3D CNNS FOR MUON REGRESSION

Giles Strong, Tommaso Dorigo, Lukas Layer

Quarks to Cosmos, Online - 13/07/21

OVERVIEW

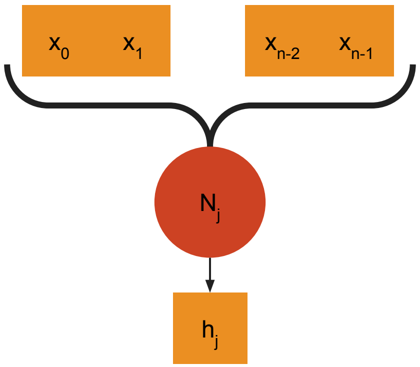
- Convolutional Kernels
- CNNs for muon regression
- Paper architecture

CONVOLUTIONAL KERNELS

LINEAR TRANSFORMATION

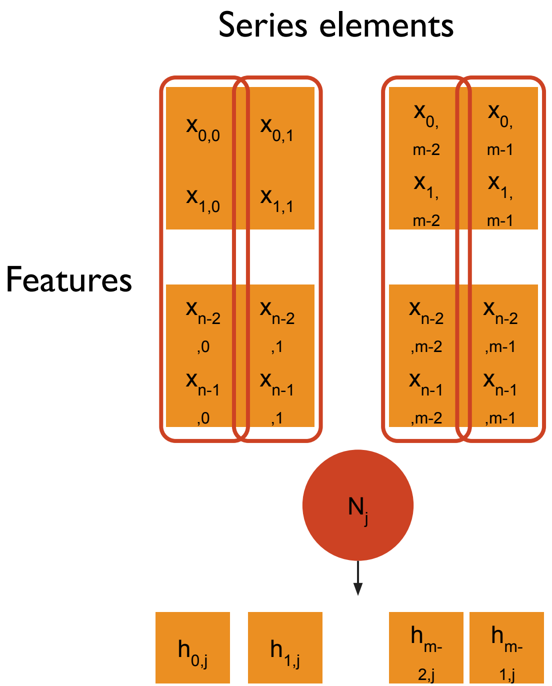
- Given a vector of *n* features $\underline{x}$, a neuron (N_j) in a fully connected layer transforms the vector according to:
- $$\bar{w_j} \cdot \bar{x} + b_j \equiv \sum_{i=0}^{n-1} [w_{j,i} \times x_i] + b_j = h_j$$
- Where $\underline{w}_j$ and b_j are the weight vector and bias parameters of neuron N_j
- Other neurons have their own weights and biases, and so produce differing values of elements in the output *h*:

$$\mathbf{W} \cdot \bar{x} + \bar{b} = \bar{h}$$



SERIES DATA

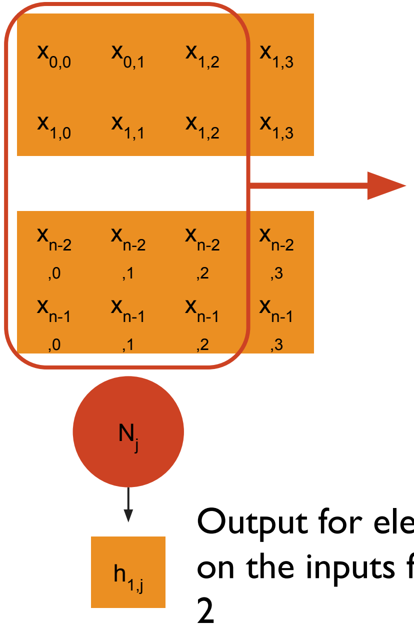
- Consider now if the input data consist of a series of *m* elements, each with their own set of features $\underline{x}$
- We can apply a fully-connected layer to this data by transforming each feature vector separately
  - The linear transformation is *convolved* over the 1D series



KERNEL SIZE

- In the previous case, we can refer to the neuron as having *kernel size* of 1
  - For each transformation, it only considers the features of the current element of the 1D series of data
- If our series is ordered somehow, we can instead base the output of the neuron on the neighbourhood of elements, not just the current element
  - The weight of N_j is now a 2D matrix but is still applied as an element-wise product and sum to the inputs inside the kernel

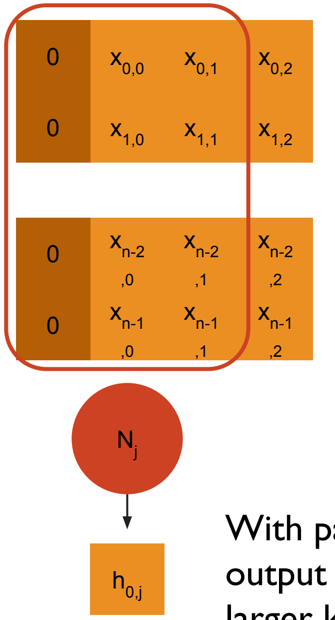
$$\sum \left[\mathbf{W}_j \odot \bar{x}\right] + b_j = h_j$$



