Mount Google Drive and Load Data

In this step, you'll mount your Google Drive so that you can access .npy files directly from it. Then, load your training, validation, and test sets using NumPy. Be sure to update the paths to point to your own data files.

Make sure to set to runtime session to use GPU otherwise your training is going to be slow.

```
# import files and mount google drive
import numpy as np
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force
```

```
# Step 1: Load your .npy files from your Google Drive
# TODO: Replace the paths below with the correct paths to your own files

X_train = np.load('/content/drive/MyDrive/ML remotesensing/HW5/data/landis_chlorophyl_regression_train.npy')
y_train = np.load('/content/drive/MyDrive/ML remotesensing/HW5/data/landis_chlorophyl_regression_traingt.npy')

X_val = np.load('/content/drive/MyDrive/ML remotesensing/HW5/data/landis_chlorophyl_regression_val.npy')
y_val = np.load('/content/drive/MyDrive/ML remotesensing/HW5/data/landis_chlorophyl_regression_valgt.npy')

X_test = np.load('/content/drive/MyDrive/ML remotesensing/HW5/data/landis_chlorophyl_regression_test.npy')
y_test = np.load('/content/drive/MyDrive/ML remotesensing/HW5/data/landis_chlorophyl_regression_testgt.npy')

print("Training set shape:", X_train.shape)
```

```
Training set shape: (6000, 425)
```

Standardize the Data & Create Dataset

Here, you will standardize your dataset using StandardScaler. This ensures that each feature has zero mean and unit variance. Remember to fit the scaler only on the training data, and then transform all splits using that same scaler.

You'll define a custom PyTorch Dataset to wrap your data and make it compatible with DataLoader. After that, instantiate the datasets and set up data loaders with appropriate batch size and parallel loading settings (num_workers, prefetch_factor).

```
import torch
from torch.utils.data import Dataset
from sklearn.preprocessing import StandardScaler

# Step 1: Standard scaling based on training data only
# TODO: Scale your validation and test sets using statistics from the training set only
scaler = StandardScaler()
scaler.fit(X_train)

X_train_scaled = scaler.transform(X_train)        # COMPLETE ME
X_val_scaled = scaler.transform(X_val) # COMPLETE ME
X_test_scaled = scaler.transform(X_test) # COMPLETE ME

# Step 2: Create custom dataset class
class ChlorophyllDataset(Dataset):
    def __init__(self, X, y):
      # TODO: Convert your numpy arrays to torch tensors with the correct dtype
        self.X = torch.tensor(X, dtype=torch.float32)   # COMPLETE ME
        self.y = torch.tensor(y, dtype=torch.float32)   # COMPLETE ME

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

# Step 3: Instantiate your datasets
train_dataset = ChlorophyllDataset(X_train_scaled, y_train)
val_dataset = ChlorophyllDataset(X_val_scaled, y_val)
test_dataset = ChlorophyllDataset(X_test_scaled, y_test)

# Step 4: Create DataLoaders
# TODO: Set batch_size, num_workers, and prefetch_factor based on your system
from torch.utils.data import DataLoader
batch_size = 64 # COMPLETE ME
```

```
num_workers = 2 # COMPLETE ME
prefetch_factor = 2 # COMPLETE ME
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
                          num_workers=num_workers, prefetch_factor=prefetch_factor)# COMPLETE ME
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,
                        num_workers=num_workers, prefetch_factor=prefetch_factor)   # COMPLETE ME
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,
                         num_workers=num_workers, prefetch_factor=prefetch_factor)# COMPLETE ME

print("train dataset size is:", len(train_dataset))
print("val dataset size is:", len(val_dataset))
print("test dataset size is:", len(test_dataset))
```

```
train dataset size is: 6000
val dataset size is: 2000
test dataset size is: 2000
```

Define Your 1D Convolutional Model

In this step, you'll define a custom SpectralCNN model using PyTorch's nn Module. This model consists of a sequence of 1D convolutional layers with increasing filter sizes and stride to reduce the spectral dimension. The final Linear layer maps the learned features to a single output value for regression.

