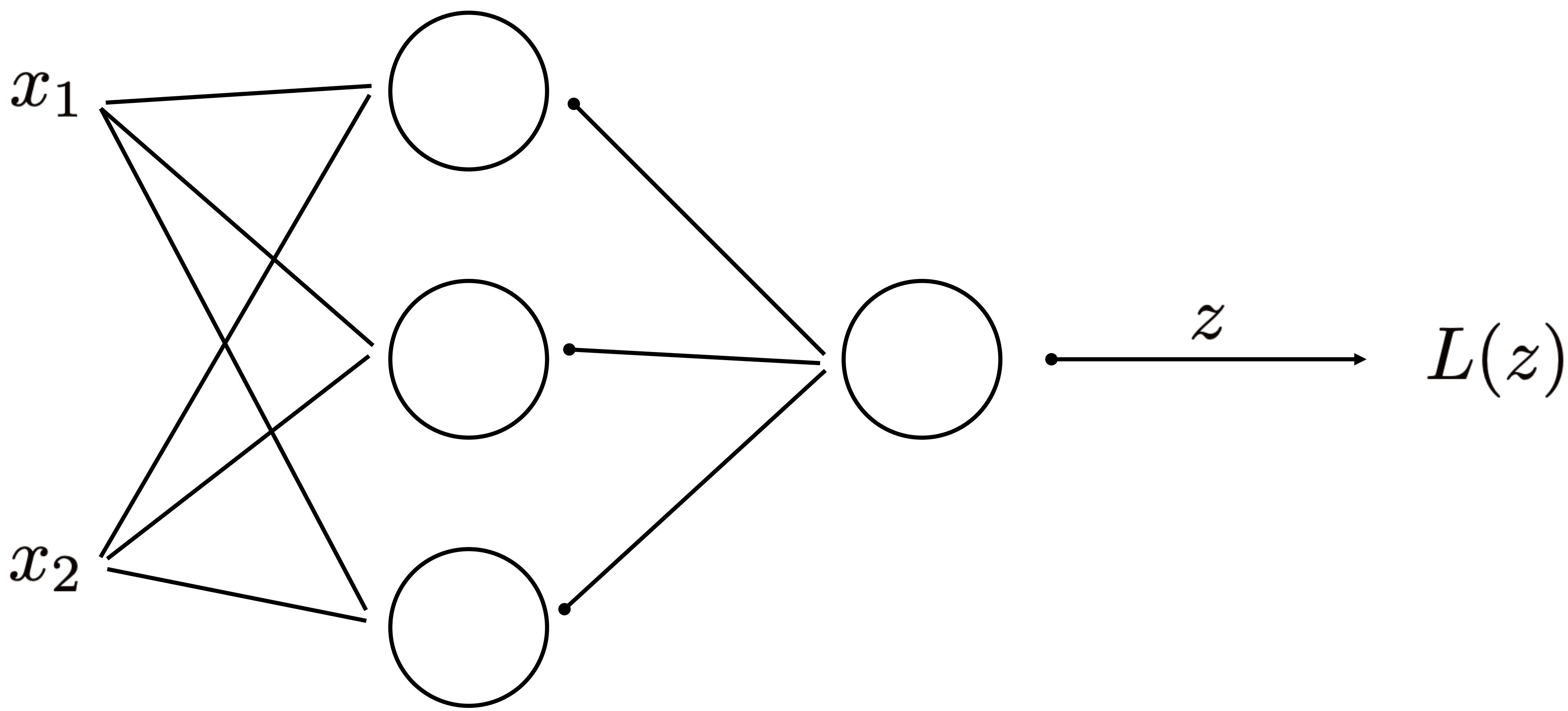
Convolutional Neural Networks and Applications in the NEXT Experiment

G. Díaz, J.A. Hernando, J. Renner
IFGAE / Universidade de Santiago de Compostela

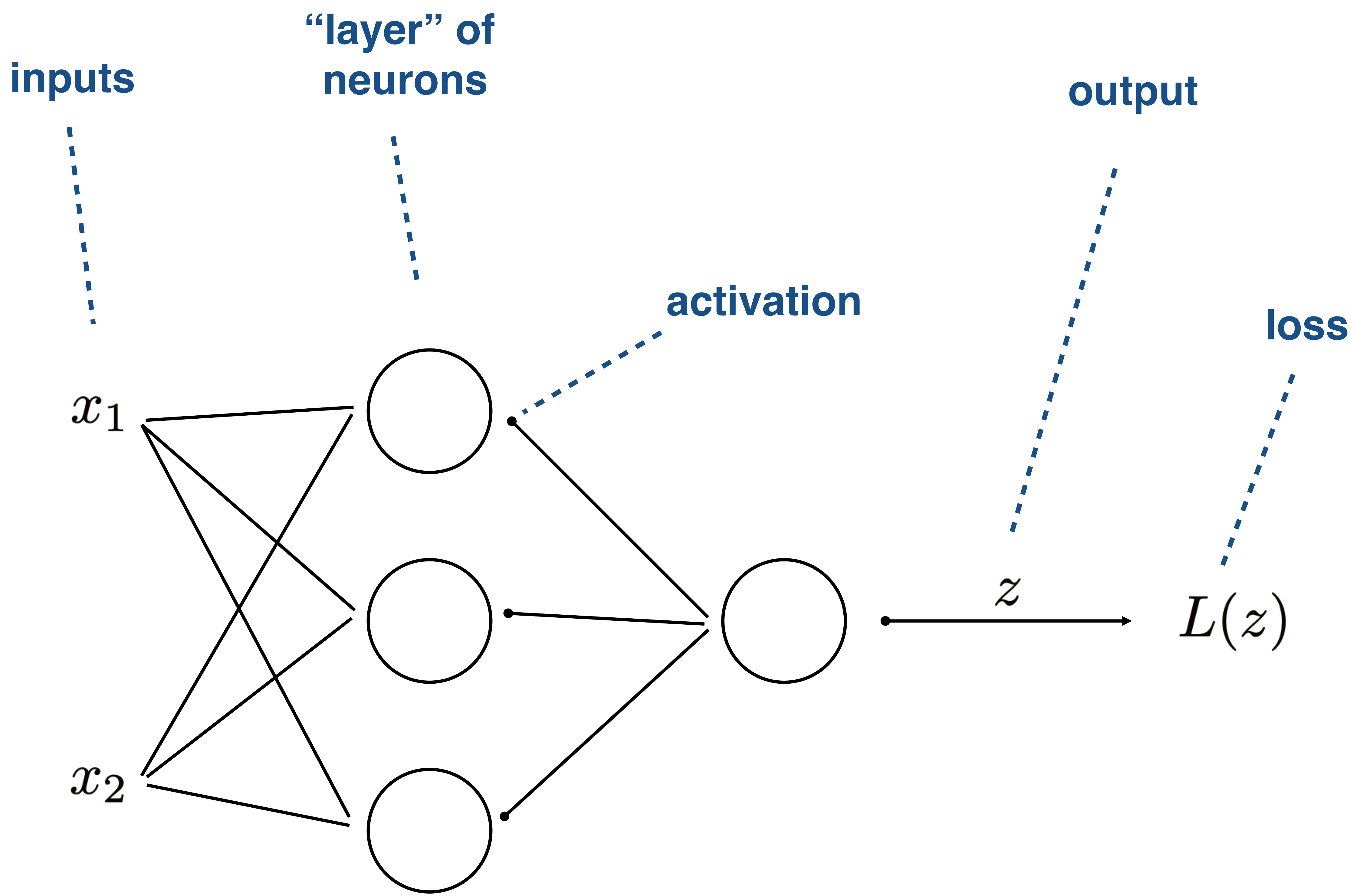
Advanced Computing and Machine Learning
June 1, 2020

