SmartPixels Meeting Report - 11/03/2026

Mila Bileska, Lino Gerlach, Isobel Ojalvo

Introduction

Motivation

- LGNs well suited for: small models deployed on ASICs
- Our goal: Evaluate LGNs for SmartPixels

What we have done

- Reproduce public SmartPixels results with conventional models (*)
- Design & train LGNs w/ comparable performance

Purpose of this talk

- Clarify some questions on public results w/ conventional NNs
- Show some (very) preliminary performance results of LGNs
- Discuss next steps (in particular ASIC synthesis)

About the Dataset

- Public “Smart pixel dataset” (id 10783560, v3) [on zenodo]
  - Simulated CMS silicon pixel detector clusters from charged pions
  - 20×13×21 (t × y × x) video-like charge deposit data
  - ~6M samples
  - Labels: track direction, momentum, charge, hit coordinates (16 labels)

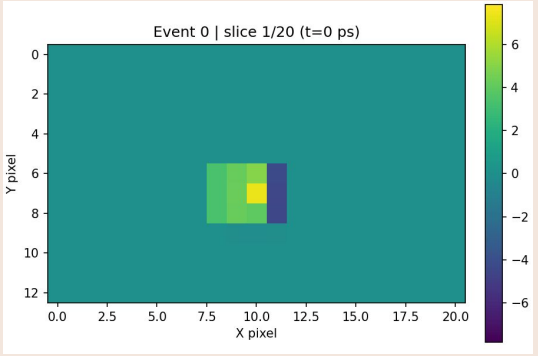

Figure 1: Example event display, event 0.

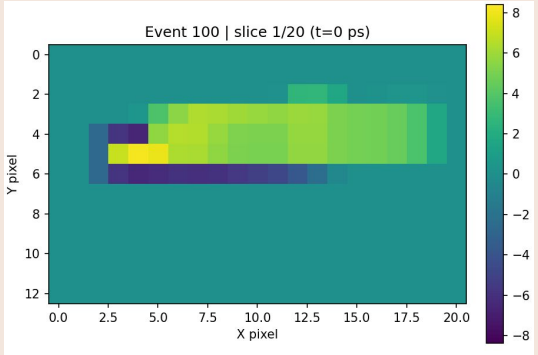

Figure 2: Example event display, event 100.

Smart pixel dataset

Morris Swartz[1] ; Jennet Dickinson[2]

Contributors

Project leader: Farah Fahim[1]

Project members:

Doug Berry[1] ; Giuseppe Di Guglielmo[1] ; Lindsey Gray[1] ; Jim Hirschauer[1] ; Shruti Kulkarni[2] ; Ronald Lipton[1] ; Petar Maksimovic[3] ; Corrinne Mills[4] ; Benjamin Parpillon[1] ; Gauri Pradhan[1] ; Nhan Tran[1] ; Jieun Yoo[4] ; Gandrakota, Abhijith[1] ; Syal, Chinar[5] ; Wen, Dahai[3] ; Young, Aaron[2] ; Kulkarni, Shruti[2] ; Badea, Anthony[6] ; DiPetrillo, Karri[6] ; Kumar, Carissa[6] ; Pan, Emily[6] ; Kovach-Fuentes, Rachel[6] ; Neubauer, Mark[7] ; Bean, Alice[8]

SmartPixel Model Inputs and Goals

- Following "*Smart Pixel Sensors: Towards On-Sensor Filtering of Pixel Clusters with Deep Learning*" (arxiv:2310.02474)

- **Goal:** Reject low-pt tracks (< 2 GeV), while keeping >90% of high-pt tracks

MACHINE LEARNING
Science and Technology

PAPER • OPEN ACCESS

Smart pixel sensors: towards on-sensor filtering of pixel clusters with deep learning

Jieun Yoo*, Jennet Dickinson, Morris Swartz, Giuseppe Di Guglielmo, Alice Bean, Douglas Berry, Manuel Blanco Valentin, Karri DiPetrillo, Farah Fahim, Lindsey Gray Show full author list

Published 14 August 2024 • © 2024 The Author(s). Published by IOP Publishing Ltd

- Train 3-way classifier:
  - low-pt(-), low-pt(+), or high-pt [>0.2 GeV]

