H5Py Test

June 2, 2022

[1]:
```
import numpy as np
import h5py
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
from collections.abc import Iterable

batchSize = 32 #Batch size of training set

#import helpers
```

[2]:
```
def trainNetwork(model, loss_function, optimizer, numEpochs, dataloader,␣
↪numOutputs):
    for epoch in range(numEpochs):

        # Print epoch
        print(f'Starting epoch {epoch+1}')

        # Set current loss value
        current_loss = 0.0

        # Iterate over the DataLoader for training data
        for i, data in enumerate(dataloader, 0):

            # Get and prepare input

            preProcessedInputs = data[:, 0:4] #This line doesn't really do␣
↪anything, delete later?
            targets = data[:, 4:(4+numOutputs)]

            #Process intensity by putting it on a log scale
            intens = data[:, 0:1]
            intens = np.log(intens)
            inputs = torch.cat((intens, data[:,1:4]), axis = 1)

            #Process targets by putting them on a log scale
            targets = np.log(targets)
```

```
            #print(type(inputs))

            #Comment the next two lines out if not using GPU
            inputs = inputs.to('cuda')
            targets = targets.to('cuda')

            #Normalize inputs
            inputs, targets = inputs.float(), targets.float()
            targets = targets.reshape((targets.shape[0], numOutputs))

            # Zero the gradients
            optimizer.zero_grad()

            # Perform forward pass
            inputs = inputs
            outputs = model(inputs)

            #The following two lines are for debugging only
    #         if i % 10 == 0:
    #             print("Targets:", targets[0:2])
    #             print("Outputs:", outputs[0:2])
    #             print()
    #             print()

            # Compute loss
            loss = loss_function(outputs, targets)

            # Perform backward pass
            loss.backward()

            # Perform optimization
            optimizer.step()

            # Print statistics
            current_loss += loss.item()
            if i % 10 == 0:
                print('Loss after mini-batch %5d: %.3f' %
                     (i + 1, current_loss / 500))
                current_loss = 0.0

    # Process is complete.
    print('Training process has finished.')
```

[3]:

```
#See if I can import all of the data from a file

filename = 'Data_Fuchs_v_2.2_lambda_um_0.8_points_100000_seed_0.h5'
```

```
h5File = h5py.File(filename, 'r+')
```

[ ]:

```
#Read columns

intens = h5File['Intensity_(W_cm2)']

duration = h5File['Pulse_Duration_(fs)']

thickness = h5File['Target_Thickness (um)']

spotSize = h5File['Spot_Size_(FWHM um)']

maxEnergy = h5File['Max_Proton_Energy_(MeV)']

totalEnergy = h5File['Total_Proton_Energy_(MeV)']

avgEnergy = h5File['Avg_Proton_Energy_(MeV)']

test = zip(intens, duration, thickness, spotSize, maxEnergy, totalEnergy,␣
↪avgEnergy)

print(next(test)) #Prints the first row from the h5 file

nextRow = next(test)

print(type(nextRow))

#Convert columns into numpy arrays
npIntens = np.fromiter(intens, float)
npDuration = np.fromiter(duration, float)
npThickness = np.fromiter(thickness, float)
npSpot = np.fromiter(spotSize, float)
npMaxEnergy = np.fromiter(maxEnergy, float)
npTotalEnergy = np.fromiter(totalEnergy, float)
npAvgEnergy = np.fromiter(avgEnergy, float)

#print(npIntens)

print(npIntens.shape)

#Join all of those arrays into one big numpy array
npFile = np.dstack((npIntens, npDuration, npThickness, npSpot, npMaxEnergy,␣
↪npTotalEnergy, npAvgEnergy))

print(npFile.shape)
```

```
npFile = npFile.reshape(100000, 7)

print(npFile.shape)

print("Average Energy:", npAvgEnergy)
print("Total Energy:", npTotalEnergy)

print(npFile)

npTrain = npFile[:90000, 0:7]
npTest = npFile[90000:, 0:7]
print(npTrain)
print(npTest)
print("First element:", npTrain[0])
```

[ ]:
```
print(npFile)
```

[ ]:
```
#Check out one of the columns

intens[:]

#Can we convert intens by a log
logIntens = np.log(intens)

print(intens[:])
print(logIntens[:])
```

[7]:
```
#Print out every value in a column

# print(type(totalEnergy))

# for i in range(len(intens)):
#     print(intens[i])
```

[8]:
```
class MLP(nn.Module):
  '''
    Multilayer Perceptron for regression.
  '''
  def __init__(self):
    super().__init__()
    self.norm0 = nn.BatchNorm1d(4)
    self.linear1 = nn.Linear(in_features=4, out_features=64)
    self.norm1 = nn.BatchNorm1d(64)
    self.act1 = nn.SELU()
    self.linear2 = nn.Linear(in_features=64, out_features=16)
    self.norm2 = nn.BatchNorm1d(16)
    self.act2 = nn.SELU()
```

```python
        self.output = nn.Linear(in_features=16, out_features = 1)



    def forward(self, x):
        '''
          Forward pass
        '''
        x = self.norm0(x)
        x = self.linear1(x)
        x = self.norm1(x)
        x = self.act1(x)
        x = self.linear2(x)
        x = self.norm2(x)
        x = self.act2(x)
        x = self.output(x)


        return x
```

[9]:
```python
#Load in the data
#Unfamiliar with h5py, Hopefully this doesn't break anything

#h5File.close()




#f = h5py.File('test.hdf5', 'w')
#f = h5py.File('test', 'r+')

training_dataset = h5File.create_dataset(name=None, data=npTrain)


#print(list(h5File.keys()))
#test = h5File[0:7]
#test = h5File['Intensity_(W_cm2)','Pulse_Duration_(fs)']
```

1 Prepare our dataset

[278]:
```python
#Will the PyTorch DataLoader class work with H5Py datasets?

dataloader = DataLoader(training_dataset, batch_size=batchSize, shuffle=True)
```

[ ]:
```python
iterDataLoader = iter(dataloader)

print(next(iterDataLoader))
```

```
print(next(iterDataLoader))

row = next(iterDataLoader)
```

[ ]:
```
valueIntens, valueDuration, valueThickness, valueSpot, valueMax = row[0,0],␣
↪row[0,1], row[0,2], row[0,3], row[0,4]

print(valueIntens, valueDuration, valueThickness, valueSpot, valueMax)

inputs = row[0, 0:4]

print(inputs)
```

[ ]:
```
#Test iterating through the data loader

for i, data in enumerate(dataloader, 0):
    #Print inputs
    #print(data[0])
    inputs = data[:, 0:4]
    target = data[:, 4:7]

    print("Iteration:", i)
    print("Inputs:", inputs)
    print("Target:", target)
```

[ ]:
```
#Can we apply a log to just intens section of the inputs?

copy = inputs
intensCopy = row[:, 0:1]

intensCopy = np.log(intensCopy)

#print(intensCopy)

#processedInput =

#Can we concatanate this with the other tensors?

print("Size:", intensCopy.size())
print("Size:", row[:,1:4].size())

concatVer = torch.cat((intensCopy, row[:, 1:4]), axis = 1)

print(concatVer)
print(row[:,0:4])
```