Review: last week

Neural networks

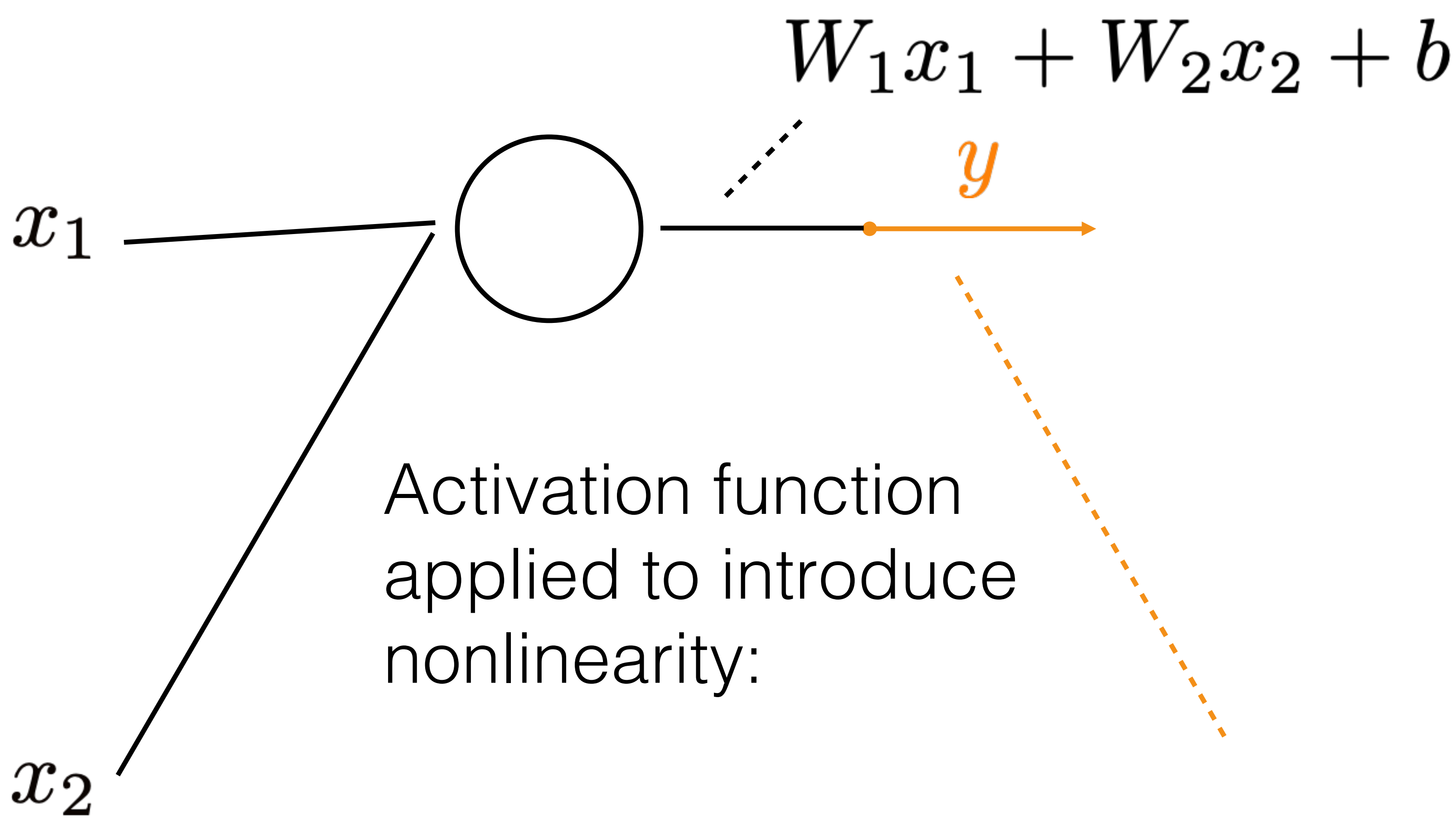


Neural networks



Neural networks

Each neuron contains a **weight** corresponding to each input, and a **bias**:



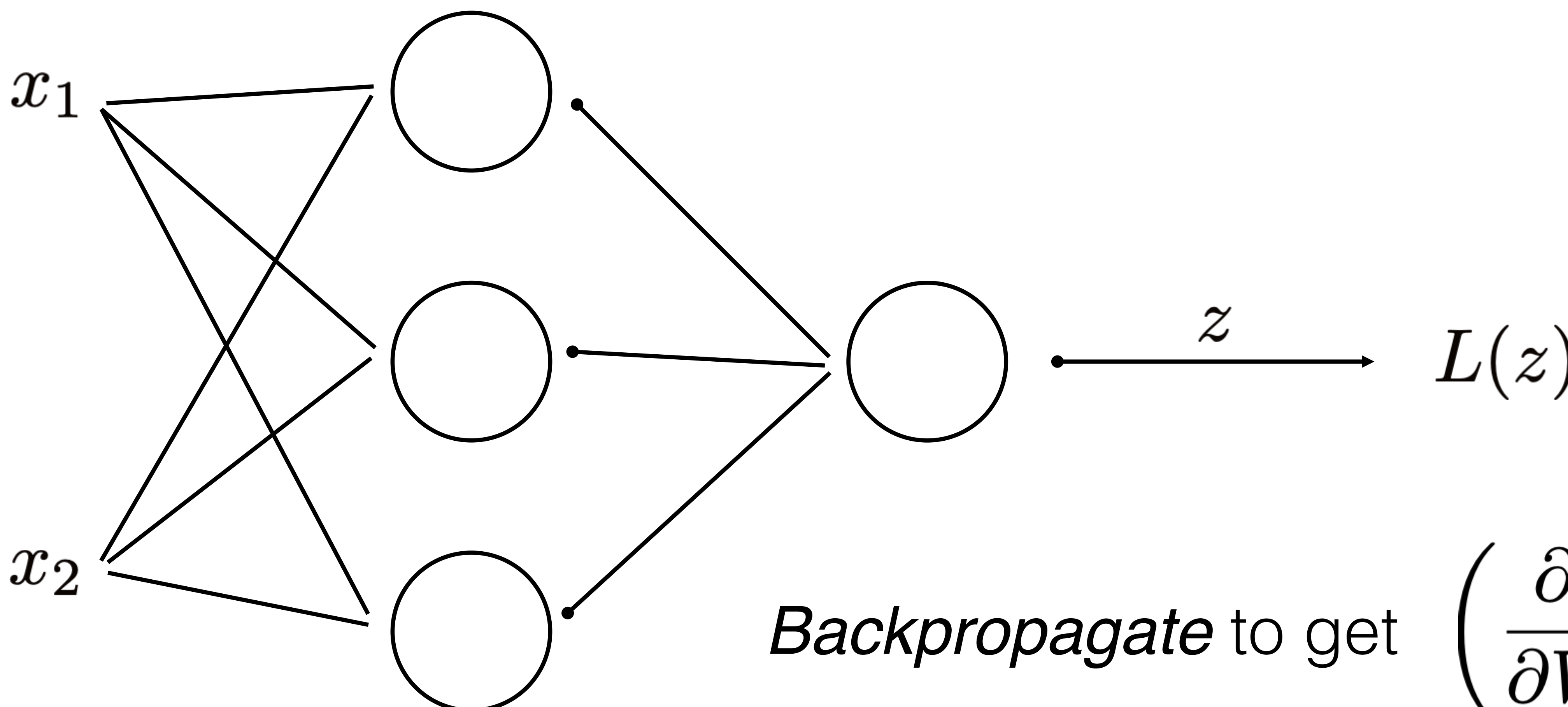
Activation function applied to introduce nonlinearity:

$$y = \sigma(W_1x_1 + W_2x_2 + b) = \frac{1}{1 + e^{-(W_1x_1 + W_2x_2 + b)}}$$

Neural networks

To minimize L, we want to push the entire network (all parameters) in the direction of -gradient(L):

