search over the `--lr` argument by following the W&B documentation.

To get you started, your sweep YAML might look something like this:

```
program: linear_regression_jax_cli.py
name: JAX Example
project: linear_regression_jax
description: JAX Sweep
method: random
parameters:
  lr:
    min: 0.0001
    max: 0.01
```

Here the `method` is changed from Bayes to `random` because otherwise we would need to provide a `metric` to optimize over. Feel free to adapt any of these settings.

If you successfully run a sweep then paste in your own YAML file here, and a screenshot of the W&B dashboard showing something about the sweep results.

Save the YAML above as `sweep.yaml`, then run:

```
# From the terminal:
# wandb sweep sweep.yaml
# wandb agent <sweep_id>
```

The sweep will launch multiple runs with different `--lr` values sampled from `[0.0001, 0.01]`. Check the W&B dashboard for parallel coordinates plots and run comparisons.

Question 2: PyTorch Neural Networks

In the repository you have code that does a linear regression with PyTorch: linear_regression_pytorch_sgd.py.

Question 2.1: Shallow MLP

Make a new file that does the same thing, but replace the `nn.Linear(M, 1, bias=False)` with code that gives a neural network with multiple layers. Maybe try:

```
M = 2 # loaded automatically in the code
num_width = 8  # You can hardcode or add to the template/yaml code
nn.Sequential(
    nn.Linear(M, num_width),
    nn.ReLU(),
    nn.Linear(num_width, 1, bias = False)
)
```

```
Sequential(
  (0): Linear(in_features=2, out_features=8, bias=True)
  (1): ReLU()
  (2): Linear(in_features=8, out_features=1, bias=False)
)
```

This is a network with one "hidden" layer. Try this for a very shallow network (e.g. `num_width = 8`) and see if it converges with Adam or SGD. It is OK if it does not! Don't spend too much time with it.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
```

```
from tqdm import tqdm

torch.manual_seed(42)

N = 500
M = 2
sigma = 0.001
theta = torch.randn(M)
X = torch.randn(N, M)
Y = X @ theta + sigma * torch.randn(N)

dataset = TensorDataset(X, Y)
batch_size = 16
train_loader = DataLoader(
    dataset, batch_size=batch_size, shuffle=True
)

def residuals(model, X, Y):
    Y_hat = model(X).squeeze()
    return ((Y_hat - Y) ** 2).mean()

num_width = 8
model = nn.Sequential(
    nn.Linear(M, num_width),
    nn.ReLU(),
    nn.Linear(num_width, 1, bias=False)
)

lr = 0.001
num_epochs = 1000
optimizer = optim.Adam(model.parameters(), lr=lr)

for epoch in tqdm(range(num_epochs), desc="Epochs"):
    for X_batch, Y_batch in train_loader:
        optimizer.zero_grad()
        loss = residuals(model, X_batch, Y_batch)
        loss.backward()
        optimizer.step()

print(f"Final loss: {residuals(model, X, Y).item():.6f}")
```

```
Epochs:   0%|          | 0/1000 [00:00<?, ?it/s]Epochs:   1%|          | 6/1000 [00:00<00:18
```

```
Final loss: 0.000001
```

Question 2.2: Deep MLP

Now replace this with a deeper and wider network by the same pattern. Maybe something like:

```
M = 2 # loaded automatically in the code
num_width = 256  # You can hardcode or add to the template/yaml code
nn.Sequential(
    nn.Linear(M, num_width),
    nn.ReLU(),
    nn.Linear(num_width, num_width),
    nn.ReLU(),
    nn.Linear(num_width, num_width),
    nn.ReLU(),
    nn.Linear(num_width, num_width),
    nn.ReLU(),
    nn.Linear(num_width, 1, bias = False)
)
```

```
Sequential(
  (0): Linear(in_features=2, out_features=256, bias=True)
  (1): ReLU()
  (2): Linear(in_features=256, out_features=256, bias=True)
  (3): ReLU()
  (4): Linear(in_features=256, out_features=256, bias=True)
  (5): ReLU()
  (6): Linear(in_features=256, out_features=256, bias=True)
  (7): ReLU()
  (8): Linear(in_features=256, out_features=1, bias=False)
)
```

Try to optimize this now and see if it fits well with this deeper network. If this is too slow, change the `num_width` or remove a layer to see if it helps.