- Three different model complexities
  - From dense models on high-level features to conv models on several time slices

Inputs and Models from the “towards…” Paper

Y-Profile: Dense Model (“Model 2”)

- **Input =** 14 features:
  - 13 y-profile values: a sum over x pixels, and
  - 1 y0 value: the location of the hit on the sensor array.
- **Design:** fully connected dense layers
- 2307 trainable params

Y-Profile + Timing: CNN (“Model 3”)

- **Input =** 105 features:
  - 13x8 y-profile values over the first 8 time slices, and
  - 1 y0 value: the location of the hit on the sensor array.
- **Design:** Convolution + dense layers
- 83 331 trainable params

We ignore the 2 input-feature version “Model 1” in the following

All following plots use full-precision (unquantized) features, params & ops

Note: only the middle ground, “Model 2” in synthesized

Comparison to Original Paper

Replicated all other training details beyond model architecture: train/val/test split, batch size, learning rate, …

Observation:

- Our baseline models have lower acceptance compared to the paper
- Baseline CNN performs closer to original results than Baseline Dense.

Ours

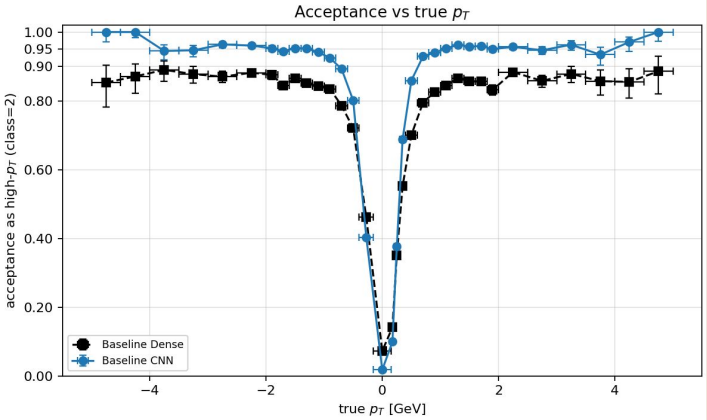

Figure 1: Classifier acceptance as a function of pT for Baseline Dense and Baseline CNN

Paper

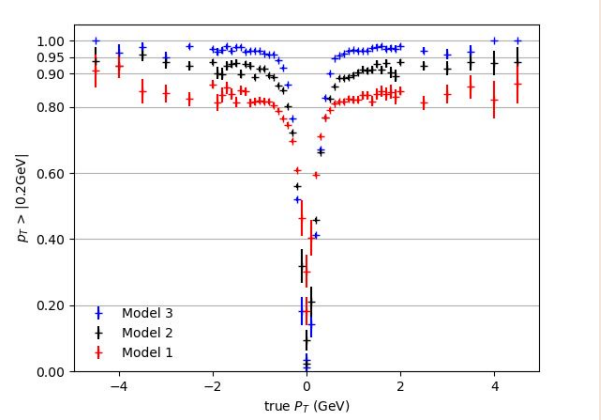

Figure 6 from (arXiv:2310.0247): Classifier acceptance as a function of pT. **Model 3** corresponds to **Baseline CNN** , **Model 2** corresponds to **Baseline Dense** , and Model 1 is a 2-input model not discussed in our studies.

Potential Differences:

- We use V3 of the dataset, paper uses V1,
- We use Pytorch instead of Keras,
- We use weighted samples instead of discarding.

LGN Architecture

Developed two LGNs:

- “LGN Dense”: operates on the same 14 input features as “Model 2”
- “LGN Convolutional”: operates on same inputs as “Model 3”

LGNs have more layers and neurons than their conventional counterparts

LGNs operate on purely boolean data -> need to “binarize” inputs

- We represent each input feature as 100 bits
  - E.g.: 8x14 floats -> 8x14x100 bools
- This can be heavily optimized (e.g. with learnable thresholding)