$$W_1 \to W_1 - \left(\frac{\partial L}{\partial W_1}\right)\epsilon$$

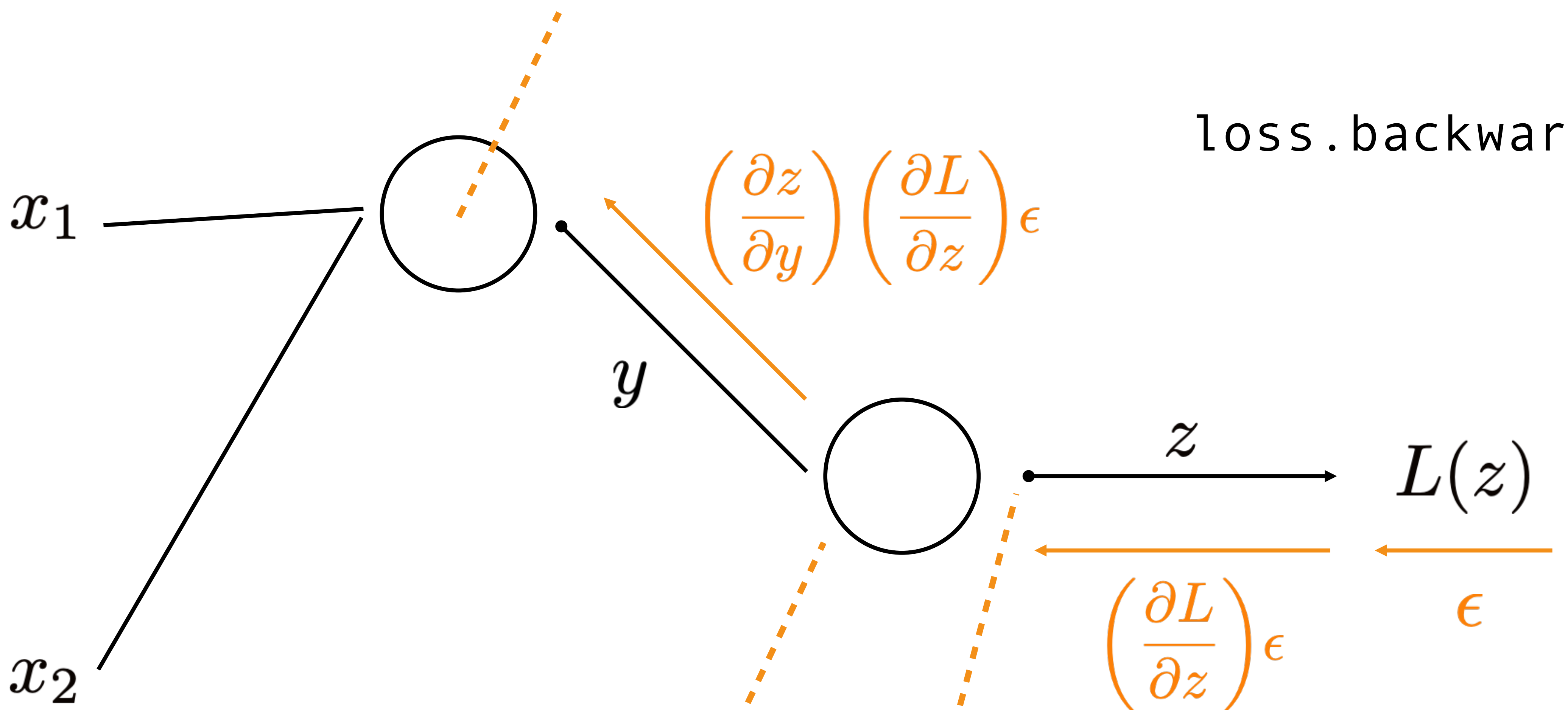


Backpropagate to get $\left(\frac{\partial L}{\partial W_1}\right)$

Neural networks

$$\left(\frac{\partial L}{\partial W_1}\right)\epsilon = \left(\frac{\partial y}{\partial W_1}\right)\left(\frac{\partial z}{\partial y}\right)\left(\frac{\partial L}{\partial z}\right)\epsilon$$

```
loss.backward()
```