```
torch.manual_seed(42)

N = 500
M = 2
sigma = 0.001
theta = torch.randn(M)
X = torch.randn(N, M)
Y = X @ theta + sigma * torch.randn(N)

dataset = TensorDataset(X, Y)
batch_size = 16
train_loader = DataLoader(
    dataset, batch_size=batch_size, shuffle=True
)

num_width = 256
model_deep = nn.Sequential(
    nn.Linear(M, num_width),
    nn.ReLU(),
    nn.Linear(num_width, num_width),
    nn.ReLU(),
    nn.Linear(num_width, num_width),
    nn.ReLU(),
    nn.Linear(num_width, num_width),
    nn.ReLU(),
    nn.Linear(num_width, 1, bias=False)
)

lr = 0.001
num_epochs = 1000
optimizer = optim.Adam(model_deep.parameters(), lr=lr)

for epoch in tqdm(range(num_epochs), desc="Epochs"):
    for X_batch, Y_batch in train_loader:
        optimizer.zero_grad()
        loss = residuals(model_deep, X_batch, Y_batch)
        loss.backward()
        optimizer.step()

print(f"Final loss: {residuals(model_deep, X, Y).item():.6f}")
```

```
Epochs:   0%|          | 0/1000 [00:00<?, ?it/s]Epochs:   0%|          | 2/1000 [00:00<00:57
```

```
Final loss: 0.000005
```

Question 2.3 (BONUS): Nonlinear DGP

The above is trying to fit a linear DGP. Instead, increase the dimension `M` to something much larger, and modify the DGP to be something nonlinear. Try out the larger network to see if it fits it well.

```
torch.manual_seed(42)

M_large = 10
N = 2000
sigma = 0.01
X_nl = torch.randn(N, M_large)
# Nonlinear DGP: Y = sin(X @ w1) + (X @ w2)^2 + noise
w1 = torch.randn(M_large)
w2 = torch.randn(M_large)
Y_nl = (
    torch.sin(X_nl @ w1)
    + (X_nl @ w2) ** 2
    + sigma * torch.randn(N)
)

dataset_nl = TensorDataset(X_nl, Y_nl)
train_loader_nl = DataLoader(
    dataset_nl, batch_size=64, shuffle=True
)

num_width = 256
model_nl = nn.Sequential(
    nn.Linear(M_large, num_width),
    nn.ReLU(),
    nn.Linear(num_width, num_width),
    nn.ReLU(),
    nn.Linear(num_width, num_width),
    nn.ReLU(),
    nn.Linear(num_width, 1, bias=False)
)
```

```
optimizer = optim.Adam(model_nl.parameters(), lr=0.001)
for epoch in tqdm(range(2000), desc="Epochs"):
    for X_batch, Y_batch in train_loader_nl:
        optimizer.zero_grad()
        loss = residuals(model_nl, X_batch, Y_batch)
        loss.backward()
        optimizer.step()

final_loss = residuals(model_nl, X_nl, Y_nl)
print(f"Final training loss: {final_loss.item():.6f}")
```

```
Epochs:   0%|          | 0/2000 [00:00<?, ?it/s]Epochs:   0%|          | 2/2000 [00:00<02:24
```

```
Final training loss: 0.075248
```

Question 3 (BONUS): JAX NNX Neural Networks

Now we can try the same thing with JAX and NNX using linear_regression_jax_nnx.py.

Question 3.1

Take the code and modify the network from the simple `nnx.Linear(M, 1, use_bias=False, rngs=rngs)` to do a nonlinear function with more parameters and layers, as in Question 2.2.

There is no builtin MLP in `nnx`, but you can construct it manually by creating a class from `nnx.Module` and then nesting calls to `nnx.Linear` with an activation like `nnx.relu`. See the docs for more information.