LGN Dense Compared to Baseline Dense

Dense Model

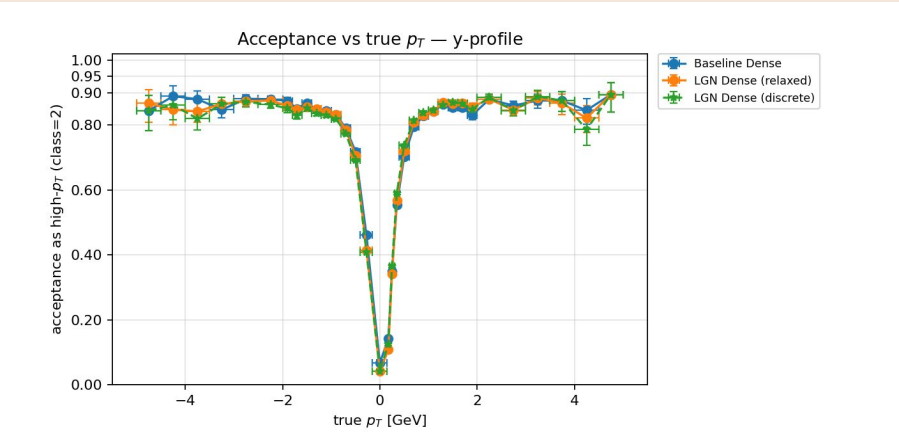

Figure 1: Acceptance vs True pT for Baseline Dense and LGN Dense (relaxed + discrete)

Conv Model

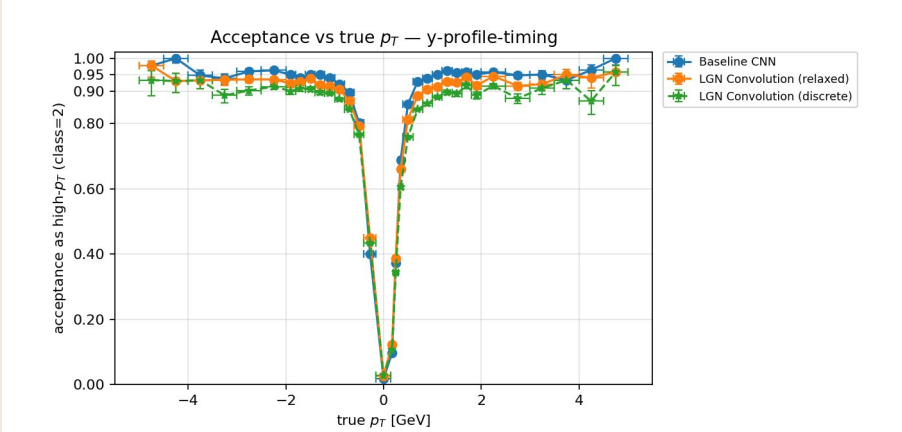

Figure 2: Acceptance vs True pT for Baseline CNN and LGN Convolution(relaxed + discrete)

Notes on figures:

- Relaxed: Model built using soft, differentiable logic during training,
- Discrete: Pure logic-gate architecture which will be implemented in ASICs.

Observations:

- Found LGNs that perform as well as conventional NNs
- Relaxed and discrete performance agrees very well
- Small gap only for conv model

ASIC #Gates Estimate Comparison

Want to check if our LGNs are reasonably sized w/o (emulating) synthesis.
-> Compare to baseline via **rough gate count** estimate

Baseline Estimate:

- Counting MAC (Multiply–Accumulate) operations in layers, then multiplying by an approximate gate-equivalent cost per MAC
- Also used reported chip area + transistor density -> same order of magnitude

LGN Estimate:

- Trainable parameters divided by 16 (number of nodes on logic DAG before any optimizations)

Model Type **(Dense)**	# of Gates
Baseline	O(10e5)
LGN	O(10e5)

Table 2: Gates Estimate for Baseline Dense and LGN Dense

Conservative LGN size estimate is in the same ball park as quantized baseline model
(before any optimizations)

Conclusion, Outlook & Open Questions

Conclusion

- (Partially) reproduced public results w/ conventional NNs
- Demonstrated comparable performance w/ LGNs
- Very rough estimate: Feasible computational complexity when deployed

Outlook

- Will work on synthesis next
- First preliminary results. Plenty of further optimizations possible

Open Questions

- Any ideas why we cannot exactly reproduce public results?
- How to proceed with synthesis?
    - We can generate Verilog code for any LGN. Started looking into OpenLane