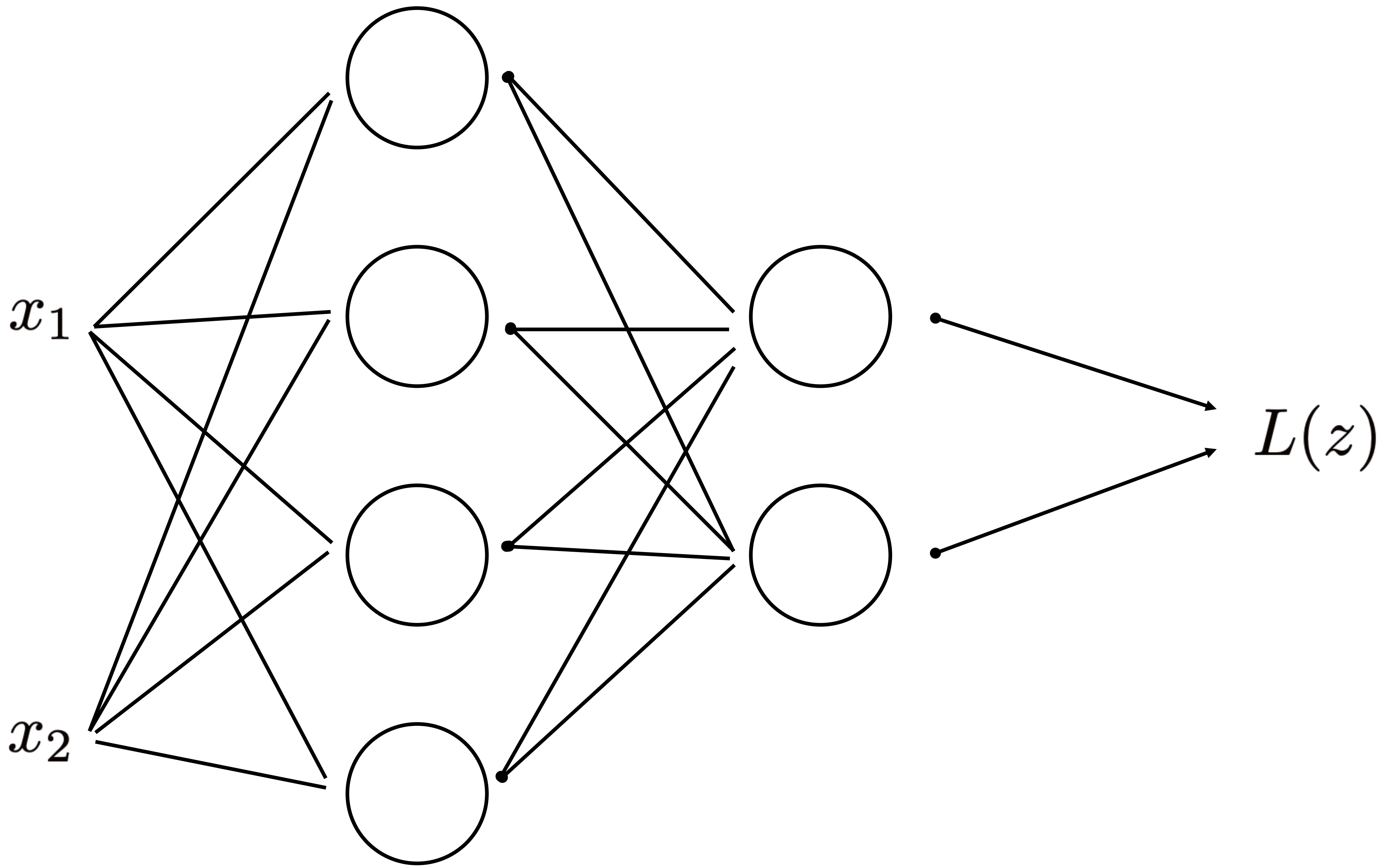


Note: ∂z/∂y will have to account for the functional forms of: **this neuron**

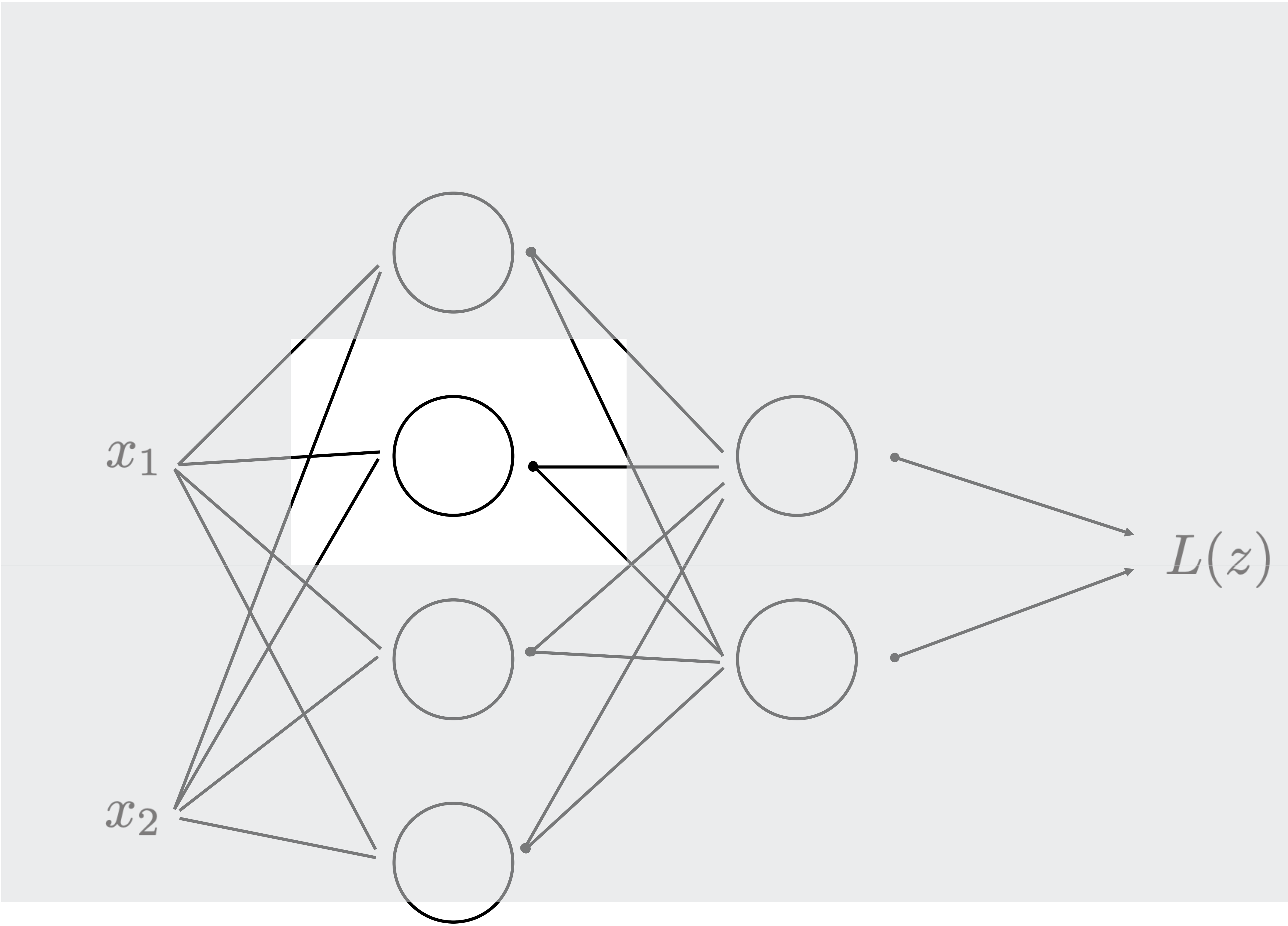
and this activation

Neural networks

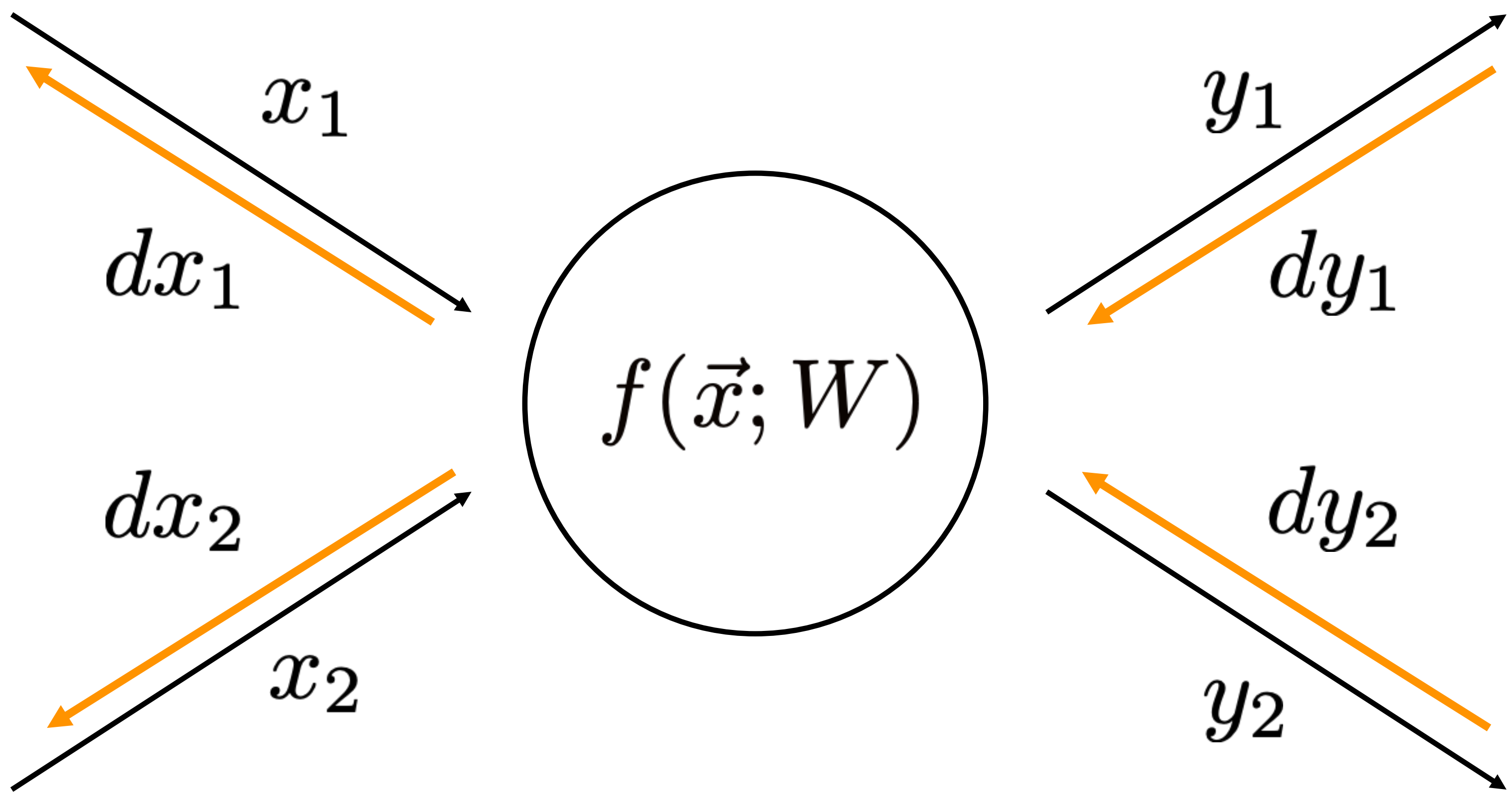
- Note that backpropagation on graphs can be performed on a node-by-node basis:



Neural networks



Neural networks

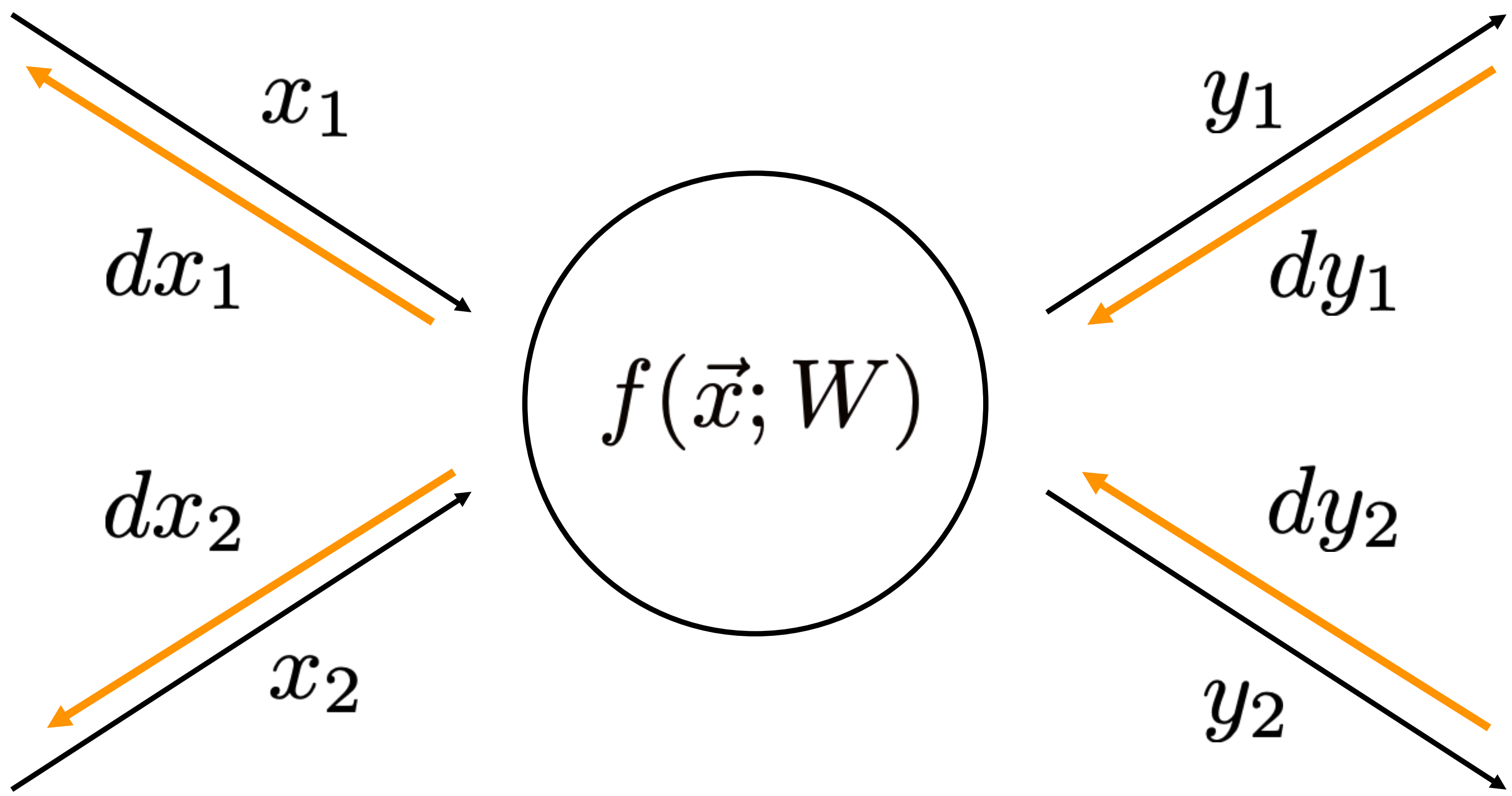


- Given gradients dy_1 and dy_2
- Given f differentiable
- Compute gradients dW for parameter updates, and gradients dx_1 and dx_2 to continue backpropagation

$$dx_i = \sum_k \frac{\partial y_k}{\partial x_i} dy_k$$

Libraries such as PyTorch handle the mechanics of this automatically

Neural networks



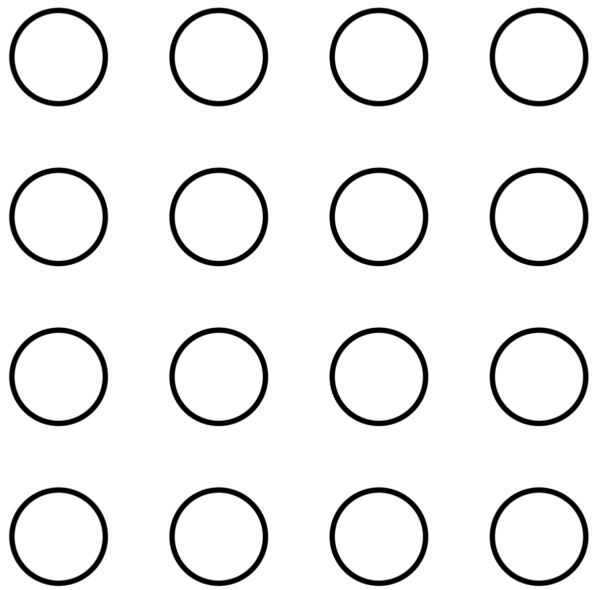
- So far, we've considered:

$$f(\vec{x}, W) = W\vec{x} + \vec{b}$$

Let's try something else!

Convolutional Neural Networks (CNNs)

CNNs



Inputs, arranged in

a 2D matrix

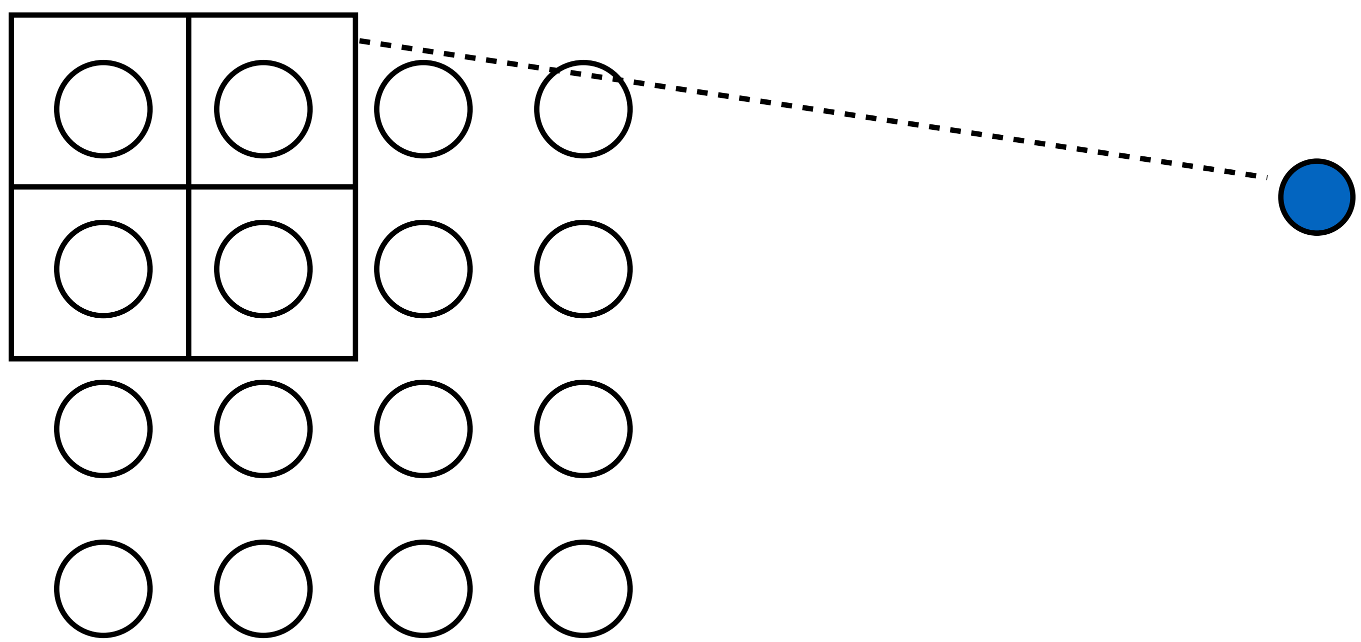
$$\begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix}$$

KxL kernel of weights
(in this case 2x2)