```
import jax
import jax.numpy as jnp
import numpy as np
from jax import random
import optax
import jax_dataloader as jdl
from jax_dataloader.loaders import DataLoaderJAX
from flax import nnx

class MLP(nnx.Module):
```

```
    def __init__(
        self, din: int, dout: int,
        width: int, *, rngs: nnx.Rngs,
    ):
        self.linear1 = nnx.Linear(
            din, width, rngs=rngs
        )
        self.linear2 = nnx.Linear(
            width, width, rngs=rngs
        )
        self.linear3 = nnx.Linear(
            width, dout, rngs=rngs
        )

    def __call__(self, x: jax.Array):
        x = self.linear1(x)
        x = nnx.relu(x)
        x = self.linear2(x)
        x = nnx.relu(x)
        x = self.linear3(x)
        return x

N = 500
M = 2
sigma = 0.001
rngs = nnx.Rngs(42)
theta = random.normal(rngs(), (M,))
X = random.normal(rngs(), (N, M))
Y = X @ theta + sigma * random.normal(rngs(), (N,))

def residual(model, x, y):
    y_hat = model(x)
    return (y_hat - y) ** 2

def residuals_loss(model, X, Y):
    return jnp.mean(
        jax.vmap(
            residual, in_axes=(None, 0, 0)
        )(model, X, Y)
    )

model = MLP(M, 1, 128, rngs=rngs)
```

```
n_params = sum(
    np.prod(x.shape)
    for x in jax.tree.leaves(
        nnx.state(model, nnx.Param)
    )
)
print(f"Number of parameters: {n_params}")

lr = 0.001
optimizer = nnx.Optimizer(
    model, optax.sgd(lr), wrt=nnx.Param
)

@nnx.jit
def train_step(model, optimizer, X, Y):
    def loss_fn(model):
        return residuals_loss(model, X, Y)
    loss, grads = nnx.value_and_grad(loss_fn)(model)
    optimizer.update(model, grads)
    return loss

num_epochs = 2000
batch_size = 512
dataset = jdl.ArrayDataset(X, Y)
train_loader = DataLoaderJAX(
    dataset, batch_size=batch_size, shuffle=True
)
for epoch in range(num_epochs):
    for X_batch, Y_batch in train_loader:
        loss = train_step(
            model, optimizer, X_batch, Y_batch
        )

    if epoch % 200 == 0:
        print(f"Epoch {epoch}, loss {loss}")

N_test = 200
X_test = random.normal(rngs(), (N_test, M))
Y_test = (
    X_test @ theta
    + sigma * random.normal(rngs(), (N_test,))
)
```

```
train_loss = residuals_loss(model, X, Y)
test_loss = residuals_loss(model, X_test, Y_test)
print(
    f"Train loss: {train_loss},"
    f" Test loss: {test_loss}"
)
```

```
Number of parameters: 17025
Epoch 0, loss 0.5089762210845947
Epoch 200, loss 0.0018912514206022024
Epoch 400, loss 0.0010070333955809474
Epoch 600, loss 0.0007703547598794103
Epoch 800, loss 0.0006354043143801391
Epoch 1000, loss 0.0005389087600633502
Epoch 1200, loss 0.0004636280603706837
Epoch 1400, loss 0.0004101044032722711 6
Epoch 1600, loss 0.0003651242004707455 6
Epoch 1800, loss 0.0003286586434114724 4
Train loss: 0.0002981825964525342, Test loss: 0.00027129799127578735
```

Question 4: GP Regression (GPyTorch)

The following code comes from the GPyTorch documentation.

```
import math
import torch
import gpytorch
from gpytorch.kernels import ScaleKernel, RBFKernel, LinearKernel
from matplotlib import pyplot as plt
# Training data is 100 points in [0,1] inclusive regularly spaced
train_x = torch.linspace(0, 1, 100)
# True function is sin(2*pi*x) with Gaussian noise
train_y = (
    torch.sin(train_x * (2 * math.pi))
    + torch.randn(train_x.size()) * math.sqrt(0.04)
)