PADDING

- With a kernel size > 1, we are unable to compute outputs for elements near the start and end of the series
  - Either we can live with it and have an output with few elements than our input
  - Or we can *pad* the input series with sufficient elements to allow us to compute outputs for all elements.
- Common approaches:
  - Zero padding - elements have feature values of zero
  - Reflection padding - imagine a mirror at the edge, the padding elements' features are equal to the mirror reflection

Kernel size = 3, centred on element 0

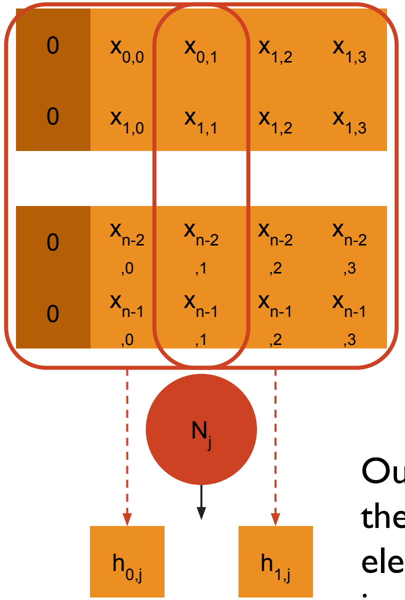


With padding, we can compute an output of element 0, even with a larger kernel

STRIDE

- So far we have stepped the kernel such that it is centred over each series element in turn
    - A *stride-1 kernel* - the kernel is shifted by a distance of 1 between applications
- The stride can be varied, e.g. a stride-2 kernel would be applied on every other series element
    - The resulting output would only have half the number elements as the original series
    - Through using larger kernels, the output can still depend on all the input elements, even if a kernel is not centred on them
    - This allows the series to be *downsampled*

Kernel size = 3, stride = 2,



Kernel skips out element 1

Output only has half the number of elements as the input

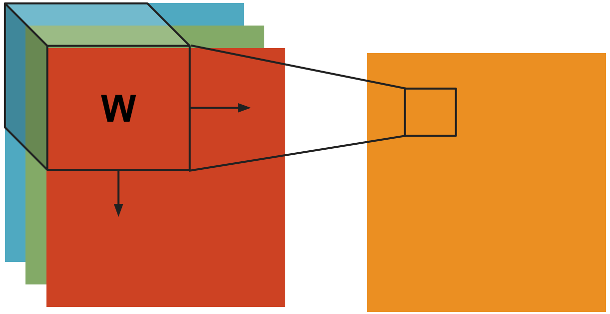
POOLING

- An alternative method of downsampling is to *pool* elements
  - Within a kernel, compute a fixed, order-invariant function of the features
  - E.g. maximum value or mean value
- Not as powerful as downsampling by strided convolutions
  - But still is sometimes useful

- Adaptive pooling:
  - Kernel size adapts to requested number of output elements
  - Useful when series vary in length by ensuring the same number of outputs are always produced

HIGHER-ORDER CONVOLUTIONS

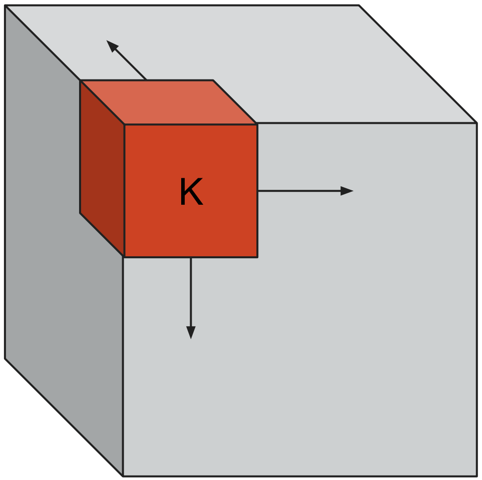
- So far just considered 1D series of elements with features
- Sometimes data contains an inherent structure which makes it beneficial to work with in a higher number of dimensions
  - E.g. pixels in an image naturally have a 2D layout
  - Could reshape into a 1D series by concatenating rows of pixels, but connections to pixels above and below would be lost



Same concepts still apply
But each neuron now has a rank-3 tensor for the weight: (kernel size x, kennel size y, number of features (channels))

HIGHER-ORDER CONVOLUTIONS

- In the case of 3D data, the kernel is now also 3D, with a rank-4 weight tensor
- Channel dimension not shown for data or kernel