Backup

Model Definitions Example: Our Conv LGN

```python
@register("lgn-conv-3-100-orpool_128_20", input_type="y-profile-timing", task_type="classification")
class ConvLGNDeep100_2_128_20(torch.nn.Module):
    n_bits = 100

    def __init__(self, thresholds):
        super().__init__()
        device = "cuda"
        self.n_bits = 100
        self.bin = FixedBinarization(thresholds=thresholds)
        pk = {"weight_init": "residual"}
        td = 3

        k1, k2, k3, k4 = 256, 512, 1024, 960

        self.conv1 = LogicConv2d(
            in_dim=(13, 8), channels=self.n_bits, num_kernels=k1,
            receptive_field_size=3, tree_depth=td, stride=1, padding=1,
            device=device, parametrization_kwargs=pk,
        )
        self.conv2 = LogicConv2d(
            in_dim=(13, 8), channels=k1, num_kernels=k2,
            receptive_field_size=3, tree_depth=td, stride=1, padding=1,
            device=device, parametrization_kwargs=pk,
        )

        self.or_pool1 = OrPooling2d(kernel_size=2, stride=2, padding=0)

        self.conv3 = LogicConv2d(
            in_dim=(6, 4), channels=k2, num_kernels=k3,
            receptive_field_size=3, tree_depth=td, stride=1, padding=1,
            device=device, parametrization_kwargs=pk,
        )
        self.conv4 = LogicConv2d(
            in_dim=(6, 4), channels=k3, num_kernels=k4,
            receptive_field_size=3, tree_depth=td, stride=2, padding=1,
            device=device, parametrization_kwargs=pk,
        )

        fc_in = self.n_bits + (k4 * 3 * 2)

        self.fc1 = LogicDense(in_dim=fc_in, out_dim=3000*4, device=device, parametrization_kwargs=pk, lut_rank=2)
        self.fc2 = LogicDense(in_dim=3000*4, out_dim=3000*4, device=device, parametrization_kwargs=pk, lut_rank=2)
        self.fc3 = LogicDense(in_dim=3000*4, out_dim=3000*2, device=device, parametrization_kwargs=pk, lut_rank=2)
        self.fc4 = LogicDense(in_dim=3000*2, out_dim=3000, device=device, parametrization_kwargs=pk, lut_rank=2)

        self.group_sum = GroupSum(3, tau=20.0, device=device)
```

```python
    def forward(self, x):
        x = self.bin(x)
        y0 = x[:, 0:self.n_bits]
        p = x[:, self.n_bits:].view(-1, self.n_bits, 13, 8)

        z = self.conv1(p)
        z = self.conv2(z)
        z = self.or_pool1(z)
        z = self.conv3(z)
        z = self.conv4(z)

        z = z.view(z.size(0), -1)
        z = torch.cat((y0, z), dim=1)
        z = self.fc4(self.fc3(self.fc2(self.fc1(z))))
        return self.group_sum(z)
```