# Simplest form of GP model, exact inference
class ExactGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(
```

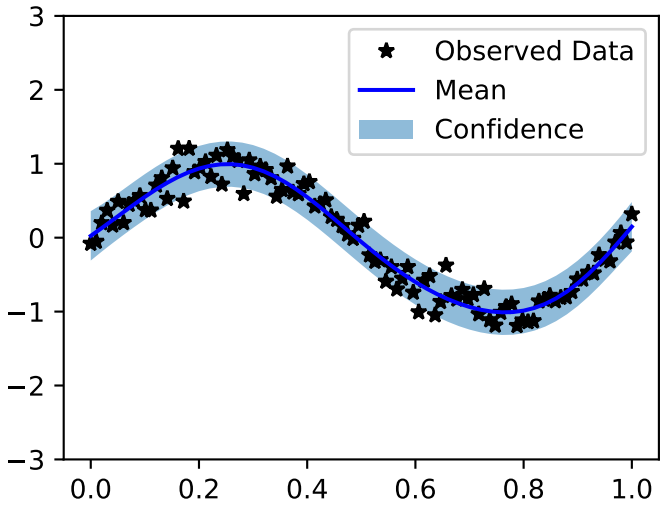
```
            train_x, train_y, likelihood
        )
        self.mean_module = (
            gpytorch.means.ConstantMean()
        )
        self.covar_module = (
            gpytorch.kernels.ScaleKernel(
                gpytorch.kernels.RBFKernel()
            )
        )

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return (
            gpytorch.distributions
            .MultivariateNormal(mean_x, covar_x)
        )

# initialize likelihood and model
likelihood = (
    gpytorch.likelihoods.GaussianLikelihood()
)
model = ExactGPModel(
    train_x, train_y, likelihood
)
model.train()
likelihood.train()

# Use the adam optimizer
optimizer = torch.optim.Adam(
    model.parameters(), lr=0.1
)

# "Loss" for GPs - the marginal log likelihood
mll = gpytorch.mlls.ExactMarginalLogLikelihood(
    likelihood, model
)

training_iter = 50
for i in range(training_iter):
    optimizer.zero_grad()
```

```
    output = model(train_x)
    loss = -mll(output, train_y)
    loss.backward()
    ls = (
        model.covar_module
        .base_kernel.lengthscale.item()
    )
    ns = model.likelihood.noise.item()
    print(
        f"Iter {i+1}/{training_iter}"
        f" - Loss: {loss.item():.3f}"
        f"   lengthscale: {ls:.3f}"
        f"   noise: {ns:.3f}"
    )
    optimizer.step()

# Get into evaluation (predictive posterior) mode
model.eval()
likelihood.eval()

# Make predictions
with (
    torch.no_grad(),
    gpytorch.settings.fast_pred_var(),
):
    test_x = torch.linspace(0, 1, 51)
    observed_pred = likelihood(model(test_x))

with torch.no_grad():
    f, ax = plt.subplots(1, 1, figsize=(4, 3))

    lower, upper = (
        observed_pred.confidence_region()
    )
    ax.plot(
        train_x.numpy(), train_y.numpy(), 'k*'
    )
    ax.plot(
        test_x.numpy(),
        observed_pred.mean.numpy(), 'b',
    )
    ax.fill_between(
```

```
        test_x.numpy(),
        lower.numpy(), upper.numpy(),
        alpha=0.5,
    )
    ax.set_ylim([-3, 3])
    ax.legend(
        ['Observed Data', 'Mean', 'Confidence']
    )
```