BATCHNORM

- Parameter initialisation schemes expect unit-Gaussian inputs, but signals can diverge from this as they pass through the network
- Batchnorm renormalises signals
  - Reduces *internal covariate shift*
- Process also applies learnable rescaling and offset to renormalised signal
  - Can be used to unlearn renormalisation (BN acts as identity function)
  - But also provides convenient parameters to rescale and shift each feature in the signal

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

γ and β are learnable via backpropagation

BATCHNORM

- Nominally:
  - During training E & Var are computed on the current batch
  - Running averages of E & Var are tracked during training, e.g. $E_{avg} \leftarrow 0.9E_{avg}+0.1E_{batch}$
  - At inference time E & Var are the tracked averages
- Running batchnorm (code example):
  - The running average of sums and sum-of-squares of features are tracked
  - E & Var computed on the fly from tracked averages
  - Same transformation applied during training & inference
  - Helps with stability & generalisation on sparse data (e.g. muir regression)

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

γ and β are learnable via backpropagation

CNNS IN PYTORCH

```
nn.Conv3d(in_channels=in_c, out_channels=out_c, kernel_size=kernel_sz, padding=padding, stride=stride, bias=bias)
```

- **in_channels**: Number of channels (features) to expect in the incoming data
- **out_channels**: Number of features to produce: Every kernel produces one channel; this sets the number of different kernels to apply to the incoming data
- **kernel_size**: Can either be an integer (kernel same size in every dimension), or a tuple of integers (size in x,y,z)
- **padding**: Number of zero-padded elements to include. Either integer for same number in each dimension, or specify per dimension as a tuple. kernel_size//2 results in no loss of elements
- **stride**: Stride in (x,y,z), or single number for same stride in every dimension
- **bias**: Whether to include a bias term. Generally not used when BatchNorm is present

CNNS FOR MUON REGRESSION

CONCEPTUAL IDEA

- Data is:
  - Single channel: recorded energy
  - 3D inherent structure
  - High multiplicity (50x32x32 = 51,200 elements total)
- Use 3D CNN to downsample and optimally process raw data
  - Output of CNN is reshaped into a flat vector of features
- Fully connected layers applied to CNN output to regress to muon energy
  - Single output: predicted energy

BASELINE CNN

- 3D grid downsampled 4 times by padded, stride-2 kernels
  - (50,32,32) → (25,16,16) → (13,8,8) → (7,4,4) → (4,2,2)
- Compensate for downsampling by increasing number channels in data
  - 1→ 8 in first downsample
  - channel_coef * n_channels on subsequent downsamples
- CNN output reshaped into a flat vector with 16*channels elements
  - In this case: 27 channels = 432 elements

```
cnn3d(
  (conv_layers): Sequential(
    (0): Sequential(
      (0): Conv3d(1, 8, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
    (1): Sequential(
      (0): Conv3d(8, 12, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
    (2): Sequential(
      (0): Conv3d(12, 18, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
    (3): Sequential(
      (0): Conv3d(18, 27, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
  )
  (fc_layers): Sequential(
    (0): Sequential(
      (0): Linear(in_features=432, out_features=216, bias=True)
      (1): ReLU()
    )
    (1): Sequential(
      (0): Linear(in_features=216, out_features=108, bias=True)
      (1): ReLU()
    )
    (2): Sequential(
      (0): Linear(in_features=108, out_features=1, bias=True)
    )
  )
)
```

DEEPER CNN

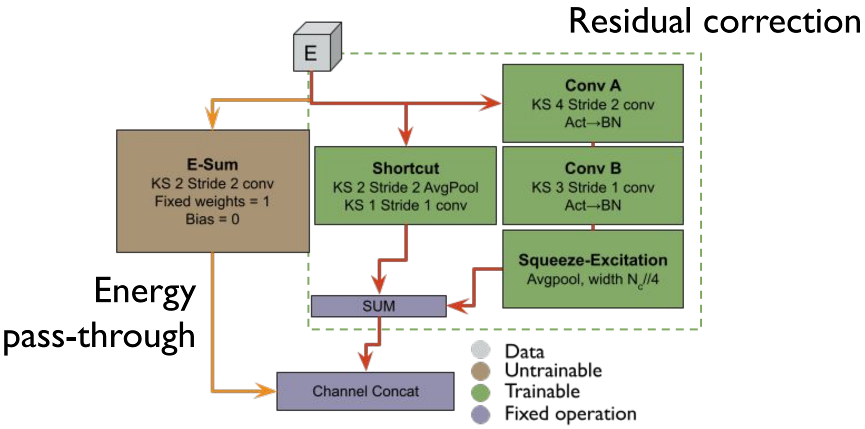
- Can use n_layers_per_res to add additional stride-1 layers in between each stride-2 layer
    - Allows use to build deeper networks whilst retaining the same output size

```
cnn3d(
  (conv_layers): Sequential(
    (0): Sequential(
      (0): Conv3d(1, 8, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
    (1): Sequential(
      (0): Conv3d(8, 8, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
    (2): Sequential(
      (0): Conv3d(8, 12, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
    (3): Sequential(
      (0): Conv3d(12, 12, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
    (4): Sequential(
      (0): Conv3d(12, 18, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
    (5): Sequential(
      (0): Conv3d(18, 18, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
    (6): Sequential(
      (0): Conv3d(18, 27, kernel_size=(3, 3, 3), stride=(2, 2, 2), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
    (7): Sequential(
      (0): Conv3d(27, 27, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1), bias=False)
      (1): ReLU()
    )
  )
  (fc_layers): Sequential(
    (0): Sequential(
      (0): Linear(in_features=432, out_features=216, bias=True)
      (1): ReLU()
    )
    (1): Sequential(
      (0): Linear(in_features=216, out_features=108, bias=True)
      (1): ReLU()
    )
    (2): Sequential(
      (0): Linear(in_features=108, out_features=1, bias=True)
    )
  )
)
```