[ ]:
```
#Initialize neural network

#model = MLP()
model = MLP().to('cuda') #GPU Version, comment out if not using GPU

loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)
numEpochs = 20

print(model)

trainNetwork(model, loss_function, optimizer, numEpochs, dataloader, numOutputs␣
↪= 1)
```

[16]:
```
model.eval()
```

[16]:
```
MLP(
  (norm0): BatchNorm1d(4, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (linear1): Linear(in_features=4, out_features=64, bias=True)
  (norm1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (act1): SELU()
  (linear2): Linear(in_features=64, out_features=16, bias=True)
  (norm2): BatchNorm1d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (act2): SELU()
  (output): Linear(in_features=16, out_features=1, bias=True)
)
```

[17]:
```
# Specify a path
PATH = "test_regression.pt"

# Save
torch.save(model, PATH)
```

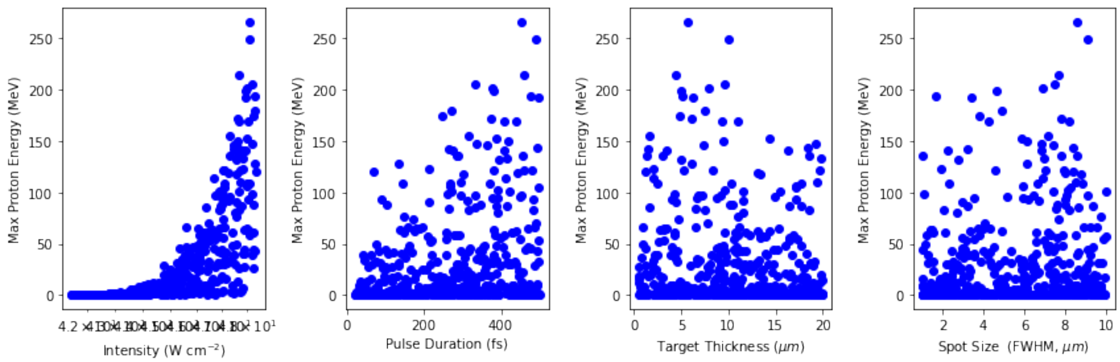
[ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt

#Let's put in the training dataset in now

test_dataset = h5File.create_dataset(name=None, data=npTest)
testDataloader = DataLoader(test_dataset, batch_size=500, shuffle=True)
iterDataLoader = iter(testDataloader)
testData = next(iterDataLoader)

#Originally had tested with the full dataset
```

```
# fulldata = DataLoader(training_dataset, batch_size=5000, shuffle=True)
# iterDataLoader = iter(fulldata)
# allData = next(iterDataLoader)

print(testData)

#Get each separate input for future plotting reasons
intens = testData[:, 0:1]
duration = testData[:, 1]
thickness = testData[:, 2]
spotSize = testData[:, 3]

#Transform intens by a log function
logIntens = np.log(intens)

#print(logIntens.size(), allData[:,1:4].size())
#Process intensity by putting it on a log scale
inputs = torch.cat((logIntens, testData[:,1:4]), axis = 1)

#Turn logIntens back into a tensor that can be passed to a plot
intens = testData[:, 0]
logIntens = np.log(intens)

target = testData[:, 4]

inputs = inputs.to('cuda')
target = target.to('cuda')

inputs, target = inputs.float(), target.float()
target = target.reshape((target.shape[0], 1))

output = model(inputs)

#print(output)
print(output)
print(target)
```

[19]:
```
#Plot just actual data itself

print(target.size())
print(intens.size())
print(output.size())
print(intens[0])

#Max KE Energy plots

fig=plt.figure(figsize=(12,4))
```

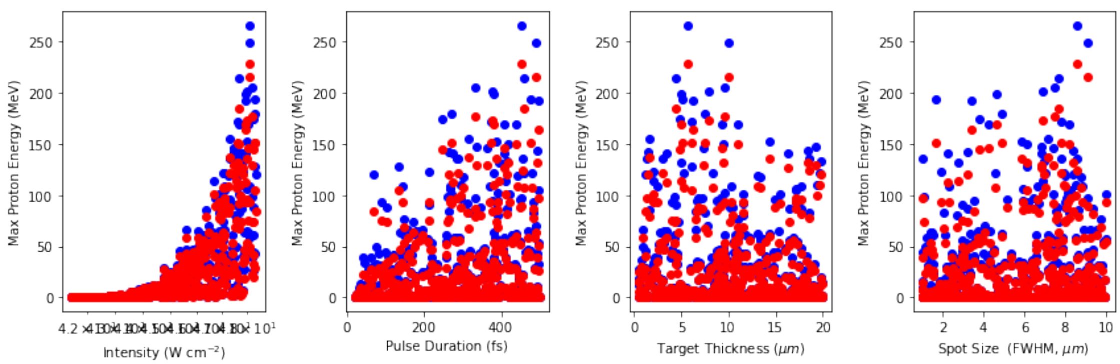
```
plt.subplot(1, 4, 1)
plt.scatter(logIntens,target[:].cpu().detach().numpy(), color = 'blue')
plt.xscale('log')
plt.xlabel(r'Intensity (W cm$^{-2}$)')
plt.ylabel('Max Proton Energy (MeV)')

plt.subplot(1, 4, 2)
plt.scatter(duration,target[:].cpu().detach().numpy(), color = 'blue')
plt.xlabel('Pulse Duration (fs)')
plt.ylabel('Max Proton Energy (MeV)')

plt.subplot(1, 4, 3)
plt.scatter(thickness,target[:].cpu().detach().numpy(), color = 'blue')
plt.xlabel(r'Target Thickness ($\mu m$)')
plt.ylabel('Max Proton Energy (MeV)')

plt.subplot(1, 4, 4)
plt.scatter(spotSize,target[:].cpu().detach().numpy(), color = 'blue')
plt.xlabel(r'Spot Size  (FWHM, $\mu m$)')
plt.ylabel('Max Proton Energy (MeV)')
plt.tight_layout()
plt.show()
```

```
torch.Size([500, 1])
torch.Size([500])
torch.Size([500, 1])
tensor(5.1352e+20, dtype=torch.float64)
```



[20]:
```
#Plot the actual data and the regressor's predicted data