```
Iter 1/50 - Loss: 0.936   lengthscale: 0.693   noise: 0.693
Iter 2/50 - Loss: 0.905   lengthscale: 0.644   noise: 0.644
Iter 3/50 - Loss: 0.870   lengthscale: 0.598   noise: 0.598
Iter 4/50 - Loss: 0.832   lengthscale: 0.555   noise: 0.554
Iter 5/50 - Loss: 0.788   lengthscale: 0.514   noise: 0.513
Iter 6/50 - Loss: 0.739   lengthscale: 0.476   noise: 0.474
Iter 7/50 - Loss: 0.686   lengthscale: 0.439   noise: 0.437
Iter 8/50 - Loss: 0.632   lengthscale: 0.405   noise: 0.402
Iter 9/50 - Loss: 0.581   lengthscale: 0.372   noise: 0.369
Iter 10/50 - Loss: 0.534   lengthscale: 0.342   noise: 0.339
Iter 11/50 - Loss: 0.491   lengthscale: 0.315   noise: 0.310
Iter 12/50 - Loss: 0.452   lengthscale: 0.292   noise: 0.284
Iter 13/50 - Loss: 0.414   lengthscale: 0.272   noise: 0.259
Iter 14/50 - Loss: 0.378   lengthscale: 0.256   noise: 0.236
Iter 15/50 - Loss: 0.342   lengthscale: 0.243   noise: 0.215
Iter 16/50 - Loss: 0.306   lengthscale: 0.232   noise: 0.196
Iter 17/50 - Loss: 0.270   lengthscale: 0.224   noise: 0.178
Iter 18/50 - Loss: 0.234   lengthscale: 0.218   noise: 0.162
Iter 19/50 - Loss: 0.198   lengthscale: 0.213   noise: 0.147
Iter 20/50 - Loss: 0.163   lengthscale: 0.211   noise: 0.133
Iter 21/50 - Loss: 0.128   lengthscale: 0.209   noise: 0.121
Iter 22/50 - Loss: 0.093   lengthscale: 0.210   noise: 0.110
Iter 23/50 - Loss: 0.059   lengthscale: 0.211   noise: 0.100
Iter 24/50 - Loss: 0.026   lengthscale: 0.213   noise: 0.090
Iter 25/50 - Loss: -0.006   lengthscale: 0.217   noise: 0.082
Iter 26/50 - Loss: -0.036   lengthscale: 0.221   noise: 0.074
Iter 27/50 - Loss: -0.065   lengthscale: 0.226   noise: 0.068
Iter 28/50 - Loss: -0.091   lengthscale: 0.232   noise: 0.062
Iter 29/50 - Loss: -0.115   lengthscale: 0.239   noise: 0.056
Iter 30/50 - Loss: -0.137   lengthscale: 0.245   noise: 0.051
Iter 31/50 - Loss: -0.155   lengthscale: 0.252   noise: 0.047
Iter 32/50 - Loss: -0.171   lengthscale: 0.259   noise: 0.043
Iter 33/50 - Loss: -0.183   lengthscale: 0.265   noise: 0.039
```

```
Iter 34/50 - Loss: -0.193   lengthscale: 0.271   noise: 0.036
Iter 35/50 - Loss: -0.199   lengthscale: 0.276   noise: 0.034
Iter 36/50 - Loss: -0.203   lengthscale: 0.280   noise: 0.031
Iter 37/50 - Loss: -0.204   lengthscale: 0.283   noise: 0.029
Iter 38/50 - Loss: -0.204   lengthscale: 0.284   noise: 0.028
Iter 39/50 - Loss: -0.202   lengthscale: 0.284   noise: 0.026
Iter 40/50 - Loss: -0.200   lengthscale: 0.283   noise: 0.025
Iter 41/50 - Loss: -0.197   lengthscale: 0.280   noise: 0.024
Iter 42/50 - Loss: -0.194   lengthscale: 0.276   noise: 0.023
Iter 43/50 - Loss: -0.192   lengthscale: 0.271   noise: 0.022
Iter 44/50 - Loss: -0.190   lengthscale: 0.266   noise: 0.022
Iter 45/50 - Loss: -0.189   lengthscale: 0.261   noise: 0.021
Iter 46/50 - Loss: -0.188   lengthscale: 0.255   noise: 0.021
Iter 47/50 - Loss: -0.188   lengthscale: 0.249   noise: 0.021
Iter 48/50 - Loss: -0.189   lengthscale: 0.244   noise: 0.021
Iter 49/50 - Loss: -0.190   lengthscale: 0.239   noise: 0.022
Iter 50/50 - Loss: -0.191   lengthscale: 0.235   noise: 0.022
```



Question 4.1: Different Kernel

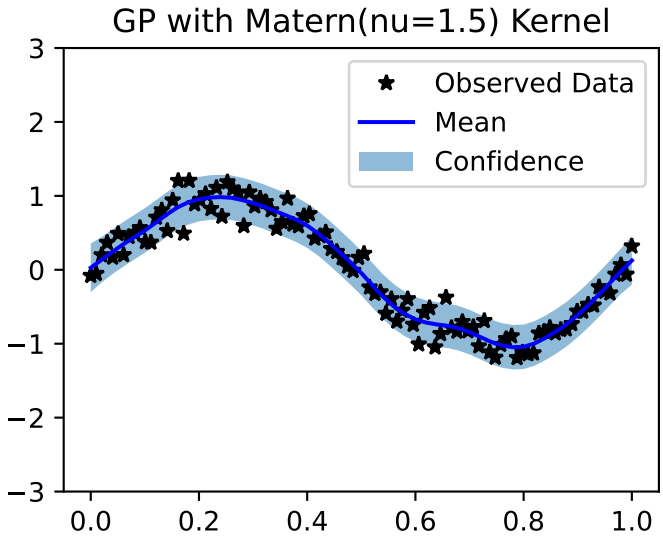
Using the code above, try a different kernel from the docs and plot the results. Doesn't matter if it is better or worse, but you should try to pick a kernel that has different parameters to fit.