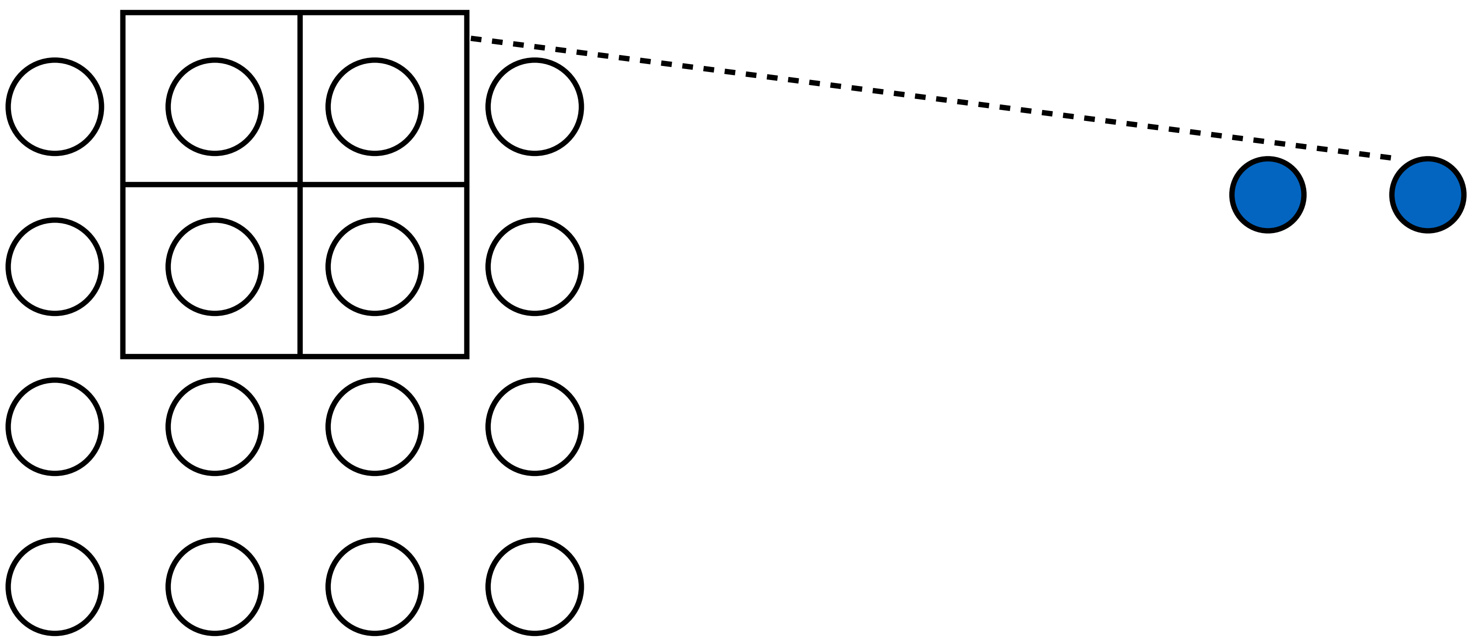
CNNs

Multiply and add:

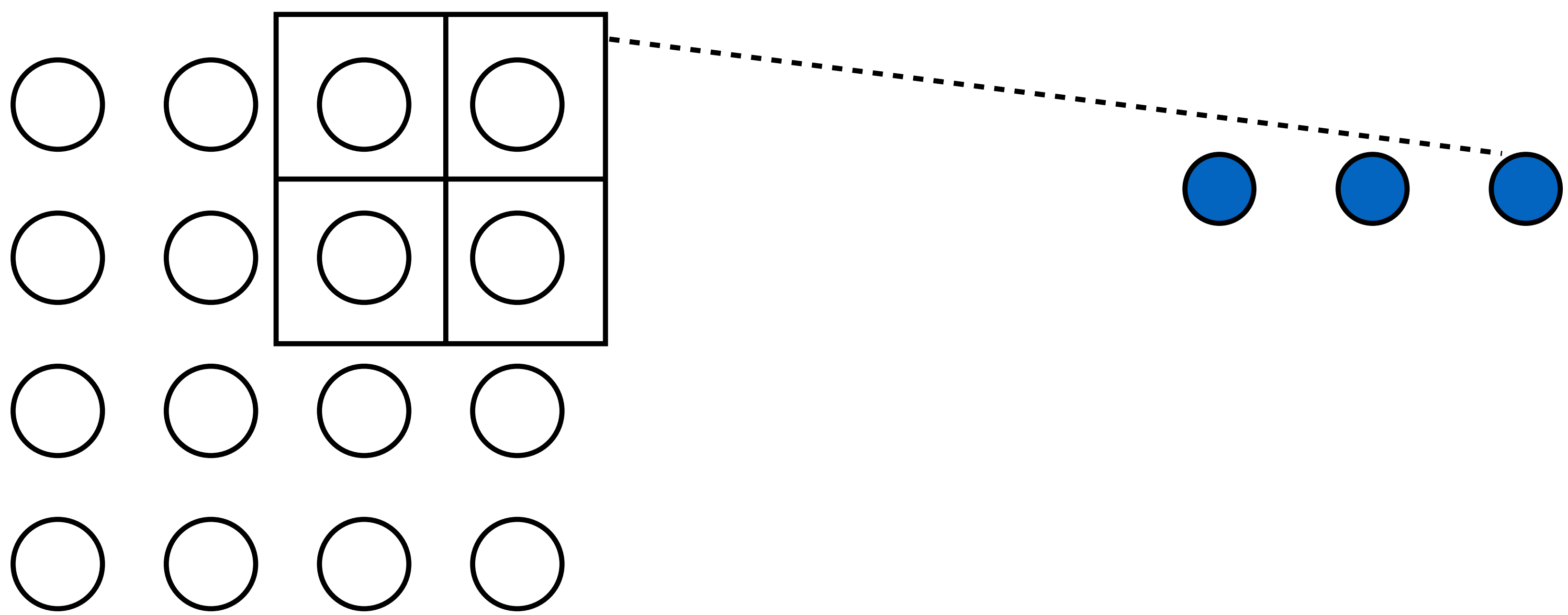
$$z_{11} = W_{11}x_{11} + W_{12}x_{12} + W_{21}x_{21} + W_{22}x_{22}$$