□ Note: Since your input has shape [batch_size, 425], you'll need to change the shape accordingly.

```
import torch.nn as nn

class SpectralCNN(nn.Module):
    def __init__(self):
        super(SpectralCNN, self).__init__()
        # 8 layer 1D CNN with kernel size 3 and stride of 2
        # stride 2 lowers the dimensionality spectral dimensionality
        # rely is the activation function introducing nonlinearity
        # the bigger the kernel size the more number of parameters to learn
        self.net = nn.Sequential(
            nn.Conv1d(1, 8, kernel_size=3, stride=2), nn.ReLU(),  # [B, 8, 212]
            nn.Conv1d(8, 16, kernel_size=3, stride=2), nn.ReLU(), # [B, 16, 105]
            nn.Conv1d(16, 32, kernel_size=3, stride=2), nn.ReLU(),# [B, 32, 52]
            nn.Conv1d(32, 64, kernel_size=3, stride=2), nn.ReLU(),# [B, 64, 25]
            nn.Conv1d(64, 128, kernel_size=3, stride=2), nn.ReLU(),# [B, 128, 12]
            nn.Conv1d(128, 128, kernel_size=3, stride=2), nn.ReLU(),# [B, 128, 5]
            nn.Conv1d(128, 256, kernel_size=3, stride=2), nn.ReLU(),# [B, 256, 2]
            nn.Conv1d(256, 256, kernel_size=2, stride=1), nn.ReLU(),# [B, 256, 1]
            nn.Flatten(),
            # we are performing regression so output is one value
            nn.Linear(256, 1)
        )

    def forward(self, x):

        # TODO: x is of siz Nx425, however, we need to define the number of channels for our 1D data
        # in this case there is only 1 channel/feature for our 425 input features
        # in 2D sp9ace we have N x C_in x H x W.
        # in 1D space we should have N x C_in x Length
        # modify the retun correctly to reflect this
        x = x.unsqueeze(1)
        # print(x.shape)
        out = self.net(x)

        return out
```

Define the Model, Optimizer, and Training Configuration

In this section, you will define the loss function for your regression task, instantiate your model, and set up the optimizer of your choice. Make sure to use weight decay to prevent overfitting.

Your training will benefit from learning rate scheduler such as to gradually reduce the learning rate during training. Finally, you'll move your model to the GPU if available.

□ Tip: Your learning rate and weight decay are hyperparameters!

```
import torch.optim as optim

# TODO: Define regression LOSS
criterion = nn.MSELoss()   # COMPLETE ME
```

```
# instantiating the model
model = SpectralCNN()

# TODO: define optimizer, suggestion is on Adam with weight decay.
# keep in mind weight decay and learning rate are hyperpareters
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5) # COMPLETE ME

# TODO: you can define a learning rate scheduler at the time of training
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=25, gamma=0.1) # COMPLETE ME

# Optional: use GPU if available
# make sure to change your runtime type to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print("Using:", device)
```

```
Using: cuda
```

Training Loop and Early Stopping

In this section, you will implement the full training loop for your model. For each epoch, you'll: • Loop through your training data • Send batches to the correct device • Perform forward and backward passes • Update the model using your optimizer

After training on each epoch, you'll evaluate your model on the validation set without computing gradients. You'll track the best-performing model based on validation loss and save it using torch.save(). Finally, you'll step the learning rate scheduler to gradually reduce the learning rate over time.

Make sure to plot loss for training and validation sets, this will help you see into overfitting and underfitting.

□ Don't forget: only update weights during training, and use torch.no_grad() during validation to save memory and speed up evaluation.

```
# Initialize variables for early stopping
best_val_loss = float("inf")
best_epoch = -1

# keepong track of validation loss COMPLETE ME
val_loss_history = []

# Training loop
epochs = 50
for epoch in range(epochs):
    model.train()
    train_losses = []

    # TOOD: keep track of any other regression metric you want per
    train_mae = []

    for X_batch, y_batch in train_loader:
        # TODO: send input features and reference target to device
        # print(X_batch.shape)
        X_batch, y_batch = X_batch.to(device), y_batch.to(device) # COMPLETE ME

        # zeroing out previous step gradients
        optimizer.zero_grad() # COMPLETE ME

        # send input features and reference target to device
        outputs = model(X_batch)

        # TODO: calculate the loss
        loss = criterion(outputs.squeeze(1), y_batch.float()) # COMPLETE ME

        # TODO: calculate the gradients by calling backward on the loss
        loss.backward() # COMPLETE ME

        # TODO: take a step by calling step on the optimizer
        optimizer.step() # COMPLETE ME

        # TODO: keep score of your training loss and any other metric you see fit for
        # your regression task
        train_losses.append(loss.item())
        mae = torch.mean(torch.abs(outputs - y_batch)).item()
        train_mae.append(mae)


    # Validation
    model.eval()
    val_losses = []
    val_mae = []
```

```
        with torch.no_grad():
            for X_batch, y_batch in val_loader:
                X_batch, y_batch = X_batch.to(device), y_batch.to(device) # TODO: Same as training but we are not up
                outputs = model(X_batch) # COMPLETE ME
                val_loss = criterion(outputs.squeeze(1), y_batch.float())
                val_losses.append(val_loss.item()) # COMPLETE ME
                mae = torch.mean(torch.abs(outputs - y_batch)).item()
                val_mae.append(mae)

        # TODO: implementing early stopping here.
        # keep track of the validation loss and save the model parameters
        # when the loss is the lowest for the validation set
        val_loss_mean = np.mean(val_losses) # COMPLETE ME
        val_loss_history.append(val_loss_mean)  # COMPLETE ME
        if val_loss_mean < best_val_loss:
            best_val_loss = val_loss_mean  # COMPLETE ME
            best_epoch = epoch # COMPLETE ME
            torch.save(model.state_dict(), "best_model.pt")

        # TODO: take a step in the scheduler to update the learning rate
        scheduler.step() # COMPLETE ME

        # Print summary
        print(f"Epoch {epoch+1}/{epochs} | Train Loss: {np.mean(train_losses):.4f} | Val Loss: {val_loss_mean:.4f} |

    print(f"\nBest model saved from epoch {best_epoch+1} with Val Loss: {best_val_loss:.4f}")
```