```
class ExactGPModelMatern(gpytorch.models.ExactGP):
    def __init__(
        self, train_x, train_y, likelihood
    ):
        super().__init__(
            train_x, train_y, likelihood
        )
        self.mean_module = (
            gpytorch.means.ConstantMean()
        )
        self.covar_module = (
            gpytorch.kernels.ScaleKernel(
                gpytorch.kernels.MaternKernel(
                    nu=1.5
                )
            )
        )

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return (
            gpytorch.distributions
            .MultivariateNormal(mean_x, covar_x)
        )

likelihood_m = (
    gpytorch.likelihoods.GaussianLikelihood()
)
model_m = ExactGPModelMatern(
    train_x, train_y, likelihood_m
)
model_m.train()
likelihood_m.train()

optimizer_m = torch.optim.Adam(
    model_m.parameters(), lr=0.1
)
mll_m = (
    gpytorch.mlls.ExactMarginalLogLikelihood(
        likelihood_m, model_m
    )
```

```
)

for i in range(50):
    optimizer_m.zero_grad()
    output = model_m(train_x)
    loss = -mll_m(output, train_y)
    loss.backward()
    if (i + 1) % 10 == 0:
        ls = (
            model_m.covar_module
            .base_kernel.lengthscale.item()
        )
        ns = model_m.likelihood.noise.item()
        print(
            f"Iter {i+1}/50"
            f" - Loss: {loss.item():.3f}"
            f"   lengthscale: {ls:.3f}"
            f"   noise: {ns:.3f}"
        )
    optimizer_m.step()

model_m.eval()
likelihood_m.eval()

with (
    torch.no_grad(),
    gpytorch.settings.fast_pred_var(),
):
    test_x = torch.linspace(0, 1, 51)
    observed_pred_m = likelihood_m(
        model_m(test_x)
    )

with torch.no_grad():
    f, ax = plt.subplots(1, 1, figsize=(4, 3))
    lower, upper = (
        observed_pred_m.confidence_region()
    )
    ax.plot(
        train_x.numpy(), train_y.numpy(), 'k*'
    )
    ax.plot(
```

```
        test_x.numpy(),
        observed_pred_m.mean.numpy(), 'b',
    )
    ax.fill_between(
        test_x.numpy(),
        lower.numpy(), upper.numpy(),
        alpha=0.5,
    )
    ax.set_ylim([-3, 3])
    ax.legend(
        ['Observed Data', 'Mean', 'Confidence']
    )
    ax.set_title(
        'GP with Matern(nu=1.5) Kernel'
    )
```