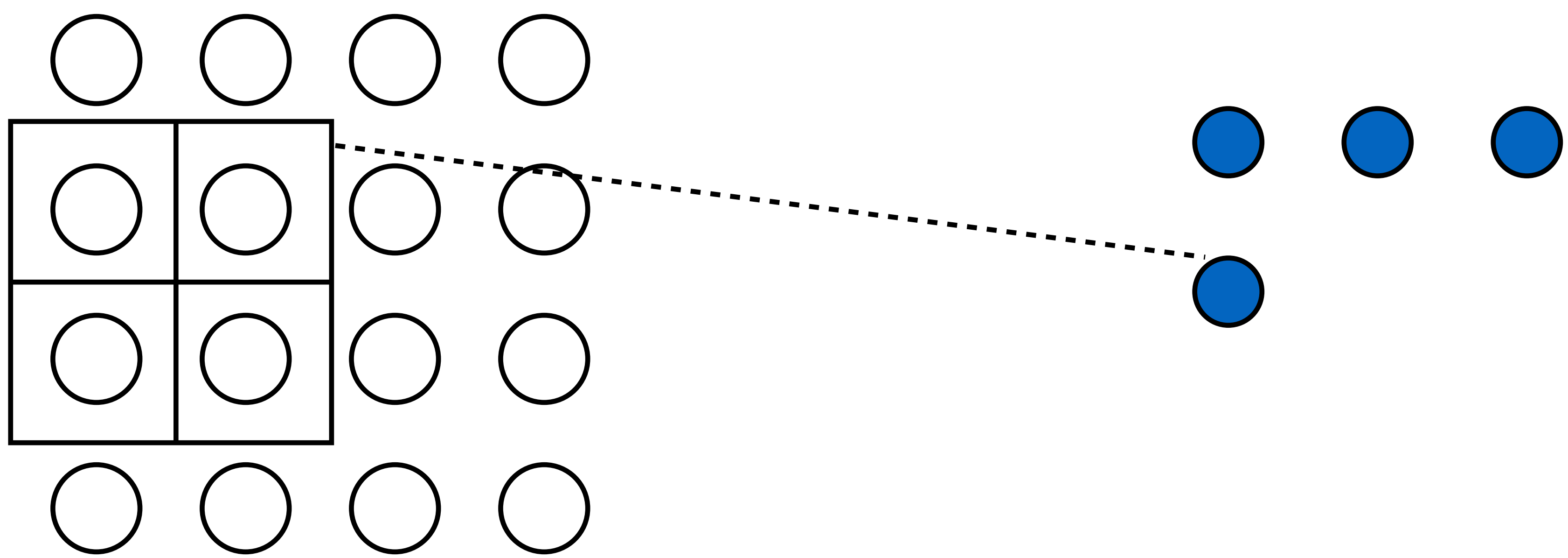
CNNs



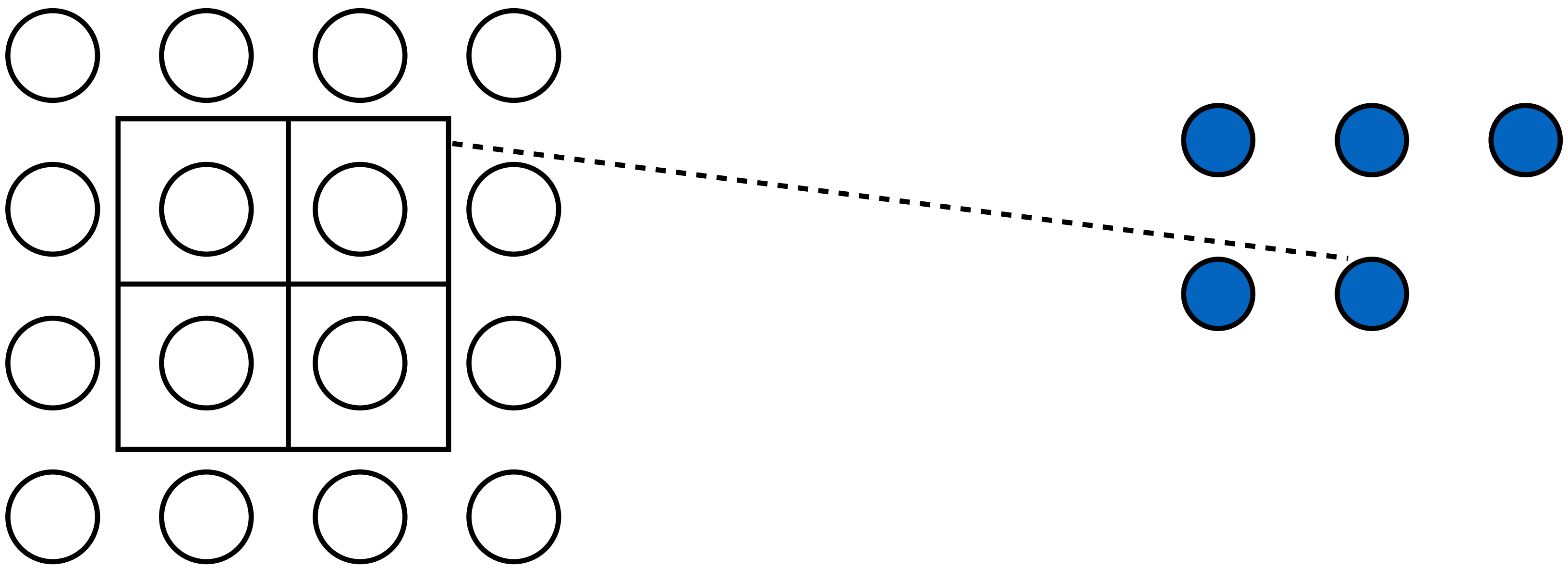
CNNs



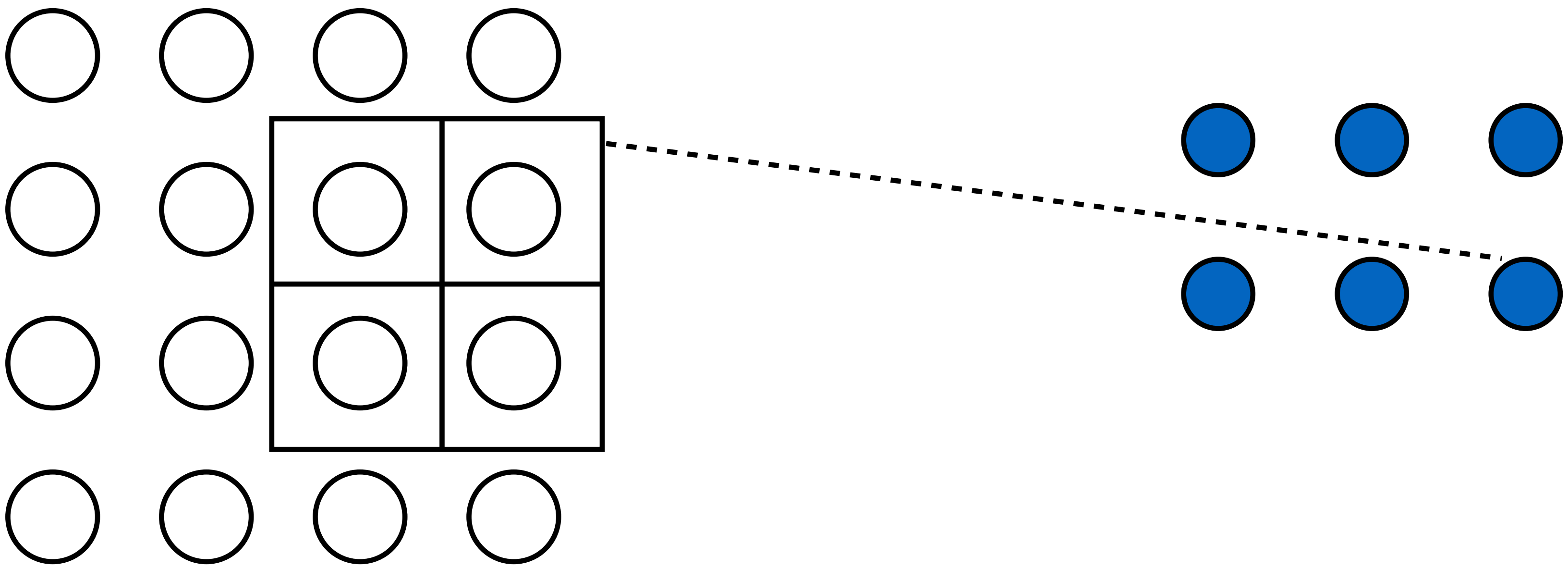
CNNs