#Max KE Energy plots

fig=plt.figure(figsize=(12,4))
plt.subplot(1, 4, 1)
```

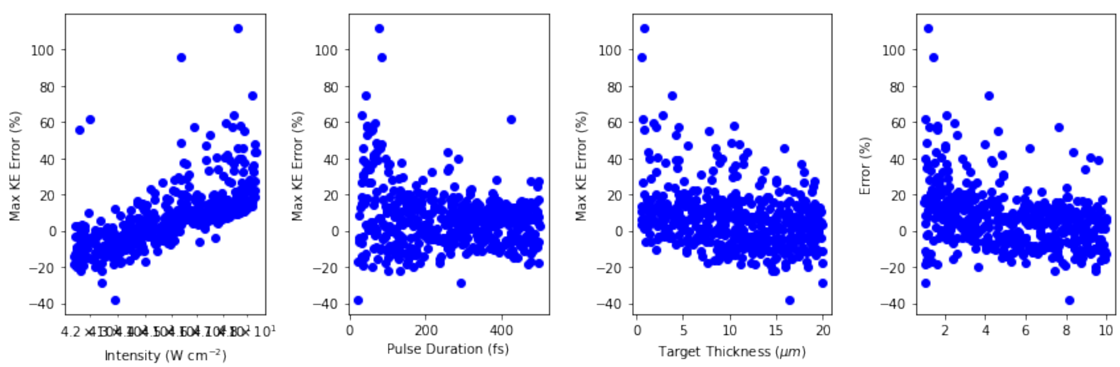
```
plt.scatter(logIntens,target[:].cpu().detach().numpy(), color = 'blue')
plt.scatter(logIntens,np.exp(output.cpu().detach().numpy()), color = 'red')
plt.xscale('log')
plt.xlabel(r'Intensity (W cm$^{-2}$)')
plt.ylabel('Max Proton Energy (MeV)')

plt.subplot(1, 4, 2)
plt.scatter(duration,target[:].cpu().detach().numpy(), color = 'blue')
plt.scatter(duration,np.exp(output.cpu().detach().numpy()), color = 'red')
plt.xlabel('Pulse Duration (fs)')
plt.ylabel('Max Proton Energy (MeV)')

plt.subplot(1, 4, 3)
plt.scatter(thickness,target[:].cpu().detach().numpy(), color = 'blue')
plt.scatter(thickness,np.exp(output.cpu().detach().numpy()), color = 'red')
plt.xlabel(r'Target Thickness ($\mu m$)')
plt.ylabel('Max Proton Energy (MeV)')

plt.subplot(1, 4, 4)
plt.scatter(spotSize,target[:].cpu().detach().numpy(), color = 'blue')
plt.scatter(spotSize,np.exp(output.cpu().detach().numpy()), color = 'red')
plt.xlabel(r'Spot Size  (FWHM, $\mu m$)')
plt.ylabel('Max Proton Energy (MeV)')
plt.tight_layout()
plt.show()
```



2 Error Plots

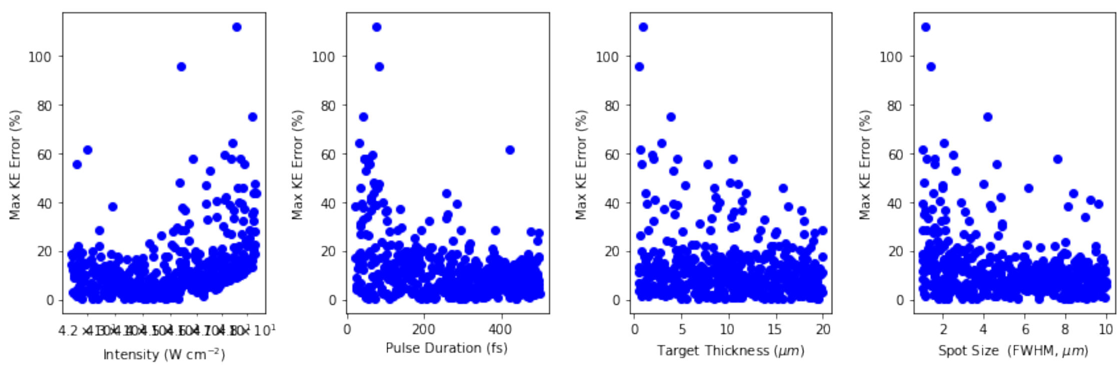
[21]:
```
#Plot relative error

difference = target[:].cpu().detach().numpy() - np.exp(output.cpu().detach().
↪numpy())
error = np.divide(difference, np.exp(output.cpu().detach().numpy())) * 100
```

```
fig=plt.figure(figsize=(12,4))
plt.subplot(1, 4, 1)
plt.scatter(logIntens, error, color = 'blue')
plt.xscale('log')
plt.xlabel(r'Intensity (W cm$^{-2}$)')
plt.ylabel('Max KE Error (%)')

plt.subplot(1, 4, 2)
plt.scatter(duration,error, color = 'blue')
plt.xlabel('Pulse Duration (fs)')
plt.ylabel('Max KE Error (%)')

plt.subplot(1, 4, 3)
plt.scatter(thickness,error, color = 'blue')
plt.xlabel(r'Target Thickness ($\mu m$)')
plt.ylabel('Max KE Error (%)')

plt.subplot(1, 4, 4)
plt.scatter(spotSize,error, color = 'blue')
plt.ylabel('Max KE Error (%)')
plt.ylabel('Error (%)')
plt.tight_layout()
plt.show()
```



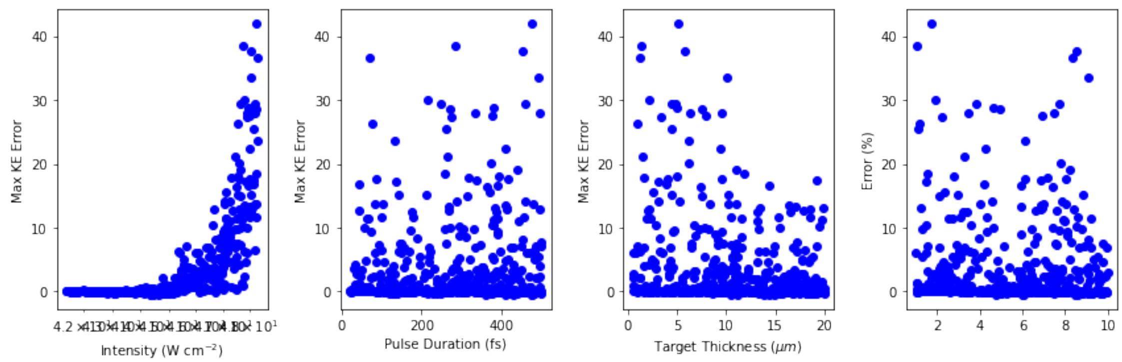
[22]:
```
#Plot relative error magnitude

difference = np.abs(target[:].cpu().detach().numpy() - np.exp(output.cpu().
↪detach().numpy()))
error = np.divide(difference, np.exp(output.cpu().detach().numpy())) * 100

fig=plt.figure(figsize=(12,4))
plt.subplot(1, 4, 1)
```

```
plt.scatter(logIntens, error, color = 'blue')
plt.xscale('log')
plt.xlabel(r'Intensity (W cm$^{-2}$)')
plt.ylabel('Max KE Error (%)')

plt.subplot(1, 4, 2)
plt.scatter(duration,error, color = 'blue')
plt.xlabel('Pulse Duration (fs)')
plt.ylabel('Max KE Error (%)')