```
Iter 10/50 - Loss: 0.546   lengthscale: 0.371   noise: 0.340
Iter 20/50 - Loss: 0.181   lengthscale: 0.390   noise: 0.137
Iter 30/50 - Loss: -0.106   lengthscale: 0.384   noise: 0.054
Iter 40/50 - Loss: -0.196   lengthscale: 0.370   noise: 0.025
Iter 50/50 - Loss: -0.179   lengthscale: 0.356   noise: 0.020
```



The Matern kernel with $\nu = 1.5$ produces slightly less smooth predictions than the RBF kernel, since the RBF corresponds to $\nu \to \infty$.

Question 4.2 (BONUS): Noiseless GP Interpolation

Now take the code above:

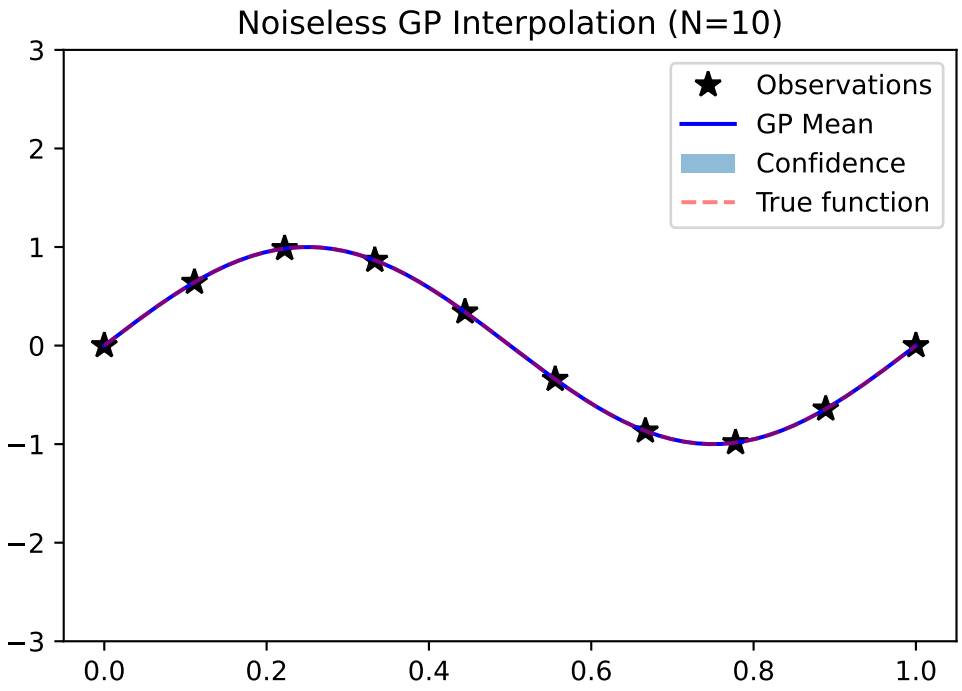
1. Remove the noise in the DGP
2. Decrease the number of generated datapoints to maybe 10 or so.
3. Try to see how to fit a GP without any observational noise, so that it interpolates the data. This may require changing the likelihood object in the loop above.

```
N_pts = 10
train_x_small = torch.linspace(0, 1, N_pts)
train_y_small = torch.sin(
    train_x_small * (2 * math.pi)
)  # no noise

# Relaxed noise constraint for near-zero noise
likelihood_noiseless = (
    gpytorch.likelihoods.GaussianLikelihood(
        noise_constraint=(
            gpytorch.constraints.GreaterThan(
                1e-8
            )
        )
    )
)
# very small noise for numerical stability
likelihood_noiseless.noise = 1e-6
# Freeze noise so optimizer doesn't change it
likelihood_noiseless.noise_covar.raw_noise.requires_grad = False

model_noiseless = ExactGPModel(
    train_x_small,
    train_y_small,
    likelihood_noiseless,
)
model_noiseless.train()
likelihood_noiseless.train()

optimizer_nl = torch.optim.Adam(
    model_noiseless.parameters(), lr=0.1
)
mll_nl = (
    gpytorch.mlls.ExactMarginalLogLikelihood(
```

```
        likelihood_noiseless, model_noiseless
    )
)

for i in range(100):
    optimizer_nl.zero_grad()
    output = model_noiseless(train_x_small)
    loss = -mll_nl(output, train_y_small)
    loss.backward()
    optimizer_nl.step()

model_noiseless.eval()
likelihood_noiseless.eval()

with (
    torch.no_grad(),
    gpytorch.settings.fast_pred_var(),
):
    test_x = torch.linspace(0, 1, 200)
    observed_pred_nl = likelihood_noiseless(
        model_noiseless(test_x)
    )

with torch.no_grad():
    f, ax = plt.subplots(1, 1, figsize=(6, 4))
    lower, upper = (
        observed_pred_nl.confidence_region()
    )
    ax.plot(
        train_x_small.numpy(),
        train_y_small.numpy(),
        'k*', markersize=10,
    )
    ax.plot(
        test_x.numpy(),
        observed_pred_nl.mean.numpy(), 'b',
    )
    ax.fill_between(
        test_x.numpy(),
        lower.numpy(), upper.numpy(),
        alpha=0.5,
    )
```