Explicit Model Definitions: LGN

```python
@register("lgn-conv-3-100-orpool_128_20", input_type="y-profile-timing", task_type="classification")
class ConvLGNDeep100_2_128_20(torch.nn.Module):
    n_bits = 100

    def __init__(self, thresholds):
        super().__init__()
        device = "cuda"
        self.n_bits = 100
        self.bin = FixedBinarization(thresholds=thresholds)
        pk = {"weight_init": "residual"}
        td = 3

        k1, k2, k3, k4 = 256, 512, 1024, 960

        self.conv1 = LogicConv2d(
            in_dim=(13, 8), channels=self.n_bits, num_kernels=k1,
            receptive_field_size=3, tree_depth=td, stride=1, padding=1,
            device=device, parametrization_kwargs=pk,
        )
        self.conv2 = LogicConv2d(
            in_dim=(13, 8), channels=k1, num_kernels=k2,
            receptive_field_size=3, tree_depth=td, stride=1, padding=1,
            device=device, parametrization_kwargs=pk,
        )

        self.or_pool1 = OrPooling2d(kernel_size=2, stride=2, padding=0)

        self.conv3 = LogicConv2d(
            in_dim=(6, 4), channels=k2, num_kernels=k3,
            receptive_field_size=3, tree_depth=td, stride=1, padding=1,
            device=device, parametrization_kwargs=pk,
        )
        self.conv4 = LogicConv2d(
            in_dim=(6, 4), channels=k3, num_kernels=k4,
            receptive_field_size=3, tree_depth=td, stride=2, padding=1,
            device=device, parametrization_kwargs=pk,
        )

        fc_in = self.n_bits + (k4 * 3 * 2)

        self.fc1 = LogicDense(in_dim=fc_in, out_dim=3000*4, device=device, parametrization_kwargs=pk, lut_rank=2)
        self.fc2 = LogicDense(in_dim=3000*4, out_dim=3000*4, device=device, parametrization_kwargs=pk, lut_rank=2)
        self.fc3 = LogicDense(in_dim=3000*4, out_dim=3000*2, device=device, parametrization_kwargs=pk, lut_rank=2)
        self.fc4 = LogicDense(in_dim=3000*2, out_dim=3000, device=device, parametrization_kwargs=pk, lut_rank=2)

        self.group_sum = GroupSum(3, tau=20.0, device=device)

    def forward(self, x):
        x = self.bin(x)
        y0 = x[:, 0:self.n_bits]
        p = x[:, self.n_bits:].view(-1, self.n_bits, 13, 8)

        z = self.conv1(p)
        z = self.conv2(z)
        z = self.or_pool1(z)
        z = self.conv3(z)
        z = self.conv4(z)

        z = z.view(z.size(0), -1)
        z = torch.cat((y0, z), dim=1)
        z = self.fc4(self.fc3(self.fc2(self.fc1(z))))
        return self.group_sum(z)
```

Figure 1: LGN Convolution Explicit Definition using Torchlogix Library

```python
@register("lgn-dense-2-dense-100_128_-lessFeat_40", input_type="y-profile", task_type="classification")
class DenseOnlyLGN_9F_Wide_128__40(torch.nn.Module):
    n_bits = 100
    k = 2000

    def __init__(self, thresholds):
        super().__init__()
        device = "cuda"
        pk = {"weight_init": "residual"}

        self.bin = FixedBinarization(thresholds=thresholds)

        in_dim = 9 * self.n_bits

        h1 = 6 * self.k
        h2 = 6 * self.k
        h3 = 6 * self.k
        h4 = 4 * self.k
        h5 = 4 * self.k
        h6 = 4 * self.k
        out_dim = 3 * self.k

        self.fc1 = LogicDense(in_dim=in_dim, out_dim=h1, device=device, parametrization_kwargs=pk)
        self.fc2 = LogicDense(in_dim=h1,     out_dim=h2, device=device, parametrization_kwargs=pk)
        self.fc3 = LogicDense(in_dim=h2,     out_dim=h3, device=device, parametrization_kwargs=pk)
        self.fc4 = LogicDense(in_dim=h3,     out_dim=h4, device=device, parametrization_kwargs=pk)
        self.fc5 = LogicDense(in_dim=h4,     out_dim=h5, device=device, parametrization_kwargs=pk)
        self.fc6 = LogicDense(in_dim=h5,     out_dim=h6, device=device, parametrization_kwargs=pk)
        self.fc7 = LogicDense(in_dim=h6,     out_dim=out_dim, device=device, parametrization_kwargs=pk)

        self.group_sum = GroupSum(3, tau=40.0, device=device)

    def forward(self, x):
        x = self.bin(x)
        z = self.fc7(self.fc6(self.fc5(self.fc4(self.fc3(self.fc2(self.fc1(x)))))))
        return self.group_sum(z)
```

Figure 2: LGN Dense Explicit Definition using Torchlogix Library

Questions on Reproducing Public Results

-

Questions on ASIC Synthesis

SmartPixel Model Inputs and Goals

- **Following procedure described in** "*Smart Pixel Sensors: Towards On-Sensor Filtering of Pixel Clusters with Deep Learning.*"
- **Goal:** Reject low-pt tracks (< 2 GeV, binary classifier)
- **Training Objective:** Predict whether a track is from a high-pt, low-pt(-), or low-pt(+) particle.
- Following the lead of Yoo, Jieun, et al. "*Smart Pixel Sensors: Towards On-Sensor Filtering of Pixel Clusters with Deep Learning.*" (arxiv:2310.02474), two model designs were tested:

Y-Profile Dense:

- **Input =** 14 features:
  - 13 y-profile values: a sum over x pixels, and
  - 1 y0 value: the location of the hit on the sensor array.
- **Design:** fully connected dense layers, no convolution (MLP).