plt.subplot(1, 4, 3)
plt.scatter(thickness,error, color = 'blue')
plt.xlabel(r'Target Thickness ($\mu m$)')
plt.ylabel('Max KE Error (%)')

plt.subplot(1, 4, 4)
plt.scatter(spotSize,error, color = 'blue')
plt.xlabel(r'Spot Size  (FWHM, $\mu m$)')
plt.ylabel('Max KE Error (%)')
plt.tight_layout()
plt.show()
```



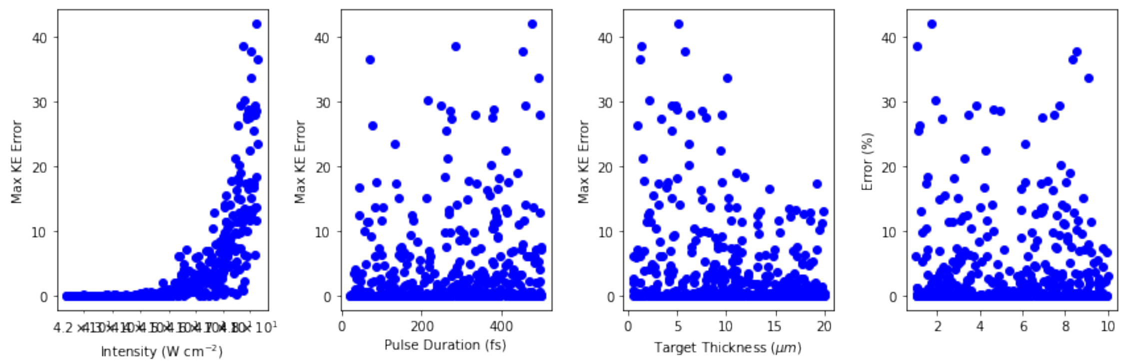
[23]:
```
#Also plot absolute errors

difference = target[:].cpu().detach().numpy() - np.exp(output.cpu().detach().
↪numpy())


fig=plt.figure(figsize=(12,4))
plt.subplot(1, 4, 1)
plt.scatter(logIntens, difference, color = 'blue')
plt.xscale('log')
plt.xlabel(r'Intensity (W cm$^{-2}$)')
```

```
plt.ylabel('Max KE Error')

plt.subplot(1, 4, 2)
plt.scatter(duration, difference, color = 'blue')
plt.xlabel('Pulse Duration (fs)')
plt.ylabel('Max KE Error')

plt.subplot(1, 4, 3)
plt.scatter(thickness, difference, color = 'blue')
plt.xlabel(r'Target Thickness ($\mu m$)')
plt.ylabel('Max KE Error')

plt.subplot(1, 4, 4)
plt.scatter(spotSize, difference, color = 'blue')
plt.ylabel('Max KE Error')
plt.ylabel('Error (%)')
plt.tight_layout()
plt.show()
```



[24]:
```
#Absolute error magnitude

difference = np.abs(target[:].cpu().detach().numpy() - np.exp(output.cpu().
↪detach().numpy()))


fig=plt.figure(figsize=(12,4))
plt.subplot(1, 4, 1)
plt.scatter(logIntens, difference, color = 'blue')
plt.xscale('log')
plt.xlabel(r'Intensity (W cm$^{-2}$)')
plt.ylabel('Max KE Error')

plt.subplot(1, 4, 2)
```

```
plt.scatter(duration, difference, color = 'blue')
plt.xlabel('Pulse Duration (fs)')
plt.ylabel('Max KE Error')

plt.subplot(1, 4, 3)
plt.scatter(thickness, difference, color = 'blue')
plt.xlabel(r'Target Thickness ($\mu m$)')
plt.ylabel('Max KE Error')

plt.subplot(1, 4, 4)
plt.scatter(spotSize, difference, color = 'blue')
plt.ylabel('Max KE Error')
plt.ylabel('Error (%)')
plt.tight_layout()
plt.show()
```



3 Could be better, but we're just using this regression neural network as practice for when we make an invertible regression neural network. There seems to be a small habit of underestimating the actual values.

3.1 The regresion neural network looks like it does a pretty good job of predicting the max proton energy values, now can we get it to predict all three values at once?

[332]:
```
#Define a new neural network that outputs three features rather than one

class MultiRegressor(nn.Module):
  '''
    Multilayer Perceptron for regression.
  '''
  def __init__(self):
```

```
        super().__init__()
        self.norm0 = nn.BatchNorm1d(4)
        self.linear1 = nn.Linear(in_features=4, out_features=64)
        self.norm1 = nn.BatchNorm1d(64)
        self.act1 = nn.LeakyReLU()
        self.dropout = nn.Dropout()
        self.linear2 = nn.Linear(in_features=64, out_features=16)
        self.norm2 = nn.BatchNorm1d(16)
        #self.dropout = nn.Dropout()
        self.act2 = nn.LeakyReLU()
        self.linear3 = nn.Linear(in_features=16, out_features=8)
        self.act3 = nn.LeakyReLU()
        #self.dropout = nn.Dropout()
        self.output = nn.Linear(in_features=8, out_features = 3)



    def forward(self, x):
        '''
          Forward pass
        '''
        x = self.norm0(x)
        x = self.linear1(x)
        x = self.norm1(x)
        x = self.act1(x)
        #x = self.dropout(x)
        x = self.linear2(x)
        x = self.norm2(x)
        #x = self.dropout(x)
        x = self.act2(x)
        x = self.linear3(x)
        x = self.act3(x)
        #x = self.dropout(x)
        x = self.output(x)


        return x
```

3.2 The following two cells were used by me to figure out some stuff about the shape of our outputs

[ ]:
```
targets = data[:, 4:7]

#print(targets)

targets = data[:, 4]
print(targets.shape[0])
```

```
targets = data[:, 4:7]
print(targets.shape[0])
```

[ ]:
```
targets = data[:, 4:7]
targets = targets.reshape((targets.shape[0], 3))
print(targets)
```

4 Train our multi-output neural network

[ ]:
```
#Initialize neural network

#model = MultiRegressor()
multiModel = MultiRegressor().to('cuda') #GPU Version, comment out if not using␣
↪GPU

loss_function = nn.MSELoss()
#loss_function = nn.L1Loss()
optimizer = torch.optim.Adam(multiModel.parameters(), lr=1e-3)
numEpochs = 20

print(multiModel)


trainNetwork(multiModel, loss_function, optimizer, numEpochs, dataloader, 3)
```

[ ]:
```
#Validate with test dataset

testDataloader = DataLoader(test_dataset, batch_size=5000, shuffle=True)
iterDataLoader = iter(testDataloader)
testData = next(iterDataLoader)


#Old code utilizing the full dataset

# fulldata = DataLoader(training_dataset, batch_size=5000, shuffle=True)
# iterDataLoader = iter(fulldata)
# allData = next(iterDataLoader)