```
    true_fn = torch.sin(
        test_x * 2 * math.pi
    ).numpy()
    ax.plot(
        test_x.numpy(), true_fn,
        'r--', alpha=0.5,
        label='True function',
    )
    ax.set_ylim([-3, 3])
    ax.legend(
        ['Observations', 'GP Mean',
         'Confidence', 'True function']
    )
    ax.set_title(
        'Noiseless GP Interpolation (N=10)'
    )
```

```
/home/runner/work/grad_econ_ML/grad_econ_ML/.venv/lib/python3.13/site-packages/gpytorch/distri
06.
  warnings.warn(
```



The key insight is constraining the likelihood noise to a very small value so the GP interpolates (passes through) the training points rather than smoothing over them.

Question 5: OpenAI API

Setting up OpenAI to access ChatGPT and embeddings programmatically.

Question 5.1: Setup and First Call

Setup an account on the OpenAI platform:

- Sign up
- Go to the `API keys` tab and create a key
- In your terminal, set `OPENAI_API_KEY` to this value (see here)

Given the change in the environment variable, you may need to restart your browser/editor/etc.

Get the following code running, which shows a prompt:

```
client = OpenAI()
response = client.responses.create(
    model="gpt-4o-mini",
    temperature=0.7,
    instructions=(
        "You are generating numbers"
        " that are easy to parse."
    ),
    input="Give me a list of 3 numbers",
)
print(response.output_text)
```

This is a setup step — just run the code above after setting the `OPENAI_API_KEY` environment variable. If it prints a list of numbers, you're done.

Question 5.2: Temperature Exploration

Take that prompt and run it a bunch of times with `temperature = 0.0`. What happens and why?

Next, run it a bunch of times with `temperature = 2.0`, which is the maximum allowed by the API. What happens and why?

```
client = OpenAI()

print("=== temperature = 0.0 (deterministic) ===")
for i in range(3):
    response = client.responses.create(
        model="gpt-4o-mini",
        temperature=0.0,
        instructions=(
            "You are generating numbers"
            " that are easy to parse."
        ),
        input="Give me a list of 3 numbers",
    )
    print(
        f"  Run {i+1}: {response.output_text}"
    )

print("\n=== temperature = 2.0 (max allowed) ===")
for i in range(3):
    response = client.responses.create(
        model="gpt-4o-mini",
        temperature=2.0,
        instructions=(
            "You are generating numbers"
            " that are easy to parse."
        ),
        input="Give me a list of 3 numbers",
    )
    print(
        f"  Run {i+1}: {response.output_text}"
    )
```

Explanation: With `temperature = 0.0`, the model always picks the highest-probability token at each step, so the output is nearly deterministic — you get the same (or very similar) list each time. With `temperature = 2.0`, the softmax distribution is flattened dramatically, making low-probability tokens much more likely. The output becomes erratic and sometimes incoherent, as the model samples from an almost-uniform distribution over the vocabulary.

Question 5.3: Parsing LLM Output

Modify the `instructions` in that prompt until you can easily parse the `response.output_text` into a list, reliably, with `temperature = 0.7`.

```
client = OpenAI()
response = client.responses.create(
    model="gpt-4o-mini",
    temperature=0.7,
    instructions(  # modify this
        "You are generating numbers"
        " that are easy to parse."
    ),
    input="Give me a list of 3 numbers",
)
print(response.output_text)
# Add parsing logic
```

```
import json

client = OpenAI()
response = client.responses.create(
    model="gpt-4o-mini",
    temperature=0.7,
    instructions(
        "You must respond with ONLY"
        " a JSON array of numbers."
        " No text, no explanation,"
        " just the JSON array."
        " Example: [1, 2, 3]"
    ),
    input="Give me a list of 3 numbers",
    text={
        "format": {"type": "json_object"}
    },
)
raw = response.output_text
print(f"Raw output: {raw}")
numbers = json.loads(raw)
print(f"Parsed list: {numbers}")
print(
    f"Type: {type(numbers)},"
    f" Length: {len(numbers)}"
)
```

The Responses API supports structured output via `text={"format": {"type": "json_object"}}`, which guarantees valid JSON. Combined with clear instructions,

`json.loads()` reliably parses the response.