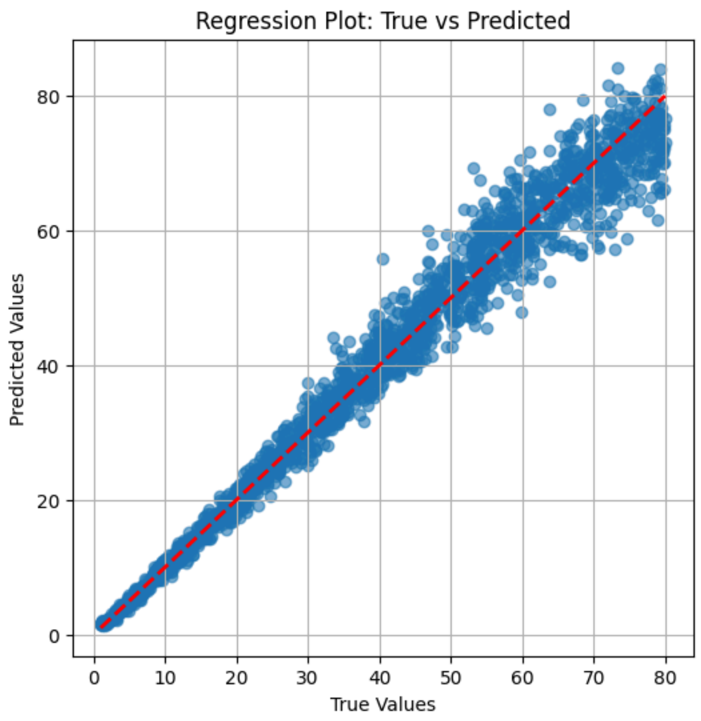
```
Epoch 1/50 | Train Loss: 799.8855 | Val Loss: 556.0847 | LR: 0.001000
Epoch 2/50 | Train Loss: 496.3422 | Val Loss: 511.3169 | LR: 0.001000
Epoch 3/50 | Train Loss: 226.0696 | Val Loss: 41.3746 | LR: 0.001000
Epoch 4/50 | Train Loss: 28.6921 | Val Loss: 20.4006 | LR: 0.001000
Epoch 5/50 | Train Loss: 19.3601 | Val Loss: 17.8575 | LR: 0.001000
Epoch 6/50 | Train Loss: 16.5392 | Val Loss: 14.3975 | LR: 0.001000
Epoch 7/50 | Train Loss: 15.9118 | Val Loss: 14.2134 | LR: 0.001000
Epoch 8/50 | Train Loss: 15.5084 | Val Loss: 13.2105 | LR: 0.001000
Epoch 9/50 | Train Loss: 14.4503 | Val Loss: 12.7840 | LR: 0.001000
Epoch 10/50 | Train Loss: 13.9044 | Val Loss: 12.4843 | LR: 0.001000
Epoch 11/50 | Train Loss: 13.5533 | Val Loss: 12.9343 | LR: 0.001000
Epoch 12/50 | Train Loss: 13.3197 | Val Loss: 13.7620 | LR: 0.001000
Epoch 13/50 | Train Loss: 13.7415 | Val Loss: 20.7861 | LR: 0.001000
Epoch 14/50 | Train Loss: 15.1432 | Val Loss: 12.4307 | LR: 0.001000
Epoch 15/50 | Train Loss: 12.9828 | Val Loss: 11.5435 | LR: 0.001000
Epoch 16/50 | Train Loss: 14.3405 | Val Loss: 13.8754 | LR: 0.001000
Epoch 17/50 | Train Loss: 12.7338 | Val Loss: 14.6874 | LR: 0.001000
Epoch 18/50 | Train Loss: 12.6202 | Val Loss: 12.0765 | LR: 0.001000
Epoch 19/50 | Train Loss: 13.4876 | Val Loss: 11.7175 | LR: 0.001000
Epoch 20/50 | Train Loss: 12.1708 | Val Loss: 13.6569 | LR: 0.001000
Epoch 21/50 | Train Loss: 12.3888 | Val Loss: 17.7044 | LR: 0.001000
Epoch 22/50 | Train Loss: 13.6931 | Val Loss: 12.0352 | LR: 0.001000
Epoch 23/50 | Train Loss: 11.3181 | Val Loss: 11.3196 | LR: 0.001000
Epoch 24/50 | Train Loss: 11.6482 | Val Loss: 12.2065 | LR: 0.001000
Epoch 25/50 | Train Loss: 11.7779 | Val Loss: 13.1914 | LR: 0.000100