# print(allData)

#Get each separate input for future plotting reasons
intens = testData[:, 0:1]
duration = testData[:, 1]
thickness = testData[:, 2]
spotSize = testData[:, 3]
```

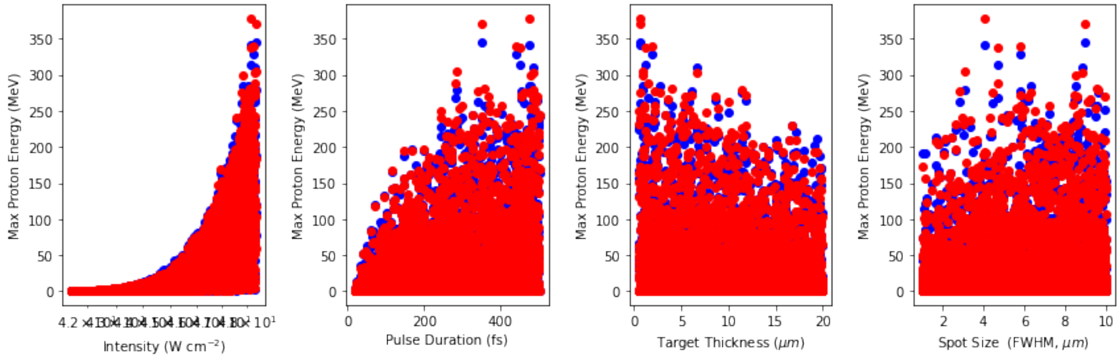
```
#Transform intens by a log function
logIntens = np.log(intens)

print(logIntens.size(), testData[:,1:4].size())
#Process intensity by putting it on a log scale
inputs = torch.cat((logIntens, testData[:,1:4]), axis = 1)

#Turn logIntens back into a tensor that can be passed to a plot
intens = testData[:, 0]
logIntens = np.log(intens)

#inputs = allData[:, 0:4]
target = testData[:, 4:7]
target = np.log(target)

inputs = inputs.to('cuda')
target = target.to('cuda')

inputs, target = inputs.float(), target.float()
target = target.reshape((target.shape[0],3))

#print("Inputs:", inputs)
#print("Targets:", target)

output = multiModel(inputs)

target = np.log(testData[:, 4:7])

print("Targets:", target)
#print(output)
print("Outputs:", output)
```

[ ]:
```
print("Target:", target[:])
print(target[:, 0])
print("Output:", output)
print(output[:, 0])
```

4.1 It seems the multi-regressor has a habit of under-predicting max and total proton energy by a slight margin. It is way off for average proton energy though, maybe try applying a log-scale?

[338]:
```
%matplotlib inline
import matplotlib.pyplot as plt

#Plot the actual data and the regressor's predicted data
```

```python
#Max KE Energy plots

index = 0

fig=plt.figure(figsize=(12,4))
plt.subplot(1, 4, 1)
plt.scatter(logIntens,np.exp(target[:, index].cpu().detach().numpy()), color = 'blue')
plt.scatter(logIntens,np.exp(output[:, index].cpu().detach().numpy()), color = 'red')
plt.xscale('log')
plt.xlabel(r'Intensity (W cm$^{-2}$)')
plt.ylabel('Max Proton Energy (MeV)')

plt.subplot(1, 4, 2)
plt.scatter(duration,np.exp(target[:, index].cpu().detach().numpy()), color = 'blue')
plt.scatter(duration,np.exp(output[:, index].cpu().detach().numpy()), color = 'red')
plt.xlabel('Pulse Duration (fs)')
plt.ylabel('Max Proton Energy (MeV)')

plt.subplot(1, 4, 3)
plt.scatter(thickness,np.exp(target[:, index].cpu().detach().numpy()), color = 'blue')
plt.scatter(thickness,np.exp(output[:, index].cpu().detach().numpy()), color = 'red')
plt.xlabel(r'Target Thickness ($\mu m$)')
plt.ylabel('Max Proton Energy (MeV)')

plt.subplot(1, 4, 4)
plt.scatter(spotSize,np.exp(target[:, index].cpu().detach().numpy()), color = 'blue')
plt.scatter(spotSize,np.exp(output[:, index].cpu().detach().numpy()), color = 'red')
plt.xlabel(r'Spot Size  (FWHM, $\mu m$)')
plt.ylabel('Max Proton Energy (MeV)')
plt.tight_layout()
plt.show()
```



5 Error plots

5.1 First max kinetic energy, then total energy, and finally average energy

[339]:
```
def plot_relative_error(target, output, logIntens, duration, thickness,␣
↪spotSize, index, label):
    difference = np.exp(target[:, index].cpu().detach().numpy()) - np.
↪exp(output[:, index].cpu().detach().numpy())
    difference = np.abs(difference)
    error = np.divide(difference, np.exp(output[:, index].cpu().detach().
↪numpy())) * 100

    fig=plt.figure(figsize=(12,8))

    plt.subplot(1, 4, 1)
    plt.scatter(logIntens, error, color = 'blue')
    plt.xscale('log')
    #plt.ylim([0, 100])
    plt.xlabel(r'Intensity (W cm$^{-2}$)')
    plt.ylabel(label)

    plt.subplot(1, 4, 2)
    plt.scatter(duration, error, color = 'blue')
    plt.xlabel('Pulse Duration (fs)')
    plt.ylabel(label)

    plt.subplot(1, 4, 3)
    plt.scatter(thickness,error, color = 'blue')
    plt.xlabel(r'Target Thickness ($\mu m$)')
    plt.ylabel(label)

    plt.subplot(1, 4, 4)
    plt.scatter(spotSize, error, color = 'blue')
```

```
    plt.xlabel(r'Spot Size  (FWHM, $\mu m$)')
    plt.ylabel(label)