Y-Profile + Timing CNN:

- **Input =** 105 features:
  - 13x8 y-profile values over the first 8 time slices, and
  - 1 y0 value: the location of the hit on the sensor array.
- **Design:** Convolution + dense layers.

LGN Dense Compared to Baseline Dense

Figure 1: Acceptance vs True pT for Baseline Dense and LGN Dense (relaxed + discrete)

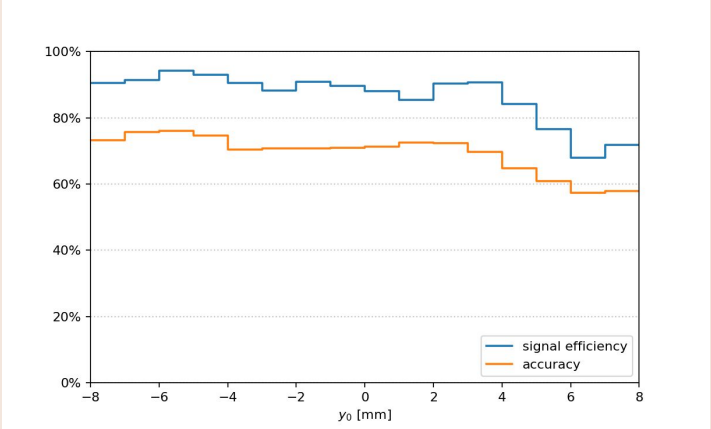

Figure 2: Efficiency and Accuracy per y0 Position for LGN Dense

Notes on figures:

- Relaxed: Model built using soft, differentiable logic during training,
- Discrete: Pure logic-gate architecture which will be implemented in ASICs.

About the LGN model:

- **Model Input:** y-profile + y0 (14 features).
- **Model Architecture:** LGN fully connected dense layers.
- **Observations:** Comparable accuracy, signal efficiency, and acceptance to the baseline.

LGN Convolution Compared to Baseline Dense

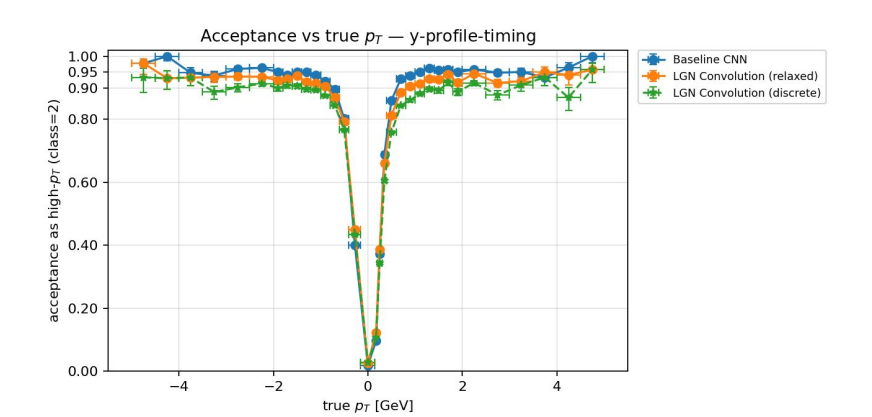

Figure 1: Acceptance vs True pT for Baseline CNN and LGN Convolution(relaxed + discrete)