Epoch 26/50 | Train Loss: 10.2772 | Val Loss: 11.2465 | LR: 0.000100
Epoch 27/50 | Train Loss: 10.1781 | Val Loss: 11.3011 | LR: 0.000100
Epoch 28/50 | Train Loss: 10.1402 | Val Loss: 11.3679 | LR: 0.000100
Epoch 29/50 | Train Loss: 10.0640 | Val Loss: 11.2739 | LR: 0.000100
Epoch 30/50 | Train Loss: 10.1383 | Val Loss: 11.4829 | LR: 0.000100
Epoch 31/50 | Train Loss: 10.1558 | Val Loss: 11.3700 | LR: 0.000100
Epoch 32/50 | Train Loss: 10.0412 | Val Loss: 11.2948 | LR: 0.000100
Epoch 33/50 | Train Loss: 9.9955 | Val Loss: 11.3583 | LR: 0.000100
Epoch 34/50 | Train Loss: 10.1167 | Val Loss: 11.6939 | LR: 0.000100
Epoch 35/50 | Train Loss: 10.0881 | Val Loss: 11.2935 | LR: 0.000100
Epoch 36/50 | Train Loss: 10.1327 | Val Loss: 11.7825 | LR: 0.000100
Epoch 37/50 | Train Loss: 10.0552 | Val Loss: 11.2866 | LR: 0.000100
Epoch 38/50 | Train Loss: 9.9382 | Val Loss: 11.2009 | LR: 0.000100
Epoch 39/50 | Train Loss: 9.9291 | Val Loss: 11.3024 | LR: 0.000100
Epoch 40/50 | Train Loss: 10.2937 | Val Loss: 11.5942 | LR: 0.000100
Epoch 41/50 | Train Loss: 10.0413 | Val Loss: 11.3876 | LR: 0.000100
Epoch 42/50 | Train Loss: 9.9060 | Val Loss: 11.2538 | LR: 0.000100
Epoch 43/50 | Train Loss: 9.9953 | Val Loss: 11.7985 | LR: 0.000100
Epoch 44/50 | Train Loss: 9.9178 | Val Loss: 11.6984 | LR: 0.000100
Epoch 45/50 | Train Loss: 10.1369 | Val Loss: 11.2966 | LR: 0.000100
Epoch 46/50 | Train Loss: 9.9294 | Val Loss: 11.4817 | LR: 0.000100
Epoch 47/50 | Train Loss: 9.9451 | Val Loss: 11.4100 | LR: 0.000100
Epoch 48/50 | Train Loss: 9.8173 | Val Loss: 11.3724 | LR: 0.000100
Epoch 49/50 | Train Loss: 9.9123 | Val Loss: 11.7871 | LR: 0.000100
Epoch 50/50 | Train Loss: 9.8320 | Val Loss: 11.3776 | LR: 0.000010

Best model saved from epoch 38 with Val Loss: 11.2009
```

