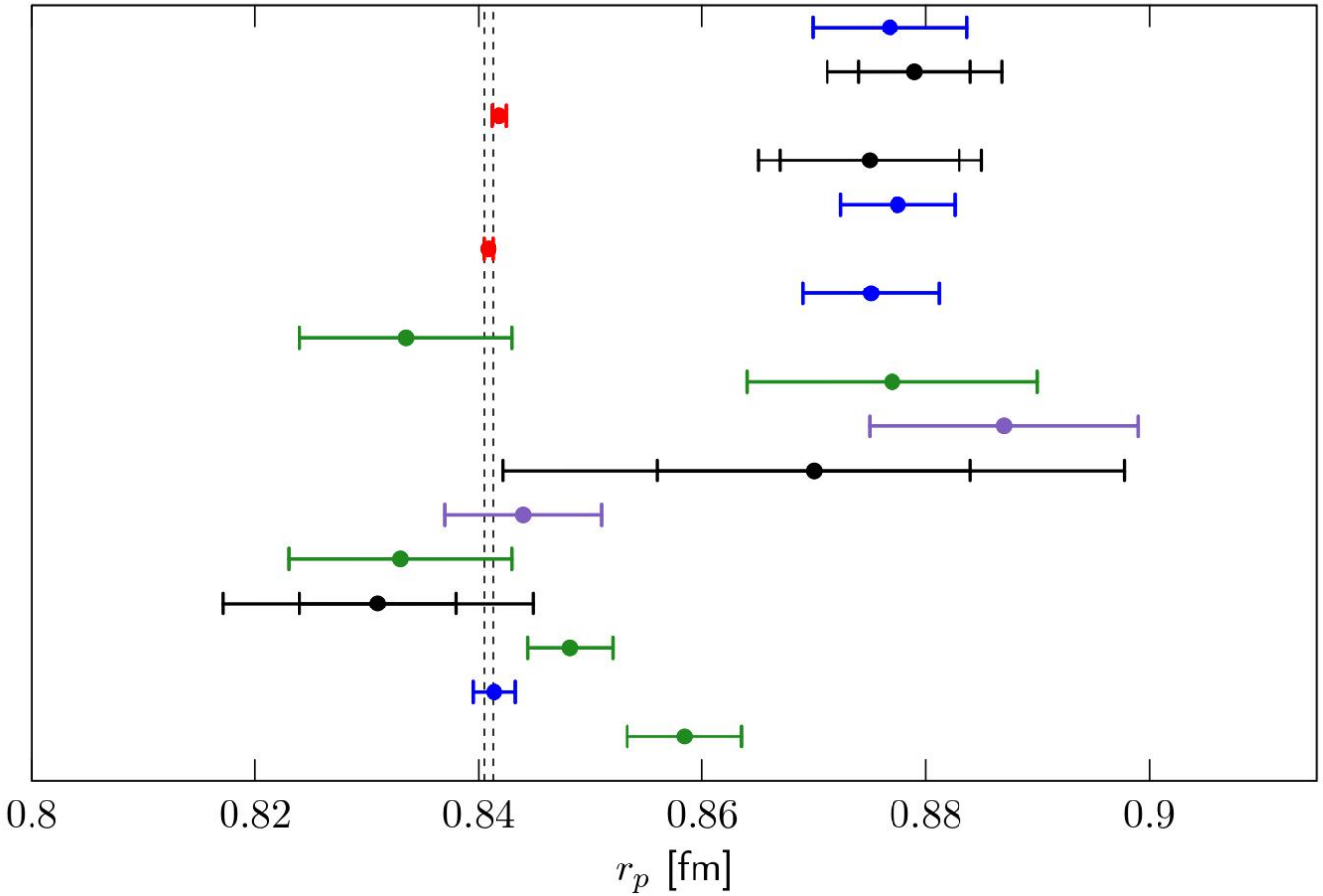
Enhancing the Resolution of the MUSE Beamline Calorimeter Using Machine Learning