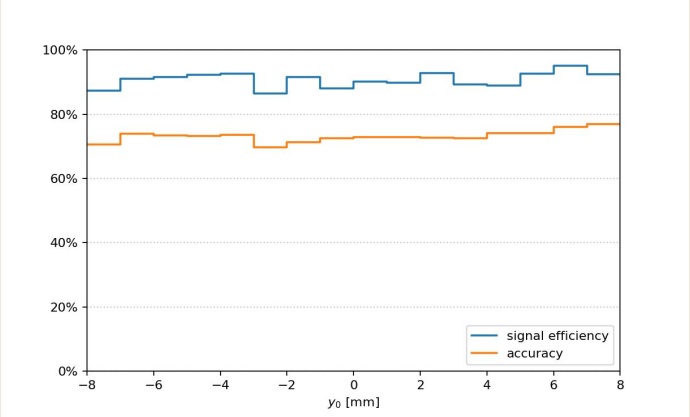

Figure 2: Efficiency and Accuracy per y0 Position for LGN Convolution

Notes on figures:

- Relaxed: Model built using soft, differentiable logic during training,
- Discrete: Pure logic-gate architecture which will be implemented in ASICs.

About the LGN model:

- **Model Input:** y-profile + y0 + timing (105 features).
- **Model Architecture:** LGN convolution and dense layers.
- **Observations:** Comparable accuracy, signal efficiency, and acceptance to the baseline, higher than LGN Dense.