PAPER ARCHITECTURE

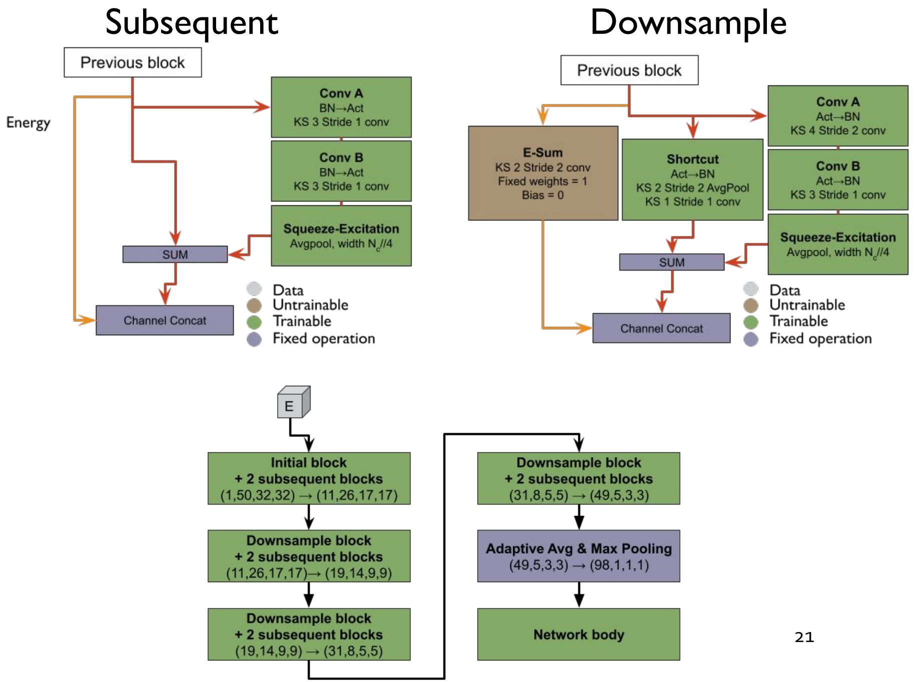
PAPER ARCHITECTURE - CNN BLOCK

- Custom 3D CNN arch aims to learn small corrections to reco. Energy
  - Reco. energy summed up by 2x2 filter
  - Correction learnt by residual convolutional layers
  - Summed reco. energy is concatenated to output, so always available to later layers
- Running BatchNorm, helps with data sparsity
  - Applies running average during training, rather than batchwise stats
- Swish-1 used as activation function
- Squeeze-excitation block further improves performance



PAPER ARCHITECTURE - FULL MODEL

- Can build deeper networks by not downsampling the grid
- Further downsampling uses pre-activation layout
- Full CNN contains 12 blocks, followed by mean and max aggregation
  - 51,200 inputs → 98 features
- CNN head outputs combined with HL features and fed through 3 FC layers



PAPER ARCHITECTURE - ABLATION

- CNN much better than flattening raw hits
- Running BN improves performance and stability
- ResNet layout is useful
- Other additional components provide indications of improvement

Ablation	MI	Change in MI [%]
Default	19.42 ± 0.08	N/A
No BN	18.5 ± 0.3	-5 ± 1
No identity path	18.72 ± 0.08	-3.6 ± 0.6
Nominal BN	19.2 ± 0.2	-1.1 ± 0.9
No E-pass	19.30 ± 0.05	-0.6 ± 0.5
No SE	19.33 ± 0.09	-0.5 ± 0.6
No pooling	19.4 ± 0.1	-0.4 ± 0.7
No CNN	17.45 ± 0.09	-10.2 ± 0.6