Charlie Brown

January 10, 2025

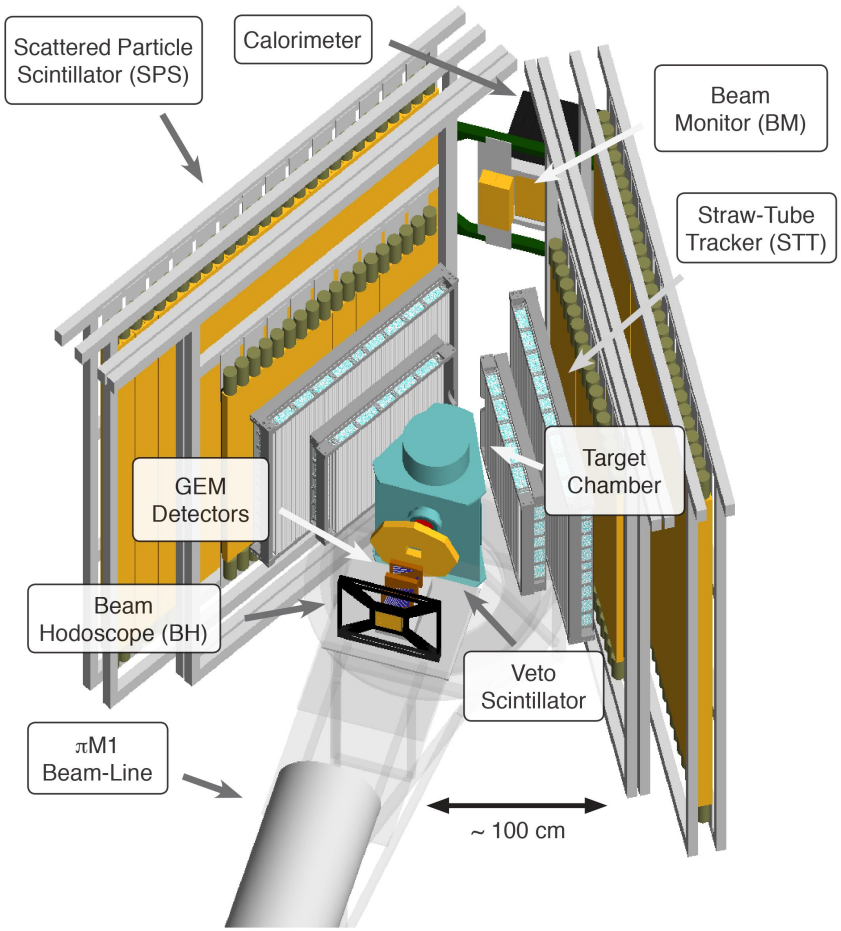
Stony Brook University

Divergent Timeline of Proton Radius Measurements

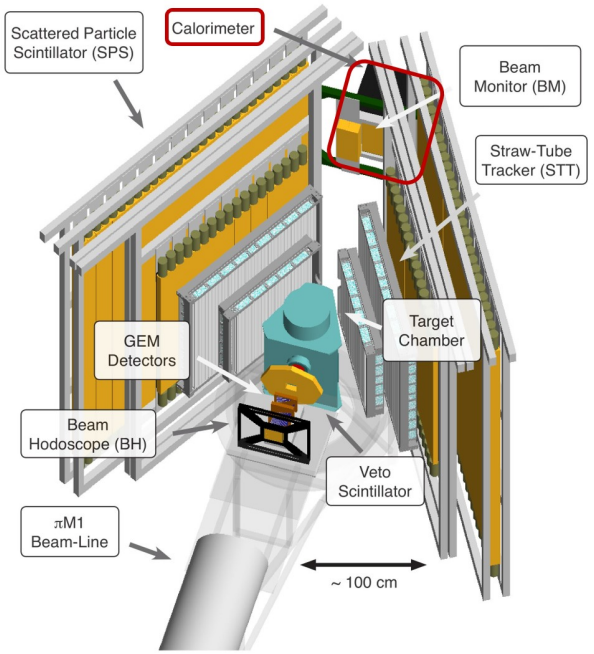


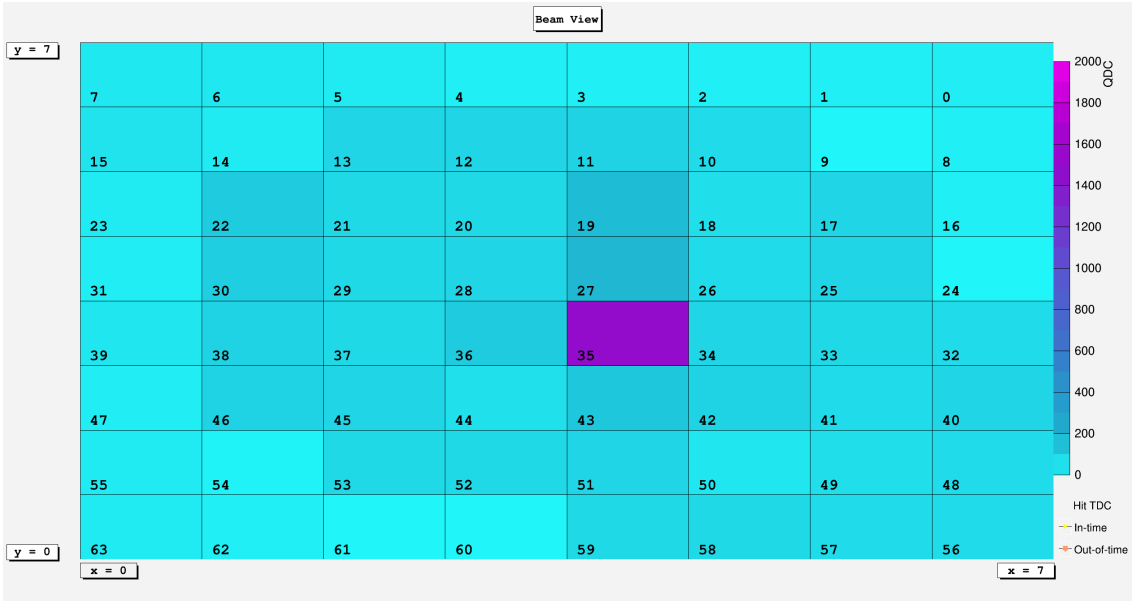
CODATA'06 (2008)
Bernauer (2010)
Pohl (2010)
Zhan (2011)
CODATA'10 (2012)
Antognini (2013)
CODATA'14 (2015)
Beyer (2017)
Fleurbaey (2018)
Sick (2018)
Mihovilovič (2019)
Alarćon (2019)
Bezginov (2019)
Xiong (2019)
Grinin (2020)
CODATA'18 (2021)
Brandt (2022)