    plt.tight_layout()
    plt.show()
```

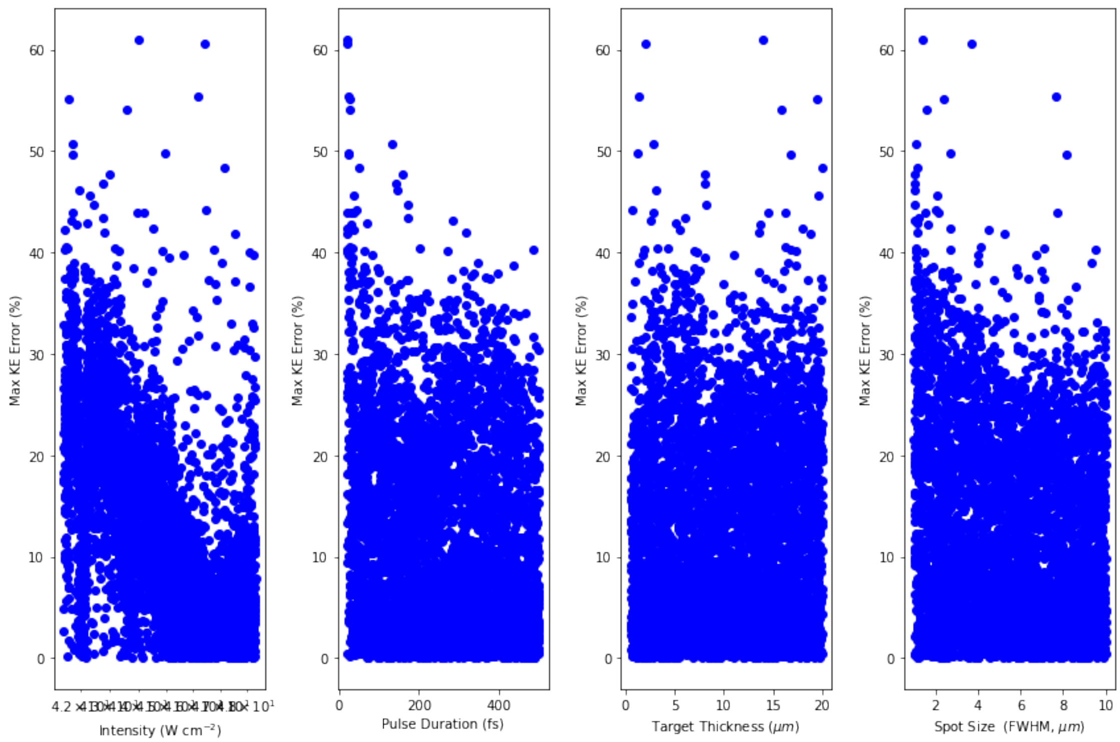
[340]:
```
# Positive absolute error means the NN had over-estimated the error
# Negative absolute error means the NN had under-estimated the error
def plot_absolute_error(target, output, logIntens, duration, thickness,␣
 ↪spotSize, index, label):
    difference = np.exp(output[:, index].cpu().detach().numpy()) - np.
 ↪exp(target[:, index].cpu().detach().numpy())
    absDifference = np.abs(difference)

    fig=plt.figure(figsize=(12,8))
    plt.subplot(2, 4, 1)
    plt.scatter(logIntens, difference, color = 'blue')
    plt.xscale('log')
    #plt.ylim([0, -1])
    plt.xlabel(r'Intensity (W cm$^{-2}$)')
    plt.ylabel(label)

    plt.subplot(2, 4, 2)
    plt.scatter(duration, difference, color = 'blue')
    plt.xlabel('Pulse Duration (fs)')
    plt.ylabel(label)

    plt.subplot(2, 4, 3)
    plt.scatter(thickness, difference, color = 'blue')
    plt.xlabel(r'Target Thickness ($\mu m$)')
    plt.ylabel(label)

    plt.subplot(2, 4, 4)
    plt.scatter(spotSize, difference, color = 'blue')
    plt.xlabel(r'Spot Size  (FWHM, $\mu m$)')
    plt.ylabel(label)

    plt.subplot(2, 4, 5)
    plt.scatter(logIntens, absDifference, color = 'blue')
    plt.xscale('log')
    #plt.ylim([0, 1])
    plt.xlabel(r'Intensity (W cm$^{-2}$)')
    plt.ylabel(label + " (Magnitude)")

    plt.subplot(2, 4, 6)
    plt.scatter(duration, absDifference, color = 'blue')
    plt.xlabel('Pulse Duration (fs)')
```

```
    plt.ylabel(label + " (Magnitude)")

    plt.subplot(2, 4, 7)
    plt.scatter(thickness, absDifference, color = 'blue')
    plt.xlabel(r'Target Thickness ($\mu m$)')
    plt.ylabel(label + " (Magnitude)")

    plt.subplot(2, 4, 8)
    plt.scatter(spotSize, absDifference, color = 'blue')
    plt.xlabel(r'Spot Size  (FWHM, $\mu m$)')
    plt.ylabel(label + " (Magnitude)")

    plt.tight_layout()
    plt.show()
```

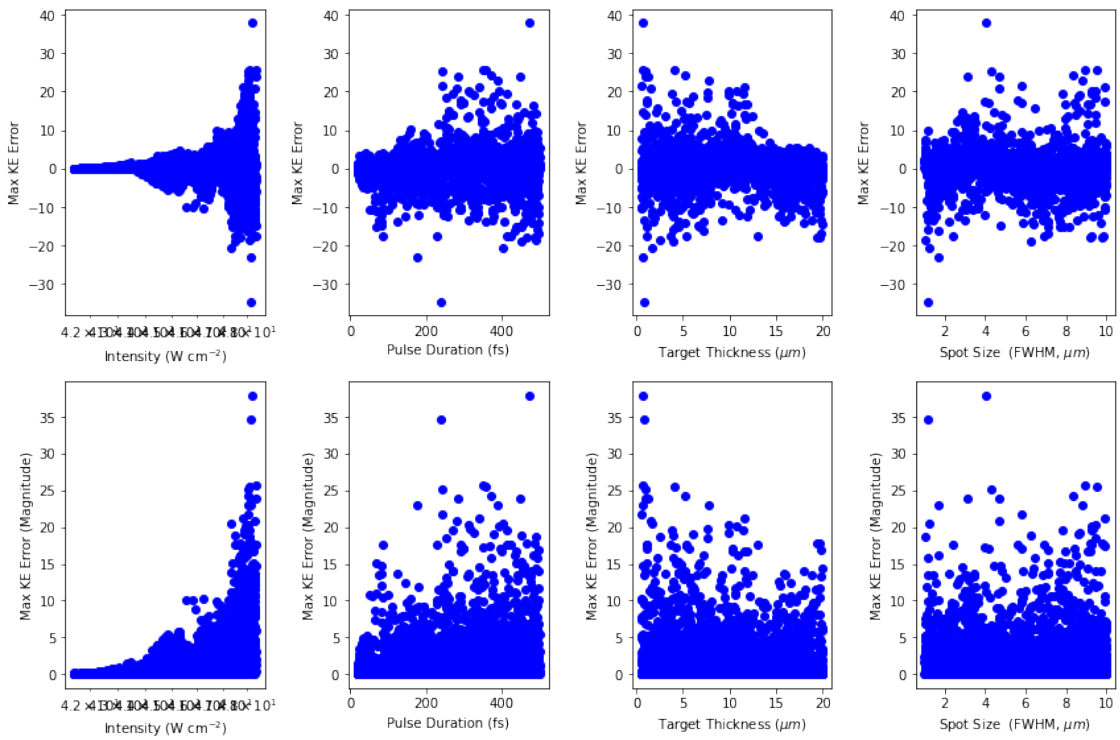
[341]:
```
def calc_MSE_Error(target, output, index):
    result = np.square(np.subtract(np.exp(target[:, index].cpu().detach().
 ↪numpy()), np.exp(output[:, index].cpu().detach().numpy()))).mean())

    return result
```

[342]:
```
def calc_Avg_Percent_Error(target, output, index):
    difference = np.exp(target[:, index].cpu().detach().numpy()) - np.
 ↪exp(output[:, index].cpu().detach().numpy())
    difference = np.abs(difference)
    error = np.divide(difference, np.exp(output[:, index].cpu().detach().
 ↪numpy())) * 100

    result = error.mean()