Baseline Dense Model Results + Architecture

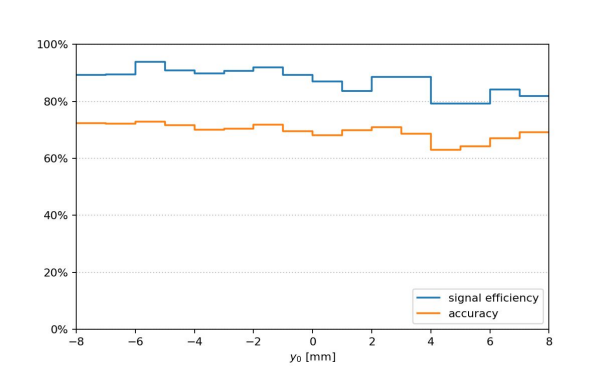

Figure 1: Efficiency and Accuracy per y0 Position for Baseline Dense

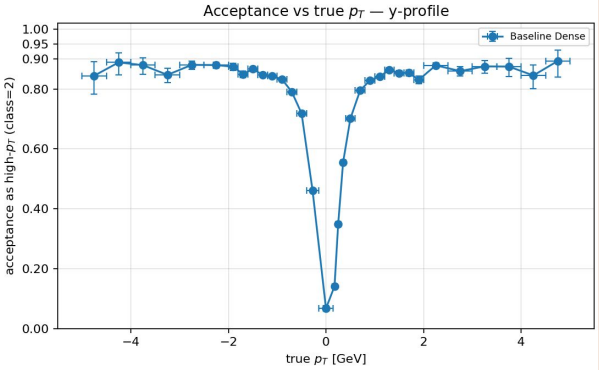

Figure 2: Acceptance vs True pT for Baseline Dense

```python
class TowardsModel2(torch.nn.Module):
    def __init__(self):
        super(TowardsModel2, self).__init__()
        self.fc1 = torch.nn.Linear(14, 128)
        self.fc2 = torch.nn.Linear(128, 3)
        self.softmax = torch.nn.Softmax(dim=1)

    def forward(self, x):
        assert x.shape == (x.shape[0], 14), f"Expected input shape (batch_size, 14), but got {x.shape}"
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Figure 3: Baseline Dense Architecture

Metric Definitions:

Acceptance:

$$\frac{\#(\text{predicted class} = 2 \text{ in bin})}{\#(\text{all tracks in } p_T \text{ bin})}$$

Efficiency:

$$\frac{\#(\text{predicted class} = 2 \ \wedge \ |p_T| > 2\,\text{GeV})}{\#(|p_T| > 2\,\text{GeV})}$$

Accuracy:

$$\frac{\#(\text{predicted class} = \text{true class})}{\#(\text{all tracks in that } y_0 \text{ bin})}$$

About the model:

- **Model Input:** y-profile + y0 (14 features).
- **Model Architecture:** Conventional Tensorflow MLP (unquantized).
- **Observations:** Lower accuracy, signal efficiency, and acceptance when compared to the CNN model, but also less input features.

Baseline CNN Model Results + Architecture

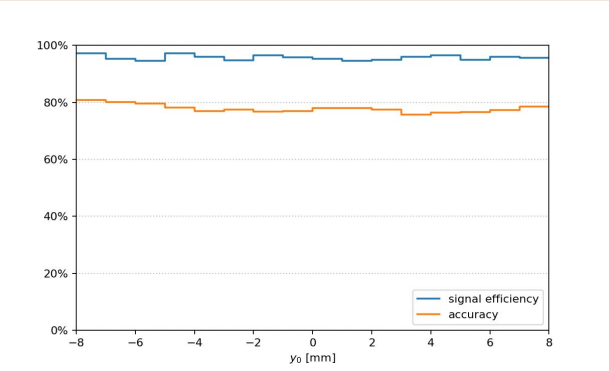

Figure 1: Efficiency and Accuracy per y0 Position for Baseline CNN

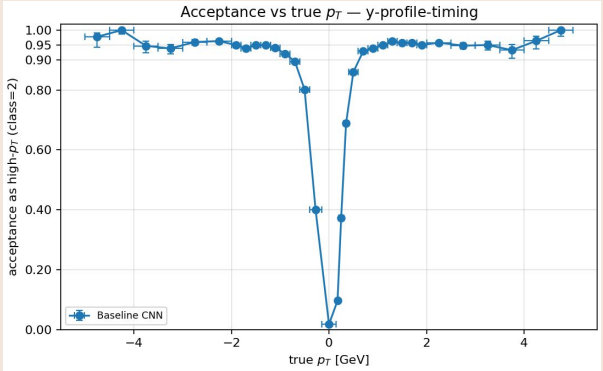

Figure 2: Acceptance vs True pT for Baseline CNN

```python
class TowardsModel13(torch.nn.Module):
    def __init__(self):
        super(TowardsModel13, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 16, kernel_size=3, stride=1)
        self.conv2 = torch.nn.Conv2d(16, 64, kernel_size=3, stride=1)
        self.fc1 = torch.nn.Linear(64 * 9 * 4 + 1, 32)
        self.dropout = torch.nn.Dropout(0.1)
        self.fc2 = torch.nn.Linear(32, 3)
        self.softmax = torch.nn.Softmax(dim=1)

    def forward(self, x):
        assert x.shape == (x.shape[0], 105), f"Expected input shape (batch_size, 105), but got {x.shape}"

        y0 = x[:, 0:1]
        profile = x[:, 1:].view(-1, 1, 13, 8)

        x = torch.relu(self.conv1(profile))
        x = torch.relu(self.conv2(x))

        x = x.view(x.size(0), -1)
        x = torch.cat((y0, x), dim=1)

        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x
```

Figure 3: Baseline CNN Architecture

Metric Definitions:

Acceptance:

$$\frac{\#(\text{predicted class} = 2 \text{ in bin})}{\#(\text{all tracks in } p_T \text{ bin})}$$

Efficiency:

$$\frac{\#(\text{predicted class} = 2 \ \wedge \ |p_T| > 2\,\text{GeV})}{\#(|p_T| > 2\,\text{GeV})}$$

Accuracy:

$$\frac{\#(\text{predicted class} = \text{true class})}{\#(\text{all tracks in that } y_0 \text{ bin})}$$

About the model:

- **Model Input:** y-profile + y0 + timing (105 features).
- **Model Architecture:** Conventional Tensorflow CNN (unquantized).
- **Purpose:** Provide a baseline for the LGN Convolution model.
- **Observations:** Overall high accuracy, signal efficiency, and acceptance.

ASIC #Gates Estimate Comparison

Baseline #Gates Estimate:

- Counting MAC (Multiply–Accumulate) operations in layers, then multiplying by an approximate gate-equivalent cost per MAC.

LGN #Gates Count:

- Trainable parameters divided by 16.

Observation:

- The LGN Convolution has fewer gates than the LGN Dense with much higher performance. Granularity is a more significant limiting factor than model complexity. (LGNs smaller than baseline)

Model Type **(Convolution)**	# of Gates
Baseline	O(10e8)
LGN	52,264

Table 1: Number of Gates Estimate for Baseline CNN and Convolution

Model Type **(Dense)**	# of Gates
Baseline	O(10e5)
LGN	62,500

Table 2: Number of Gates Estimate for Baseline Dense and Dense