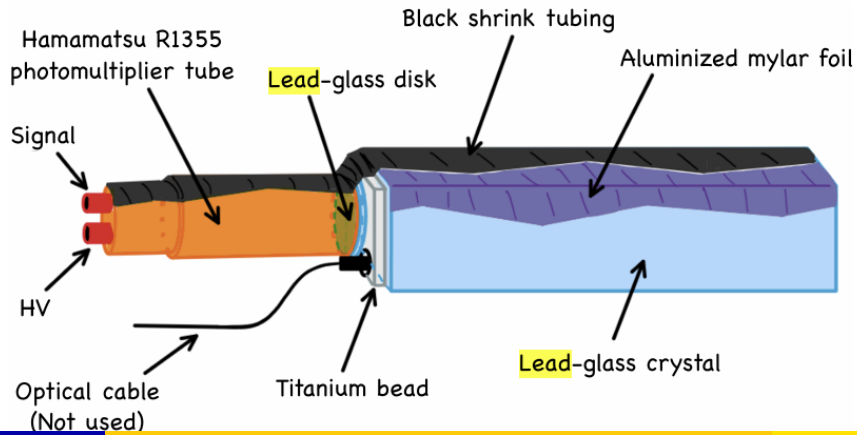
Experimental Setup



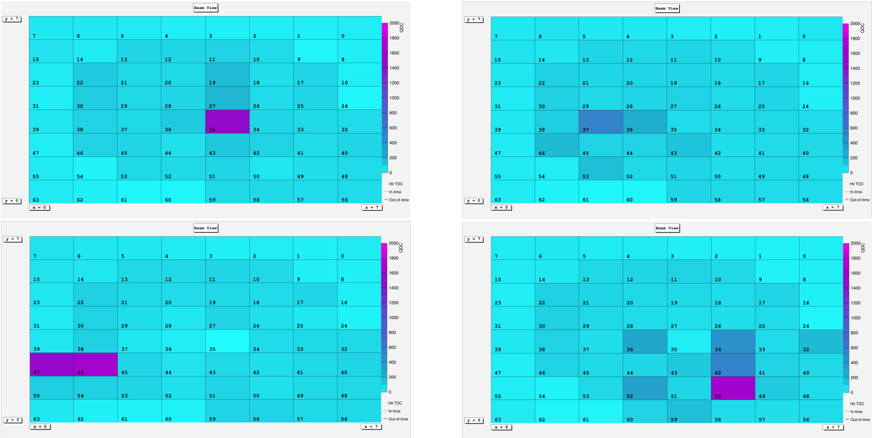
Experimental Setup



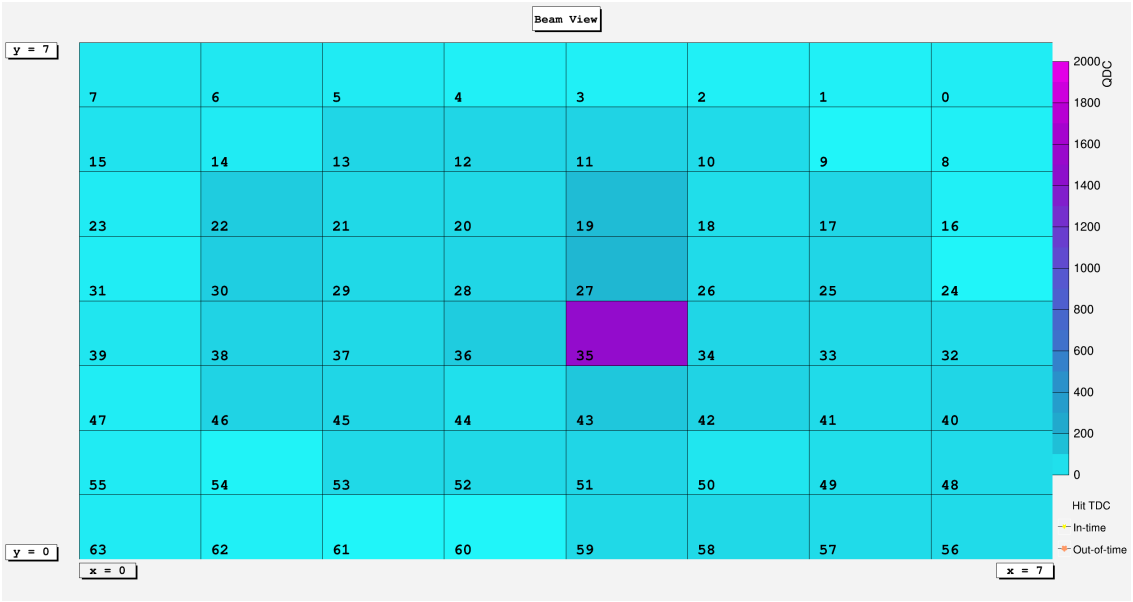


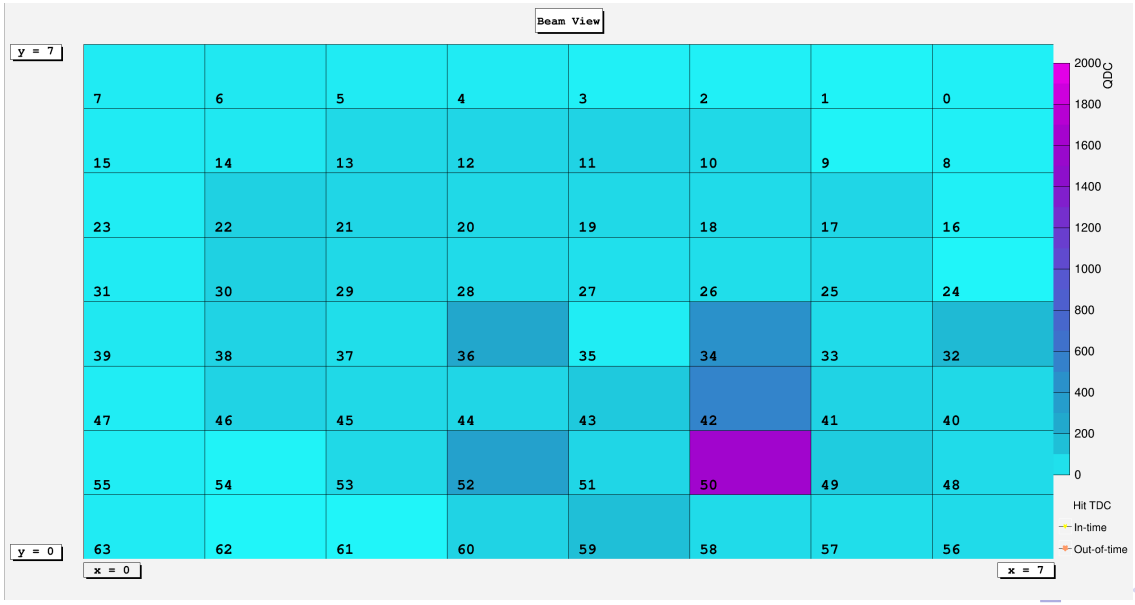


Calorimeter Examples

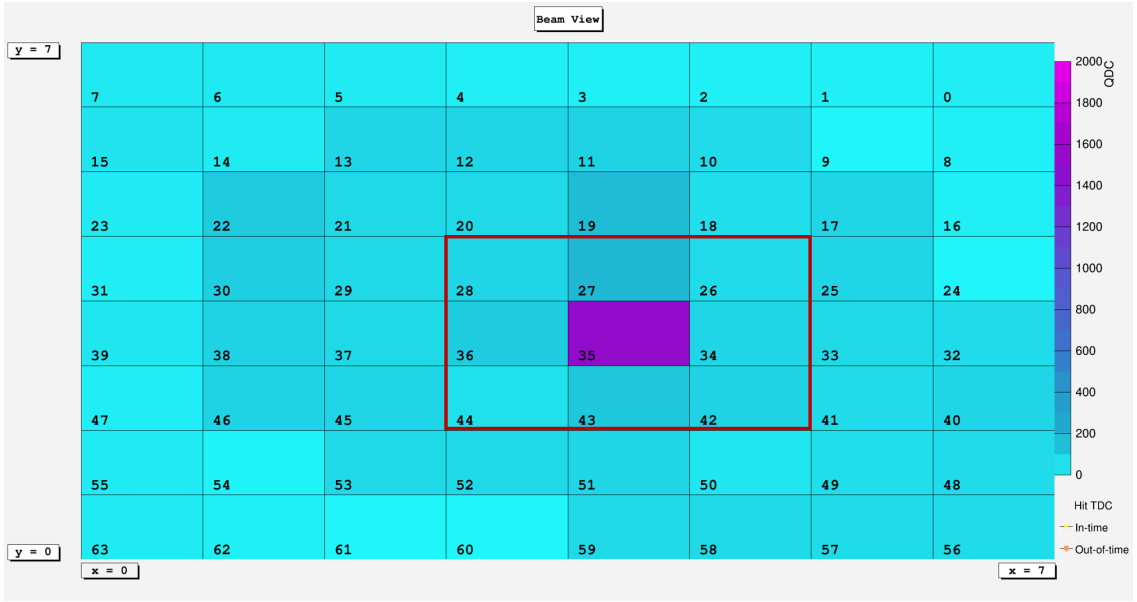


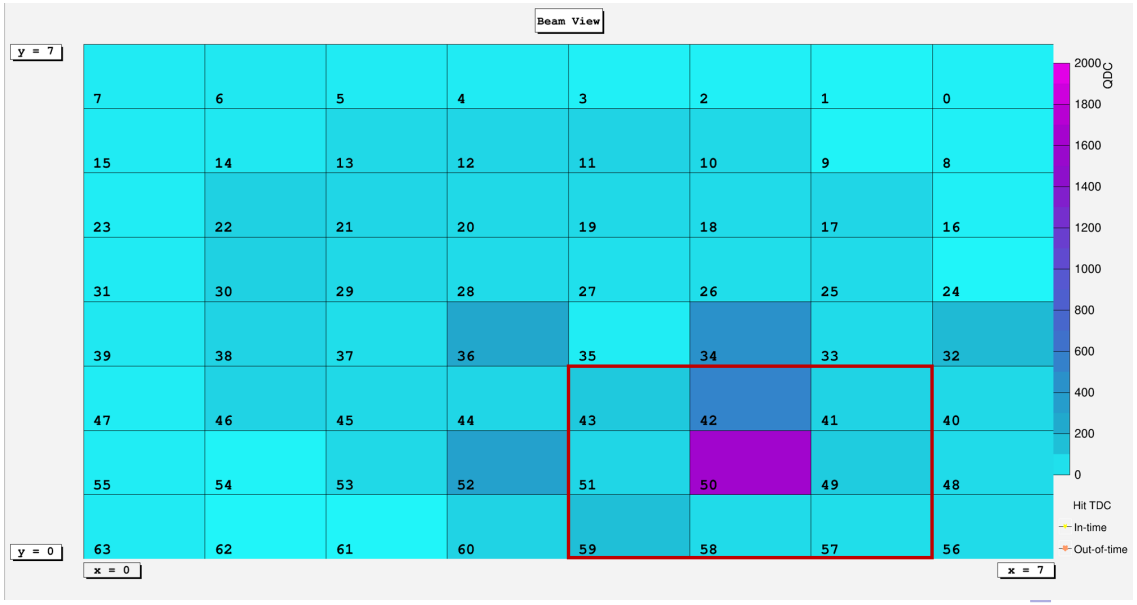
Calorimeter Examples



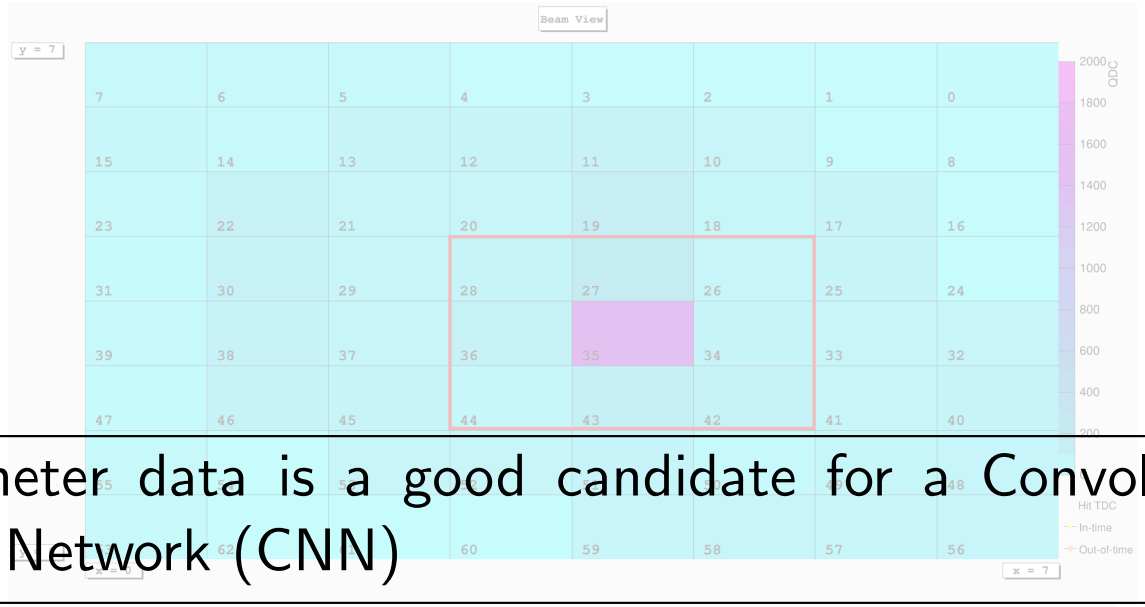


Calorimeter Examples

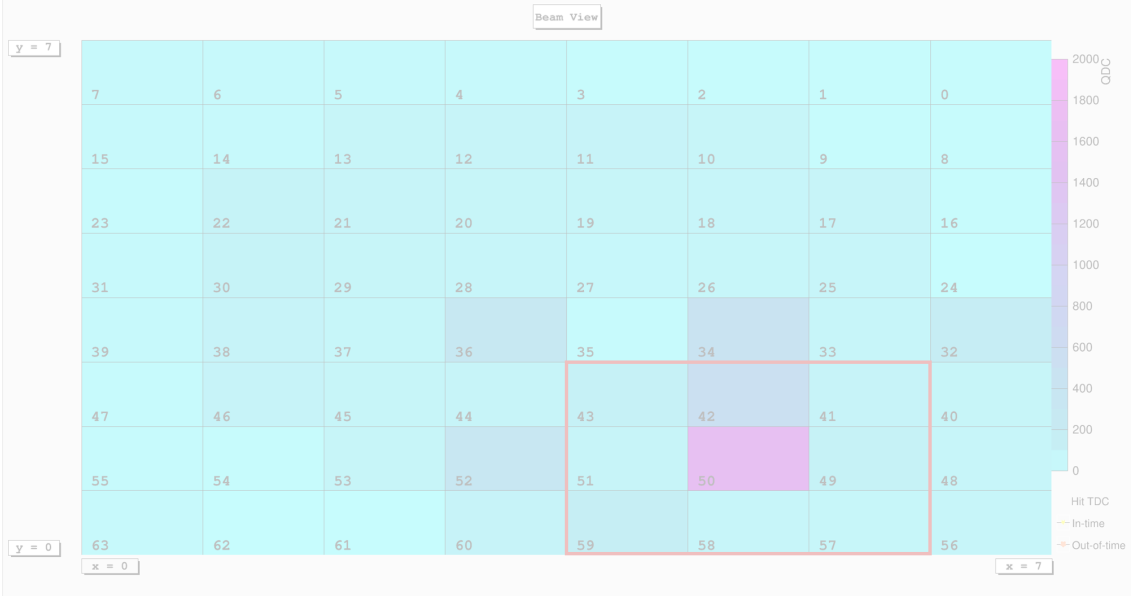




Calorimeter Examples



Calorimeter data is a good candidate for a Convolutional Neural Network (CNN)