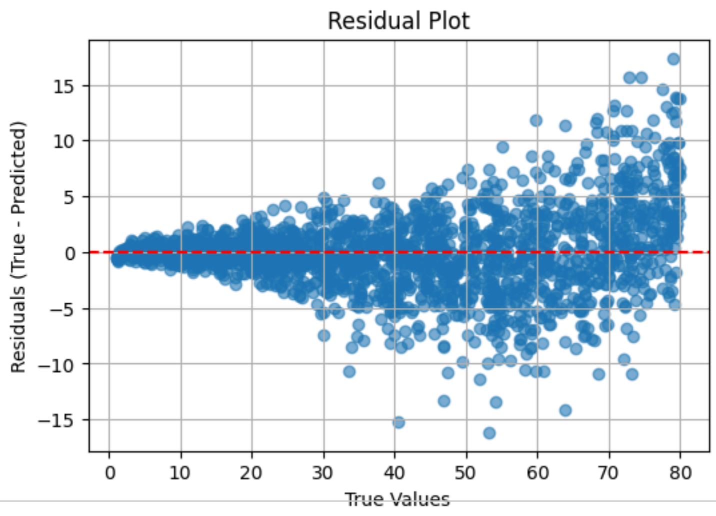
Get predictions and plot your results

Get predictions on your test set and plot your regression results for your training and test set.

```
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, r2_score

# TODO: we are to check the model's performance on the validation set
# load the "best_model.pt" and pass your test data to it and get predictions
# make sure the model is in eval mode, send the data to device
# note: make sure you handled "shuffle" properly in your validation/testing set
# loading the best model

model.load_state_dict(torch.load("best_model.pt", weights_only=True))
model.to(device) # COMPLETE ME
model.eval() # COMPLETE ME
y_true = []
y_pred = []

with torch.no_grad():
    for X_batch, y_batch in test_loader:
        X_batch = X_batch.to(device) # COMPLETE ME
        y_batch = y_batch.to(device) # COMPLETE ME
        outputs = model(X_batch) # COMPLETE ME
        y_true.extend(y_batch.cpu().numpy().flatten())
        y_pred.extend(outputs.cpu().numpy().flatten())

y_true = np.array(y_true)
y_pred = np.array(y_pred)

# Compute regression metrics
mae = mean_absolute_error(y_true, y_pred)
r2 = r2_score(y_true, y_pred)
mse = np.mean((y_true - y_pred) ** 2)
rmse = np.sqrt(mse)

print(f"Test Metrics:")
print(f"MAE  = {mae:.4f}")
print(f"RMSE = {rmse:.4f}")
print(f"R²   = {r2:.4f}")


#TODO: plot regression plot and residual plots for your regression task,
# report R2, and MAE along with any other metrics you want
# analyze your results and report on your findings.

# regression plot
plt.figure(figsize=(6,6))
plt.scatter(y_true, y_pred, alpha=0.6)
plt.plot([y_true.min(), y_true.max()], [y_true.min(), y_true.max()], 'r--', lw=2)
plt.xlabel("True Values")
plt.ylabel("Predicted Values")
plt.title("Regression Plot: True vs Predicted")
plt.grid(True)
plt.show()

# residual plots

residuals = y_true - y_pred
plt.figure(figsize=(6,4))
plt.scatter(y_true, residuals, alpha=0.6)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel("True Values")
plt.ylabel("Residuals (True - Predicted)")
plt.title("Residual Plot")
plt.grid(True)
plt.show()
```

```
Test Metrics:
MAE  = 2.3758
RMSE = 3.4916
R²   = 0.9766
```





Chlorophyll estimation was performed using a 1D convolutional neural network (CNN), achieving a mean absolute error (MAE) of 2.37 and an R^2 of 0.976. The convolutional mechanism of the CNN helps capture the underlying relationships more effectively than traditional machine learning technique Xgboost.

I am attaching the test results from Xgboost here for your reference! Your tuned model should outpefrom this performance:

MAE=3.22, R2=0.96

xgboost

Hyperparamter Tuning

This part is all you. You have a base model that is performing somewhat accurately for you. Grab a package and perform hyperparameter tuning on your hyperparameters. Read Below!!!

```
# TODO: now that we have a base model that things are working, we perform hyperparameter tuning on validation da

# change the learning rate value
# number of layers
# filter size (the bigger the filter size the more computationally expensive)
# instead of ReLU use leaky Relu or Tanh
# learning weight decay
# using max/average pooling instead of stride 2
# icorporating padding
```

```
    # you can log the lowest loss (highest accuracy) on your validation set and report that
```