    return result
```

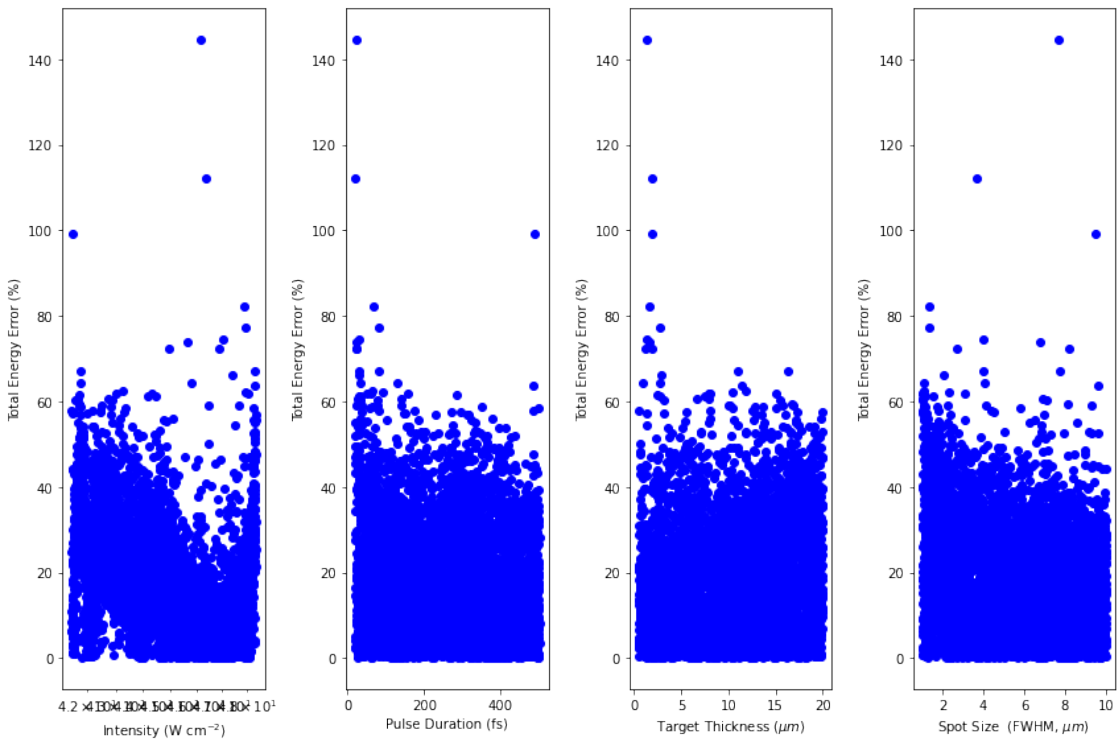
[343]:
```
#Plot relative error and relative error magnitude

plot_relative_error(target, output, logIntens, duration, thickness, spotSize,␣
 ↪0, 'Max KE Error (%)')
```



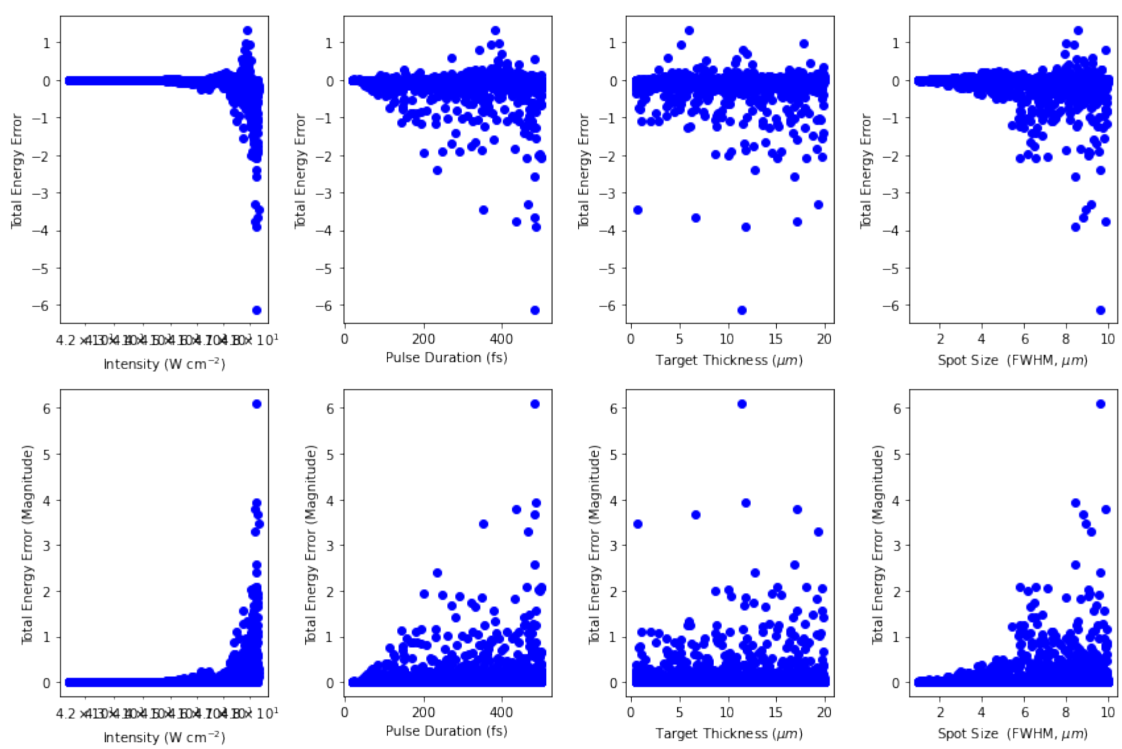
[344]:
```
#Also plot absolute errors

plot_absolute_error(target, output, logIntens, duration, thickness, spotSize,␣
 ↪0, 'Max KE Error')
```



[345]:
```
#Errors for total proton energy