Convolutional Neural Networks (Intro)

What is a CNN?

- Go-to technique for analyzing images
- Extracts spacial features
- Hierarchical feature learning

ML Review

- Two steps: forward pass and backward pass
- Forward pass: Uses filters/weights to generate guess
- Backward pass: Finds error between guess and target value, adjusts filters/weights accordingly

Convolutional Neural Network (Parts)

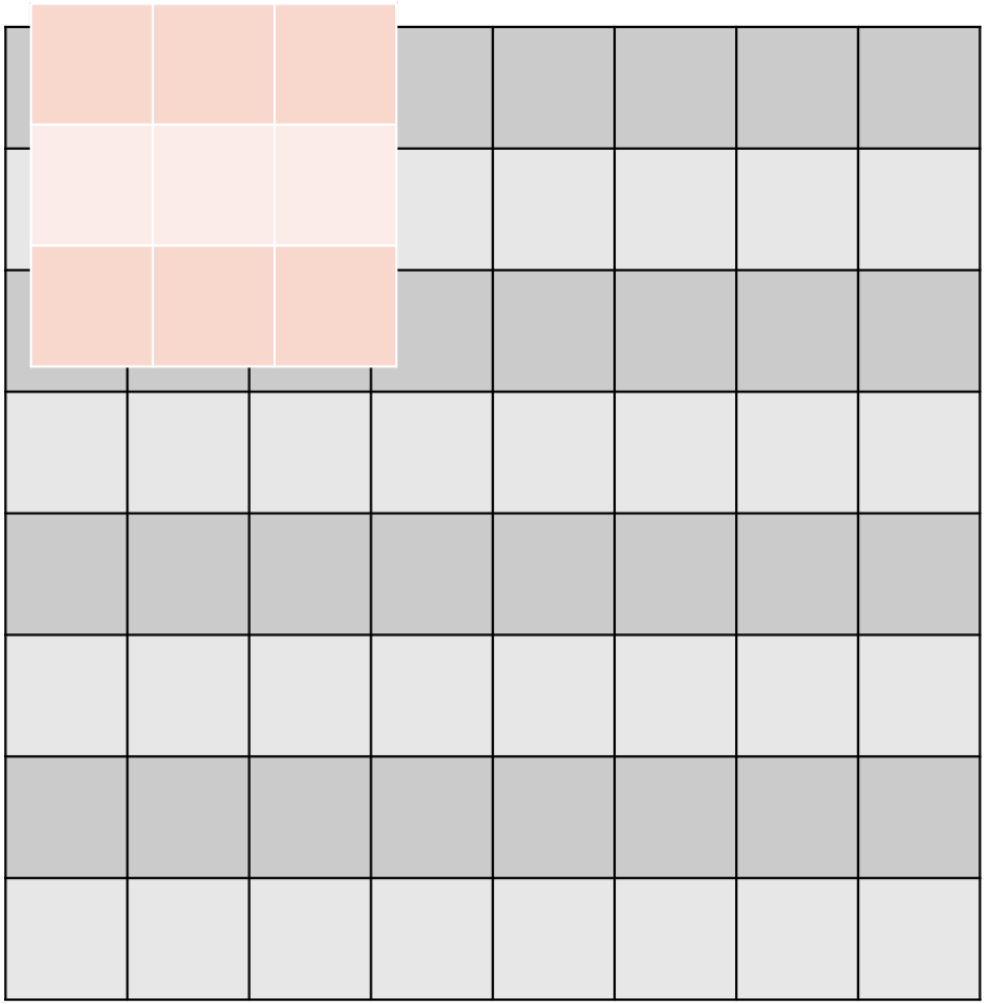
Convolution

ReLU

Pooling

Flattening

Linear



Convolutional Neural Network (Parts)

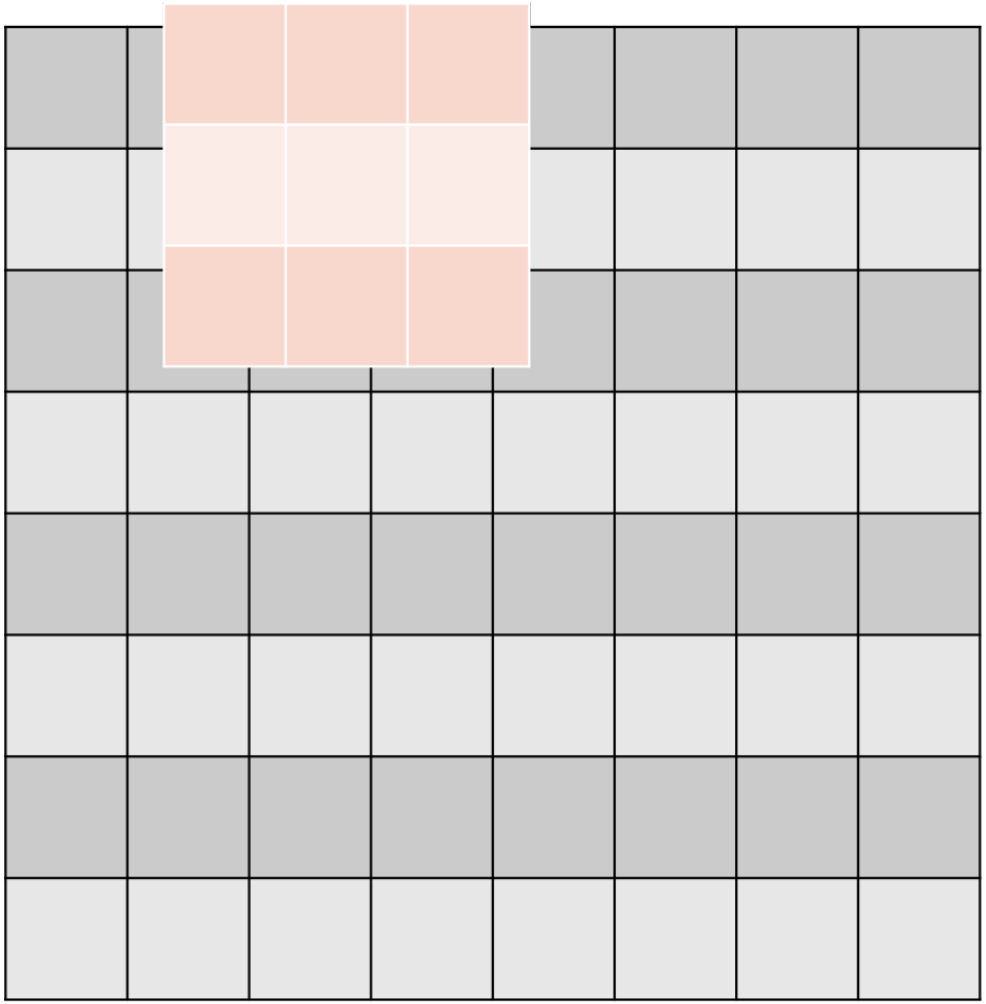
Convolution

ReLU

Pooling

Flattening

Linear



Convolutional Neural Network (Parts)

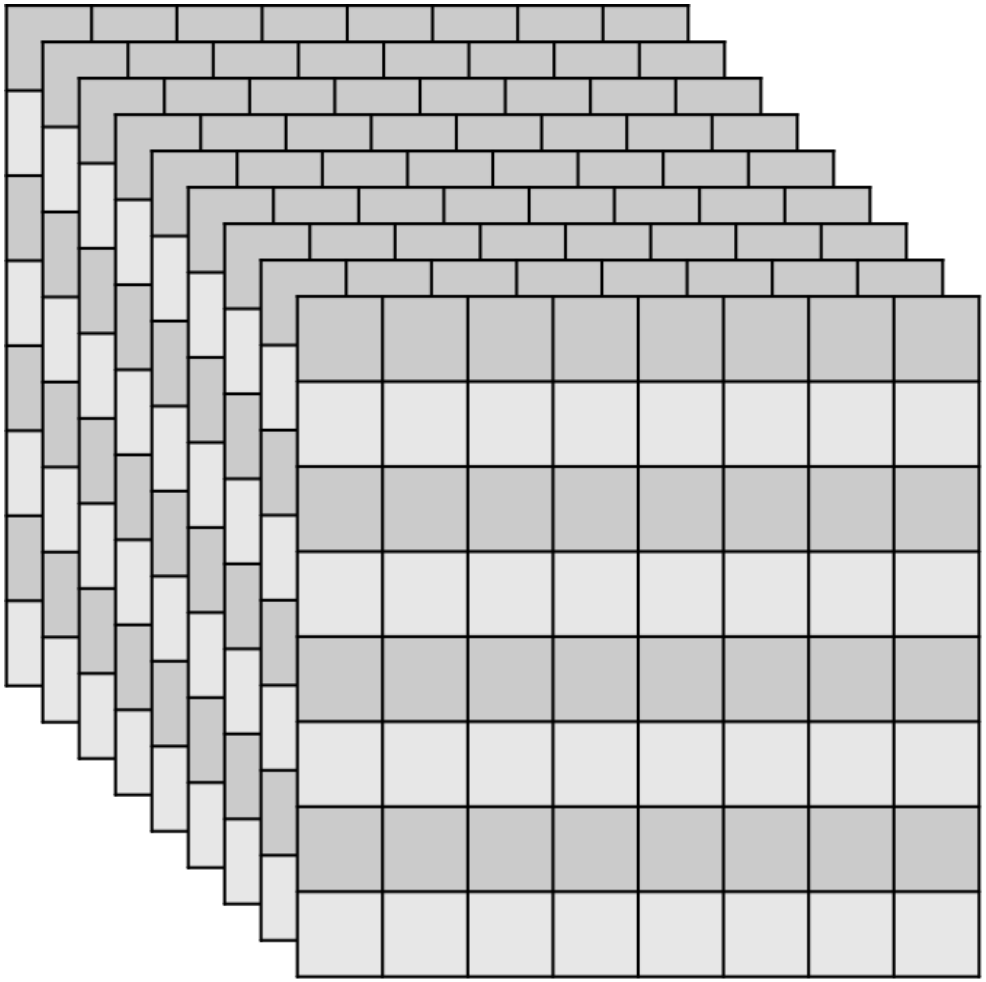
Convolution

ReLU

Pooling

Flattening

Linear



Convolutional Neural Network (Parts)

Convolution

ReLU

Pooling

Flattening

Linear

-32	-31	-30	-29	-28	-27	-26	-25
-24	-23	-22	-21	-20	-19	-18	-17
-16	-15	-14	-13	-12	-11	-10	-9
-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

Convolutional Neural Network (Parts)

Convolution

ReLU

Pooling

Flattening

Linear

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

Convolutional Neural Network (Parts)

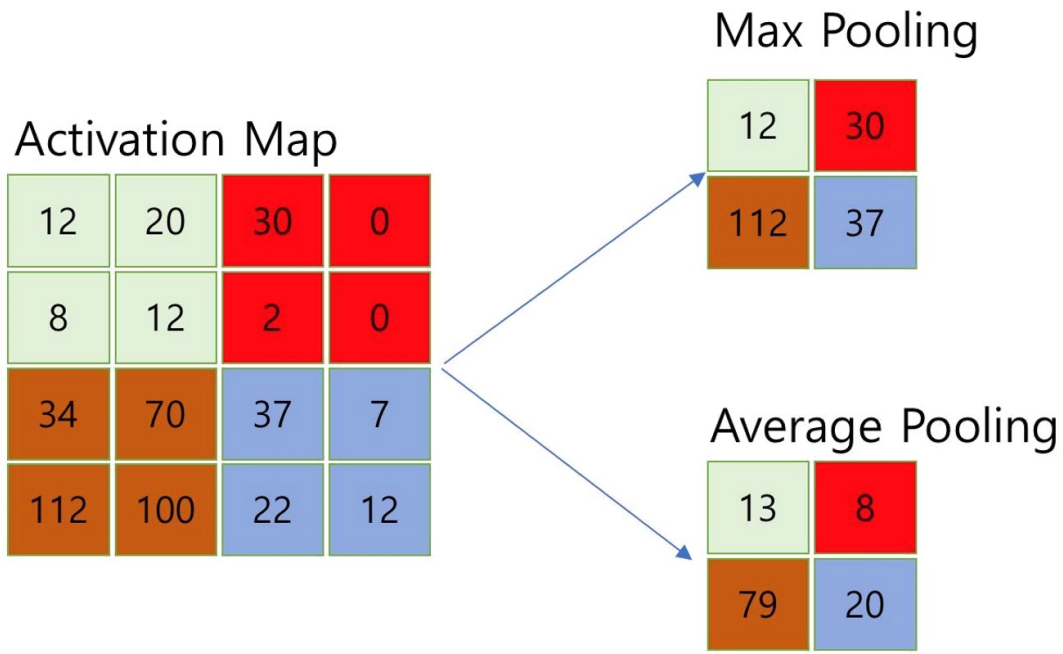
Convolution

ReLU

Pooling

Flattening

Linear



Convolutional Neural Network (Parts)