plot_relative_error(target, output, logIntens, duration, thickness, spotSize,␣
↪1, 'Total Energy Error (%)')
```

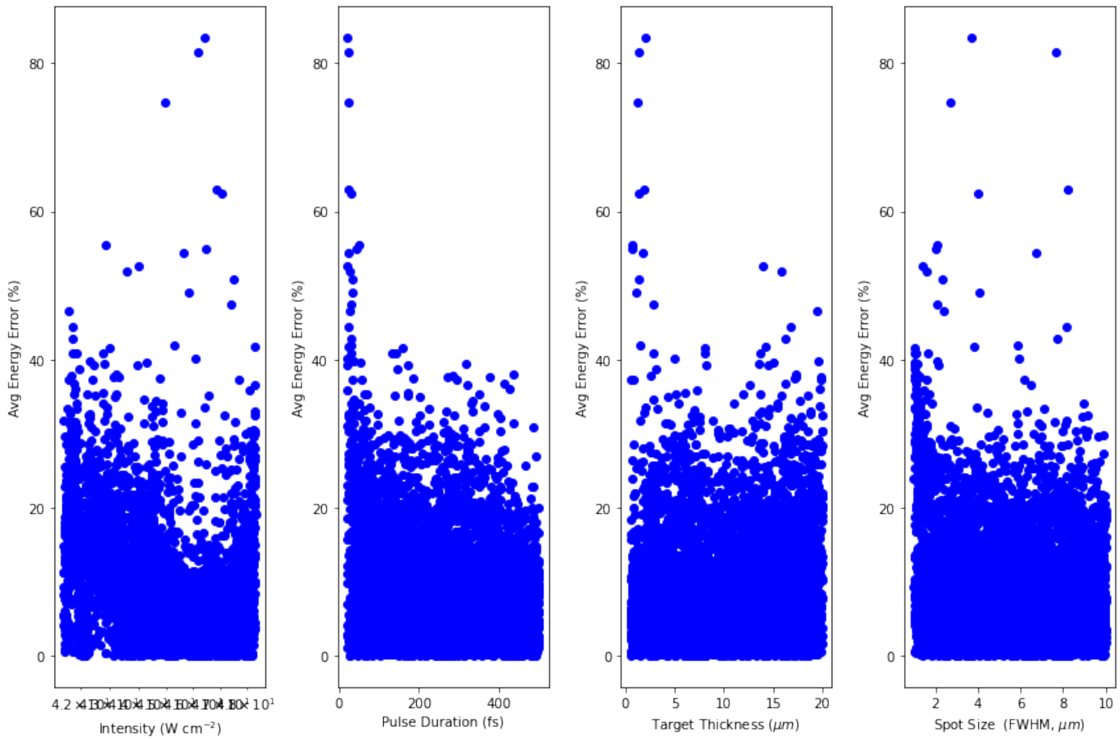


[346]:
```
plot_absolute_error(target, output, logIntens, duration, thickness, spotSize,
    1, 'Total Energy Error')
```



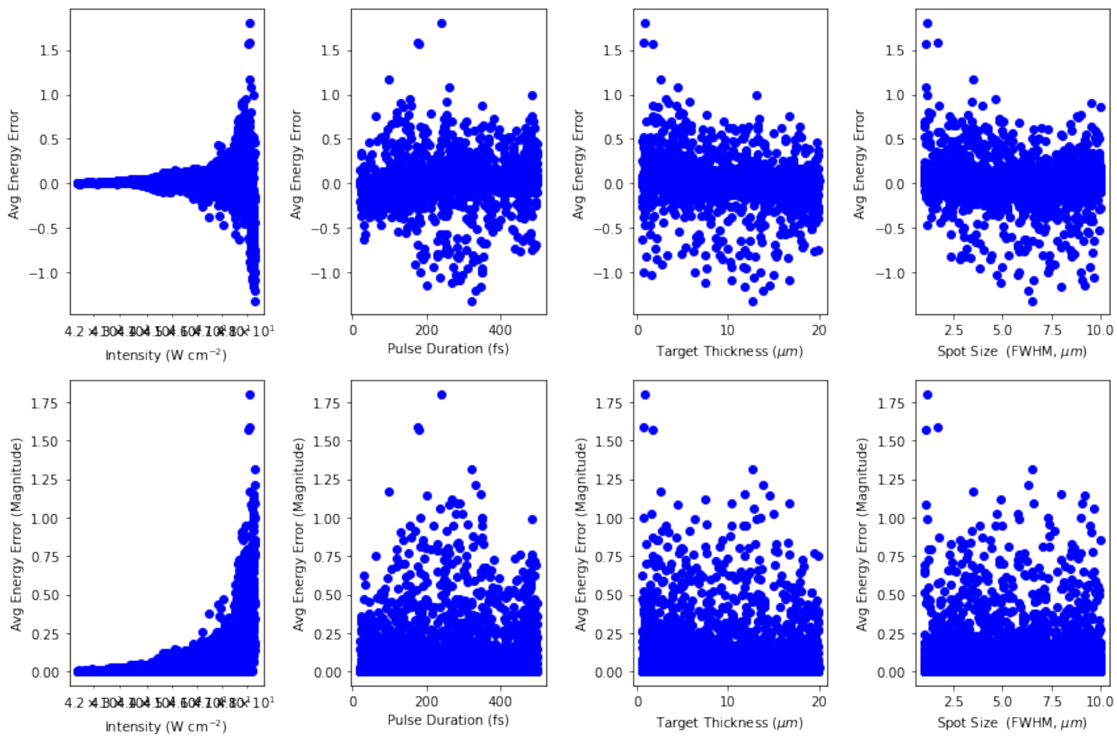
[347]:

```
#Finally, do average energy

plot_relative_error(target, output, logIntens, duration, thickness, spotSize,
    2, 'Avg Energy Error (%)')
```



[348]:

```
plot_absolute_error(target, output, logIntens, duration, thickness, spotSize,
    2, 'Avg Energy Error')
```



[349]:
```
#Calculate mean squared error values
#Also calculate the average relative error

error = [0., 1., 2.]
percentError = [0., 1., 2.]

for index in range(3):
    error[index] = calc_MSE_Error(target, output, index)
    percentError[index] = calc_Avg_Percent_Error(target, output, index)

print("Max KE MSE:", error[0])
print("Max KE Average Relative Error:", percentError[0])
print("Total Energy MSE:", error[1])
print("Total Energy Average Relative Error:", percentError[1])
print("Avg Energy MSE:", error[2])
print("Avg Energy Average Relative Error:", percentError[2])
```

```
Max KE MSE: 0.06868434309897596
Max KE Average Relative Error: 12.354636816137976
Total Energy MSE: 0.0011237955635835246
Total Energy Average Relative Error: 18.517587987041914
Avg Energy MSE: 0.00022563304342974302
Avg Energy Average Relative Error: 9.929525760678816
```

[350]:
```
%matplotlib notebook
import matplotlib.pyplot as plt

#We'll be using matplotlib notebooks so we can easily rotate the 3-D plots␣
↪interactively
#Note that when we have many, many points, these plots can be very slow to␣
↪respond to rotations

size3D = (10,6) #Controls the figure size for our plots

fig = plt.figure(figsize=size3D)

#Intensity and Pulse Duration
ax = fig.add_subplot(1, 1, 1, projection='3d')
ax.mouse_init()
ax.set_xlabel('Max Energy', fontweight ='bold')
ax.set_ylabel('Total Energy', fontweight ='bold')
ax.set_zlabel('Average Energy', fontweight ='bold')
ax.scatter3D(target[:, 0].cpu().detach().numpy(), target[:, 1].cpu().detach().
↪numpy(), target[:, 2].cpu().detach().numpy(),
            color = 'blue')
ax.scatter3D(output[:, 0].cpu().detach().numpy(), output[:, 1].cpu().detach().
↪numpy(), output[:, 2].cpu().detach().numpy(),
            color = 'red')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[350]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1cc082fe610>
```

6 Conclusion:

- Apply the log function to intensity for neural networks and all of the outputs for the energies
- Maybe look into whether or not we only need to apply a log to the average energy?

[ ]:
```
```