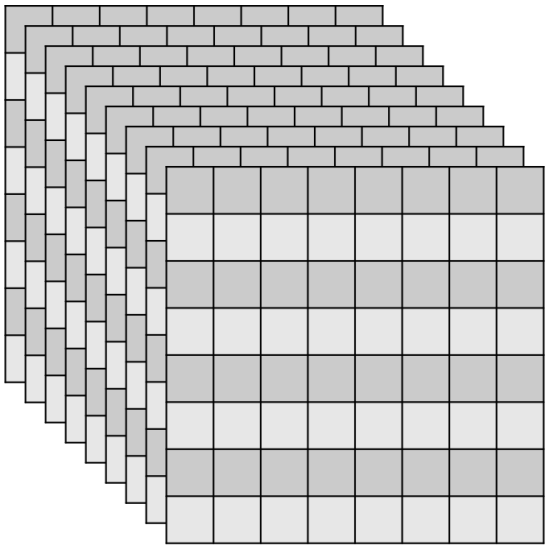
Convolution

ReLU

Pooling

Flattening

Linear



Convolutional Neural Network (Parts)

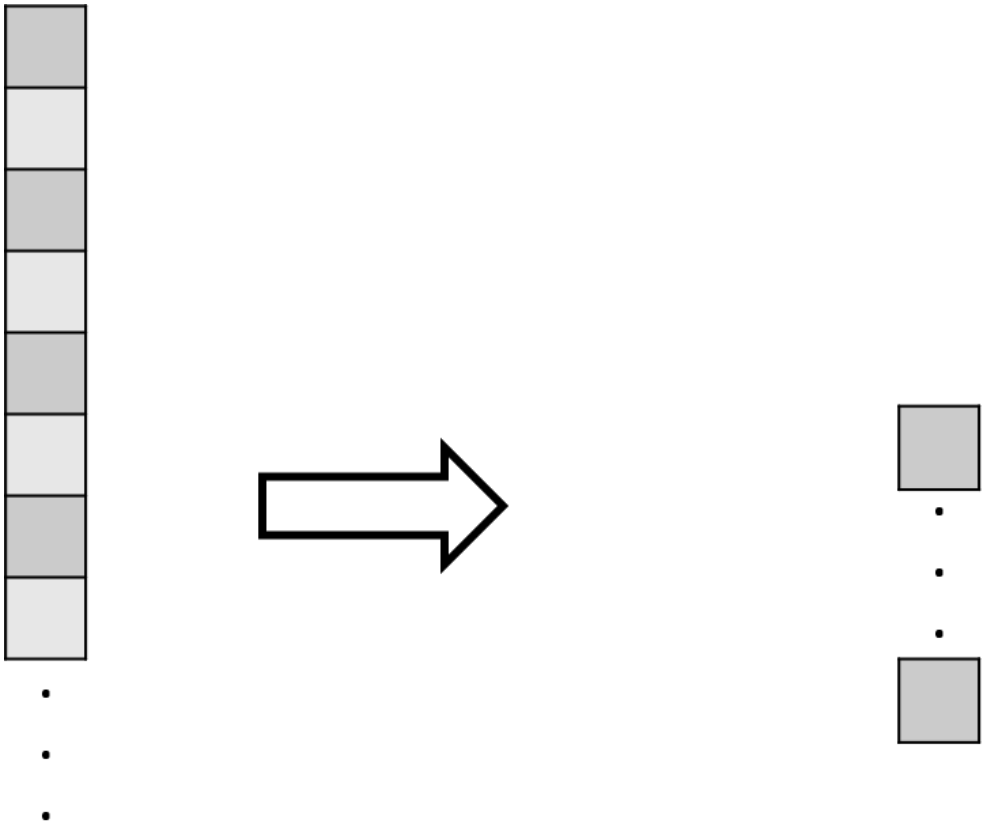
Convolution

ReLU

Pooling

Flattening

Linear

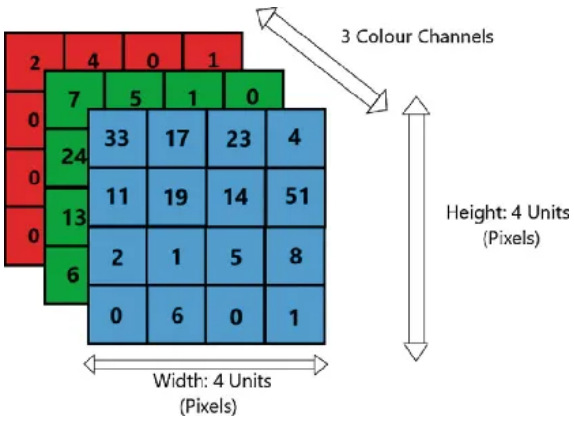


Software/Hardware

Software/Hardware

Software/Hardware

Major Data Types

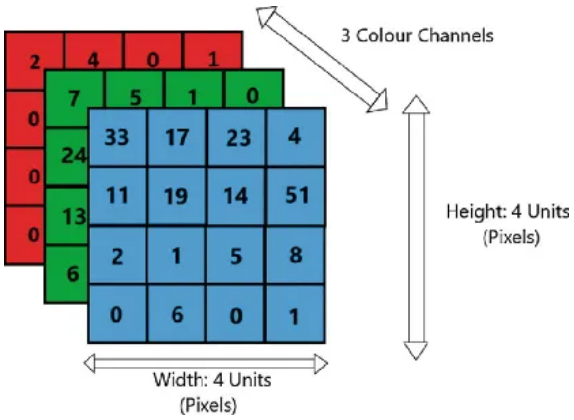


- Tensors: Multidimensional array used by PyTorch

  .
- Stores primitive data types (32/64 bit Bool, Int, and Float)

  .
- Necessary for all PyTorch processes

Major Data Types



PyTorch

Data Types, Funtions

- Tensors: Multidimensional array used by PyTorch

  .
- Stores primitive data types (32/64 bit Bool, Int, and Float)

  .
- Necessary for all PyTorch processes
- Inputs and targets are stored in DATASET (tensor input)
- Dataloader takes the dataset and generate training sets
- Set a batch size that fits your data

Convolutional Neural Network (Model)

```
    def __init__(self):
        super(EnergyPredictor, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)  # (8x8 -> 8x8)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1) # (8x8 -> 8x8)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1) # (8x8 -> 8x8)
        self.pool = nn.MaxPool2d(2, 2)                                     # (8x8 -> 4x4)

        # Fully connected layers for energy prediction
        self.fc1 = nn.Linear(64 * 4 * 4, 256)                              # Flatten ->
            Dense
        self.fc2 = nn.Linear(256, 1)                                       # Energy
            regression
```

Convolutional Neural Network (Model)

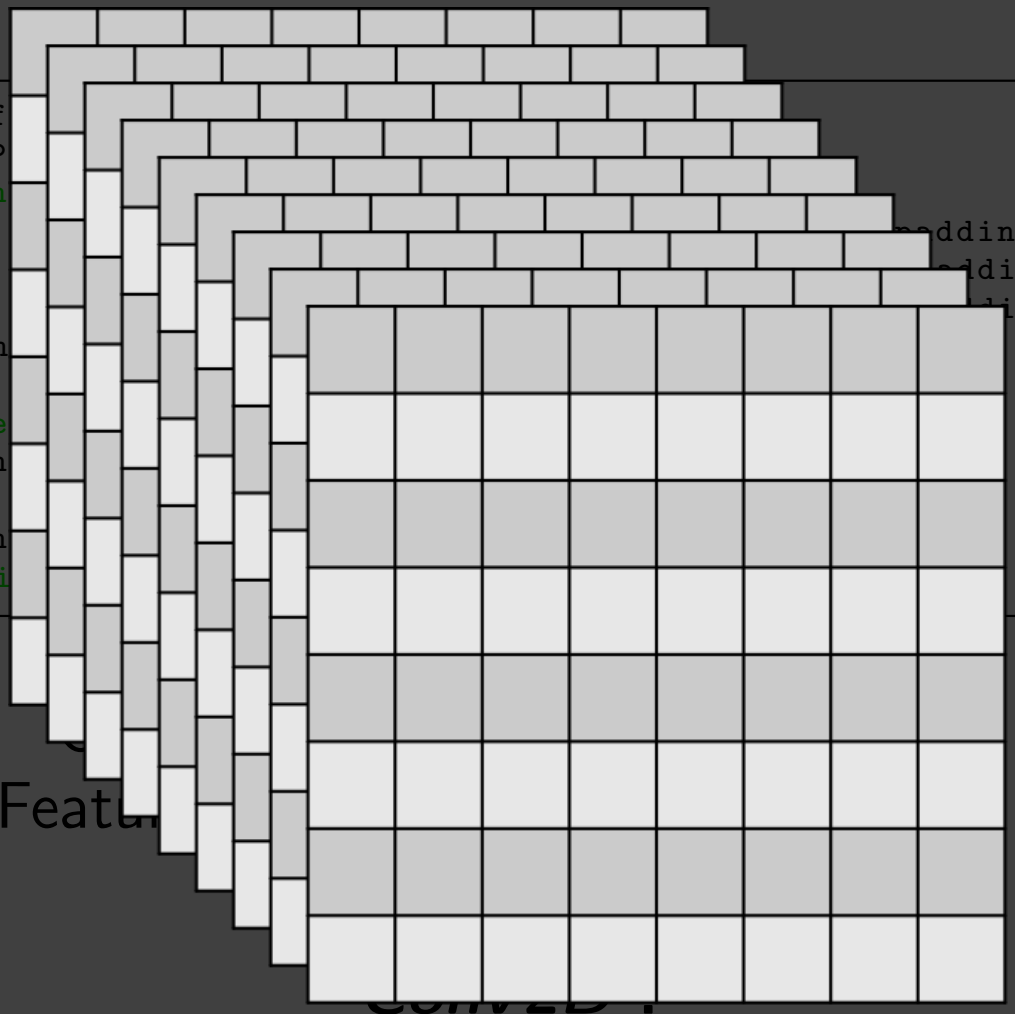
```
    def __init__(self):
        super(EnergyPredictor, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)  # (8x8 -> 8x8)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1) # (8x8 -> 8x8)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1) # (8x8 -> 8x8)
        self.pool = nn.MaxPool2d(2, 2)                                     # (8x8 -> 4x4)

        # Fully connected layers for energy prediction
        self.fc1 = nn.Linear(64 * 4 * 4, 256)                              # Flatten ->
            Dense
        self.fc2 = nn.Linear(256, 1)                                       # Energy
            regression
```

In Features | Out Features | Kernel Size | Stride | Padding

$$Conv2D:$$

Convolutional Neural Network (Model)

```
    def __init__(self
        super(EnergyP
        # Convolution
        self.conv1 =                                    dding=1)  # (8x8 -> 8x8)
        self.conv2 =                                    dding=1)  # (8x8 -> 8x8)
        self.conv3 =                                    ding=1)  # (8x8 -> 8x8)
        self.pool = n                                            # (8x8 -> 4x4)

        # Fully conne
        self.fc1 = nn                                          # Flatten ->
             Dense
        self.fc2 = nn                                          # Energy
             regressi
```



In Features

Padding

Convolutional Neural Network (Model)

```
    def __init__(self):
        super(EnergyPredictor, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)  # (8x8 -> 8x8)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1) # (8x8 -> 8x8)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1) # (8x8 -> 8x8)
        self.pool = nn.MaxPool2d(2, 2)                                     # (8x8 -> 4x4)

        # Fully connected layers for energy prediction
        self.fc1 = nn.Linear(64 * 4 * 4, 256)                              # Flatten ->
            Dense
        self.fc2 = nn.Linear(256, 1)                                       # Energy
            regression
```

In Features | Out Features | Kernel Size | Stride | Padding

$$Conv2D:$$

Convolutional Neural Network (Model)

```
    def __init__(self):
        super(EnergyPredictor, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)  # (8x8 -> 8x8)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1) # (8x8 -> 8x8)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1) # (8x8 -> 8x8)
        self.pool = nn.MaxPool2d(2, 2)                                     # (8x8 -> 4x4)

        # Fully connected layers for energy prediction
        self.fc1 = nn.Linear(64 * 4 * 4, 256)                              # Flatten ->
            Dense
        self.fc2 = nn.Linear(256, 1)                                       # Energy
            regression
```

Pool Width

Stride

.

. *MaxPool2d* :

Convolutional Neural Network (Model)

```
    def __init__(self):
        super(EnergyPredictor, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)  # (8x8 -> 8x8)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1) # (8x8 -> 8x8)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1) # (8x8 -> 8x8)
        self.pool = nn.MaxPool2d(2, 2)                                     # (8x8 -> 4x4)

        # Fully connected layers for energy prediction
        self.fc1 = nn.Linear(64 * 4 * 4, 256)                              # Flatten ->
            Dense
        self.fc2 = nn.Linear(256, 1)                                       # Energy
            regression
```

Dense Layer | Regression Layer | In Features | Out Features

.

.

Linear :

Convolutional Neural Network (Model)

```
    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = self.pool(torch.relu(self.conv3(x)))  # Downsample to 4x4
        x = x.view(x.size(0), -1)  # Flatten
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)  # Final energy output
        return x
```

Custom Dataset

```
class EnergyDataset(Dataset):
    def __init__(self, events_file):
        self.events = read_event_file(events_file)
        self.energies = compute_energy(self.events)

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

    def __getitem__(self, idx):
        return self.events[idx].unsqueeze(0), self.energies[idx]  # Add channel dimension
              for CNN input
```

Custom Dataset

- Must inheret the 'Dataset' class
- __len__, __getitem__ must be overwritten

```
class EnergyDataset(Dataset):
    def __init__(self, events_file):
        self.events = read_event_file(events_fi
        self.energies = compute_energy(self.eve

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

    def __getitem__(self, idx):
        return self.events[idx].unsqueeze(0), self.energies[idx]  # Add channel dimension
              for CNN input
```

Training Function

```
def train_model(model, dataloader, criterion, optimizer, num_epochs=10):
    loss_history = []
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, targets in dataloader:
            inputs, targets = inputs.to(device), targets.to(device)

            # Forward pass
            outputs = model(inputs).squeeze()
            loss = criterion(outputs, targets)

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
        loss_history.append(running_loss/len(dataloader))
        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {running_loss / len(dataloader):.4f
            }")
    return loss_history
```

Training Function

```
def train_model(model, dataloader, criterion, 
    loss_history = []
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, targets in dataloader:
            inputs, targets = inputs.to(device)

            # Forward pass
            outputs = model(inputs).squeeze()
            loss = criterion(outputs, targets)

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
        loss_history.append(running_loss/len(dataloader))
        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {running_loss / len(dataloader):.4f
            }")
    return loss_history
```

- Model, Dataloader, Criterion, Optimizer, and num epochs
- Criterion - Loss function
- Optimizer - Optimization algorithm for backward pass
- Num Epochs - Number of rounds (epochs) of training

Training Function

```
def train_model(model, dataloader, criterion, 
    loss_history = []
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, targets in dataloader:
            inputs, targets = inputs.to(device)

            # Forward pass
            outputs = model(inputs).squeeze()
            loss = criterion(outputs, targets)

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
        loss_history.append(running_loss/len(dataloader))
        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {running_loss / len(dataloader):.4f
            }")
    return loss_history
```

- Forward Pass:
- Outputs - List of guesses made by machine
- Loss - Difference between guess and target, by criterion

Training Function

```
def train_model(model, dataloader, criterion, 
    loss_history = []
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, targets in dataloader:
            inputs, targets = inputs.to(device)

            # Forward pass
            outputs = model(inputs).squeeze()
            loss = criterion(outputs, targets)

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
        loss_history.append(running_loss/len(dataloader))
        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {running_loss / len(dataloader):.4f
            }")
    return loss_history
```

- Backward Pass:
- Backward - Finds loss per parameter
- Step - Adjusts parameters based on loss rate

Main Logic

```
if __name__ == "__main__":
    # File paths
    events_file = "output.txt"  # Replace with your events file

    # Step 1: Create the Dataset and DataLoader
    dataset = EnergyDataset(events_file)
    dataloader = DataLoader(dataset, batch_size=100, shuffle=True)

    # Step 2: Initialize the Model
    model = EnergyPredictor()

    # Step 3: Define Loss Function and Optimizer
    criterion = nn.MSELoss()  # Mean Squared Error for regression
    optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)

    # Step 4: Train the Model
    device = (
        "cuda"
        if torch.cuda.is_available()
        else "mps"
        if torch.backends.mps.is_available()
        else "cpu"
    )
```

Main Logic

```
if __name__ == "__main__":
    # File paths
    events_file = "output.txt"  # Replace with

    # Step 1: Create the Dataset and DataLoade
    dataset = EnergyDataset(events_file)
    dataloader = DataLoader(dataset, batch_size

    # Step 2: Initialize the Model
    model = EnergyPredictor()

    # Step 3: Define Loss Function and Optimize
    criterion = nn.MSELoss()  # Mean Squared E
    optimizer = torch.optim.Adam(model.paramete

    # Step 4: Train the Model
    device = (
        "cuda"
        if torch.cuda.is_available()
        else "mps"
        if torch.backends.mps.is_available()
        else "cpu"
    )
```

- Load data to dataset
- Initialize dataloader and set batch parameters
- Initialize the model
- Choose Loss function and Optimizer
- Set learning rate
- Set processor

Main Logic

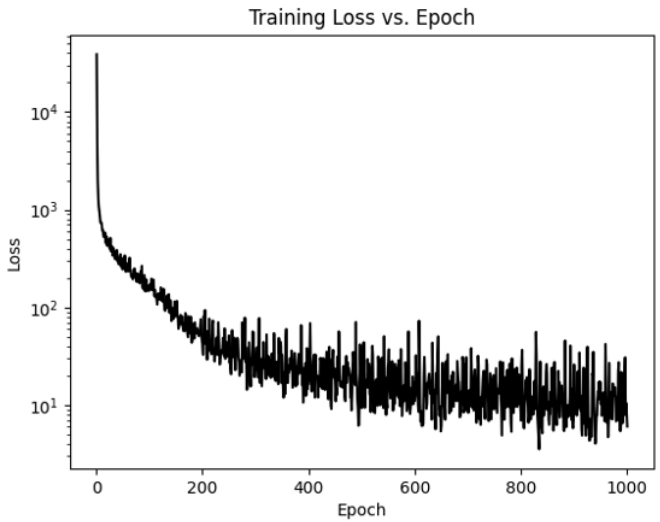
```
    print(f"Using␣device:␣{device}")
    model.to(device)
    loss_history = train_model(model, dataloader, criterion, optimizer, num_epochs=5000)
    np.savetxt("Loss.txt", loss_history, delimiter=",", fmt="%d")

    plt.plot(range(1, len(loss_history) + 1), loss_history, color='k', linestyle='-')#,
        linestyle='-')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training␣Loss␣vs.␣Epoch')
    plt.yscale('log')
    plt.savefig("Loss_100_.0001_5000.png")


    # Step 5: Save the Model
    torch.save(model.state_dict(), "single_photon_final.pth")
    print("Model␣training␣complete␣and␣saved!")
```

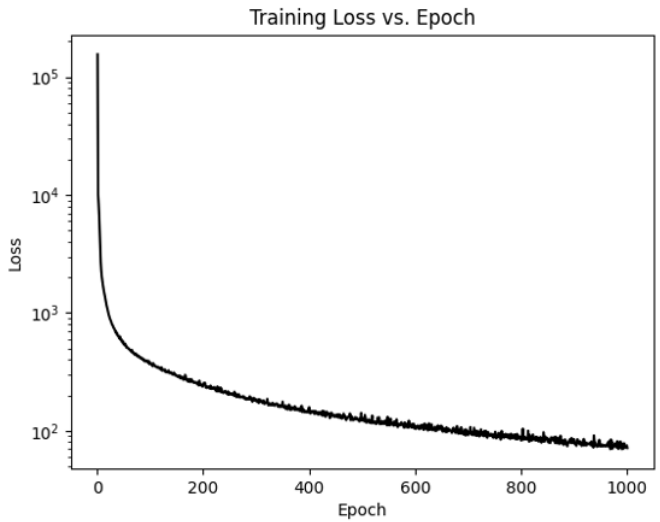
Main Logic

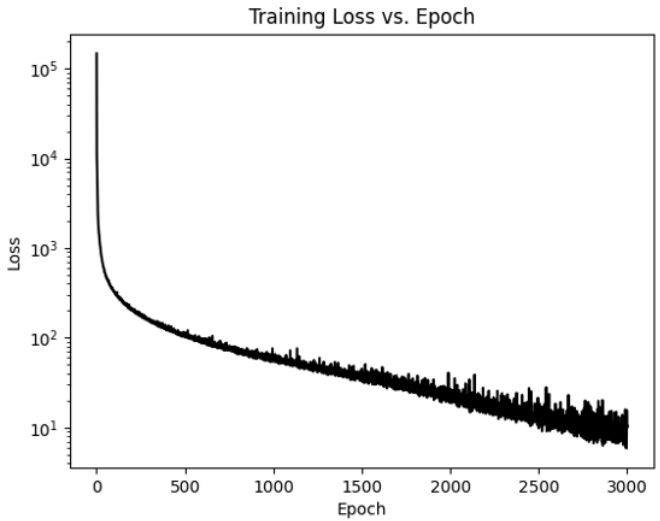
```
print(f"Using device: {device}")
model.to(device)
loss_history = train_model(model, dataloade
np.savetxt("Loss.txt", loss_history, delim

plt.plot(range(1, len(loss_history) + 1),
     linestyle='-')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss vs. Epoch')
plt.yscale('log')
plt.savefig("Loss_100_.0001_5000.png")


# Step 5: Save the Model
torch.save(model.state_dict(), "single_photon_final.pth")
print("Model training complete and saved!")
```

- Train model and record all losses
- Save loss data (plots, text files)
- Save the model to be accessed for other programs

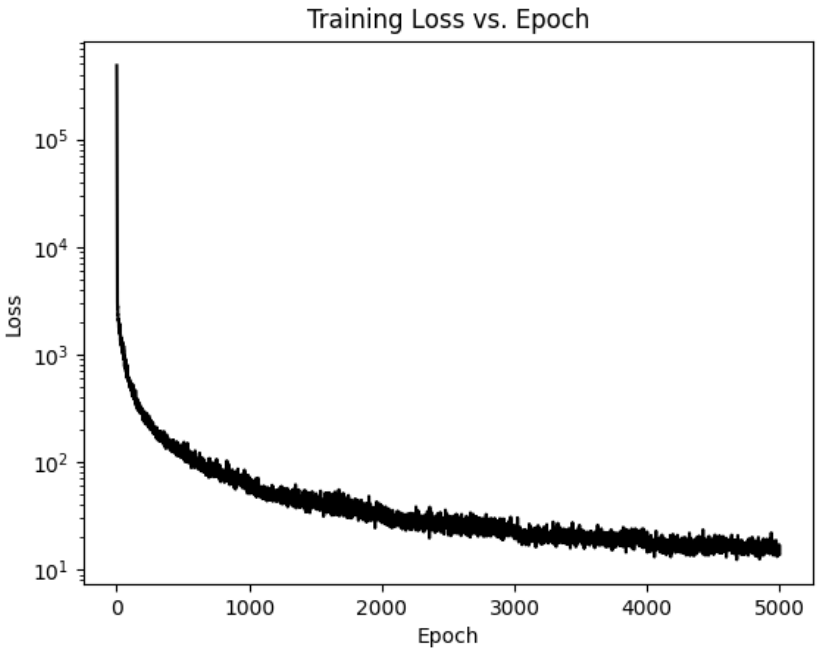
My Data


Training Loss vs. Epoch
Loss
Epoch
10^4
10^3
10^2
10^1
0
200
400
600
800
1000


Training Loss vs. Epoch
Loss
Epoch
10^5
10^4
10^3
10^2
0
200
400
600
800
1000


Training Loss vs. Epoch
Loss
Epoch
10^5
10^4
10^3
10^2
10^1
0
500
1000
1500
2000
2500
3000

Future Work



- Introduce stepped learning rate
- Optimize code
- Incorporate multi-photon events

Enhancing the Resolution of the MUSE Beamline Calorimeter Using Machine Learning

Charlie Brown

January 10, 2025

Stony Brook University