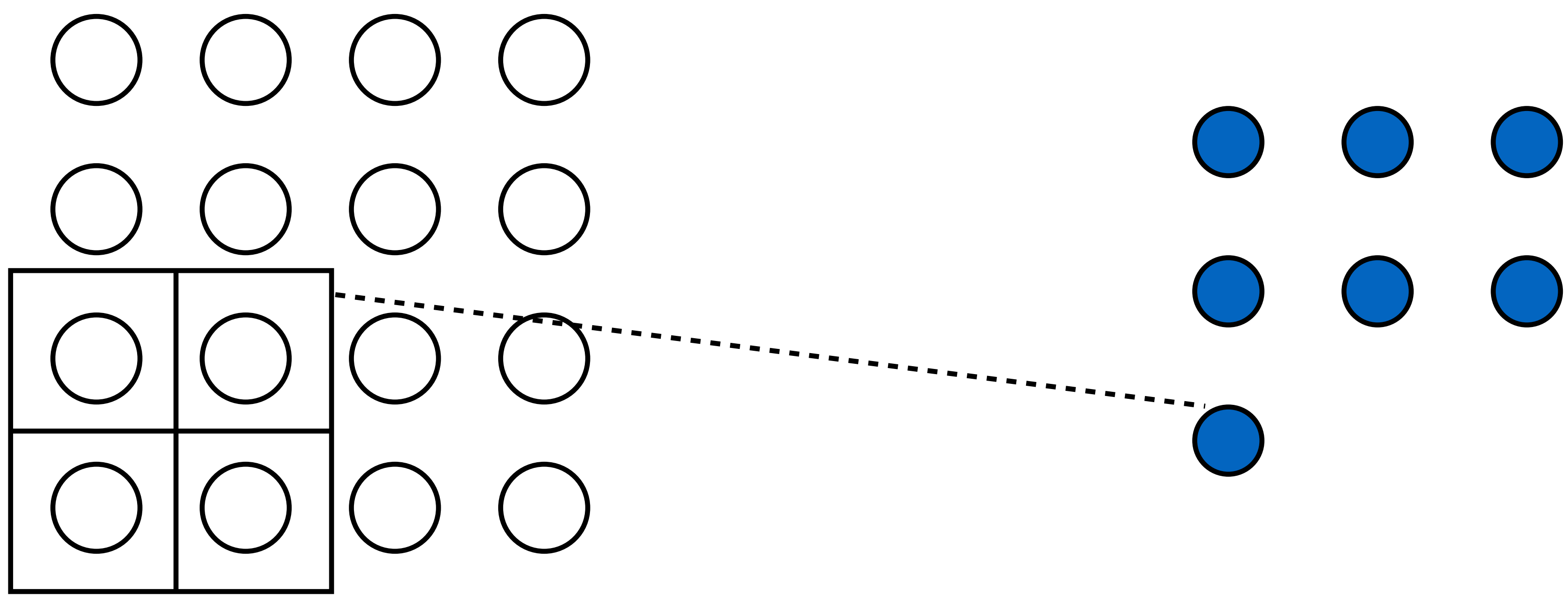
CNNs



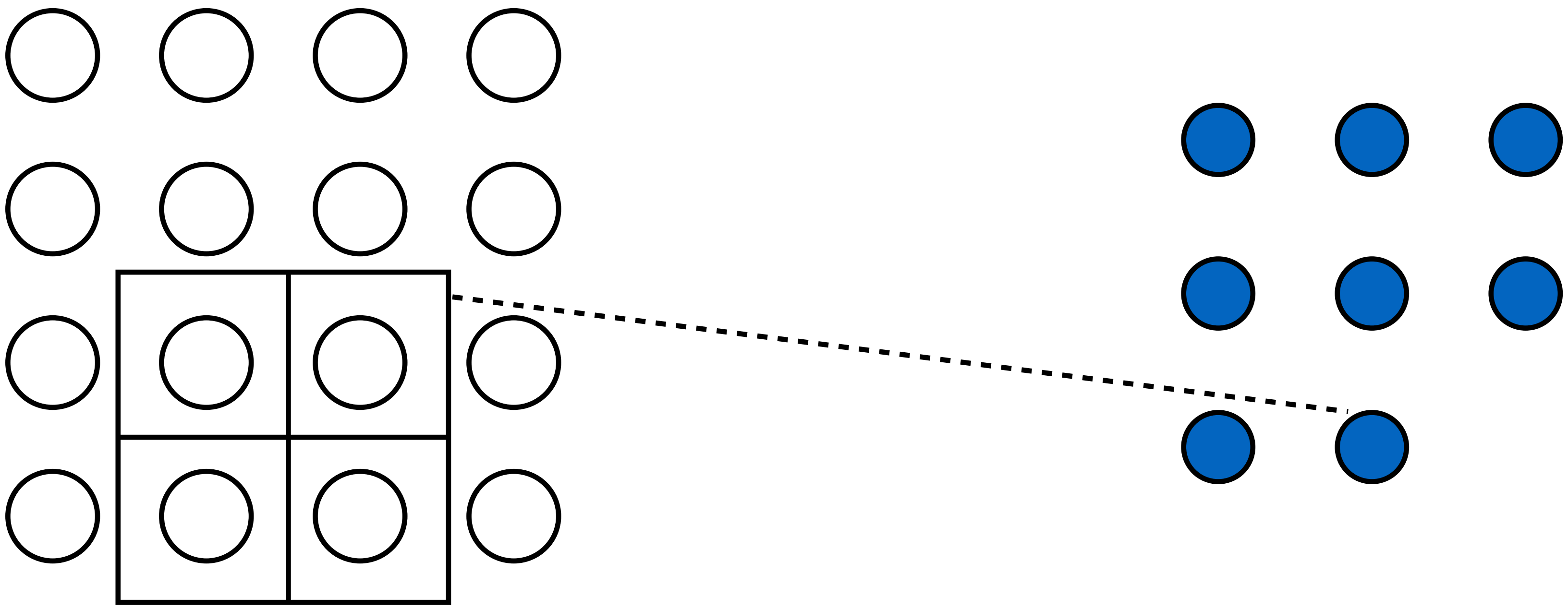
CNNs



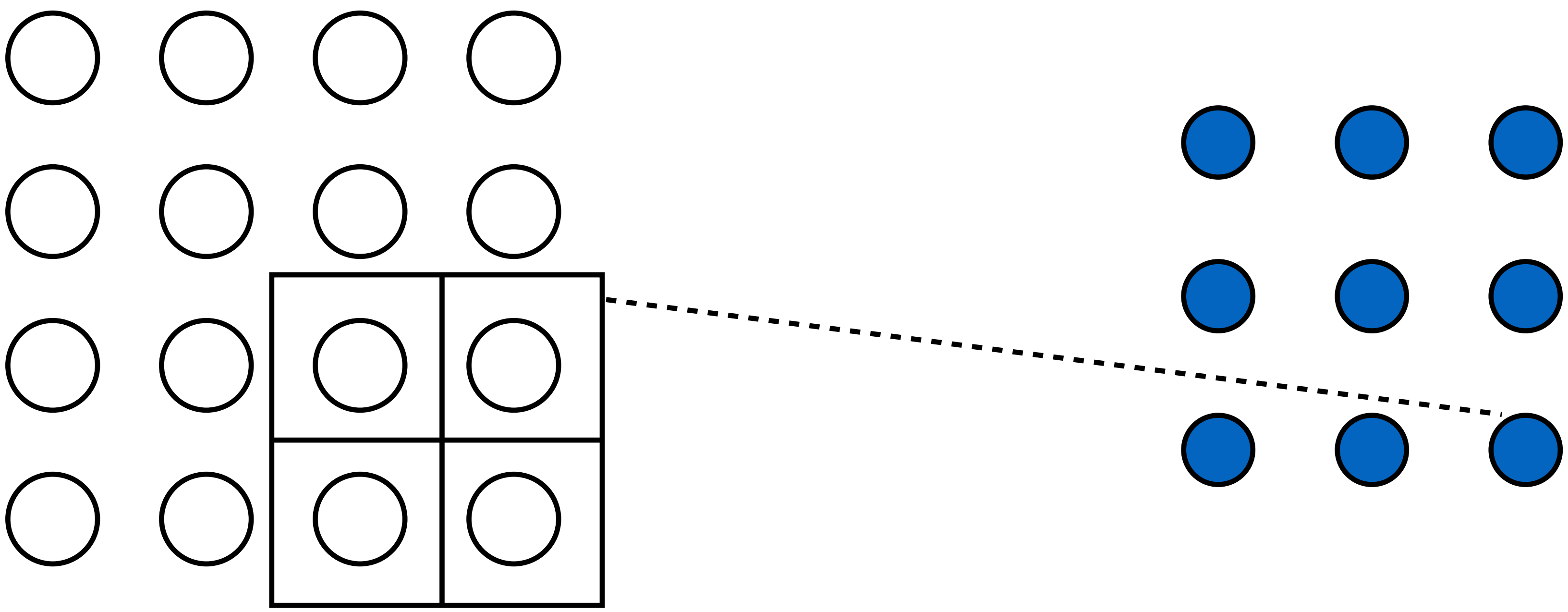
CNNs



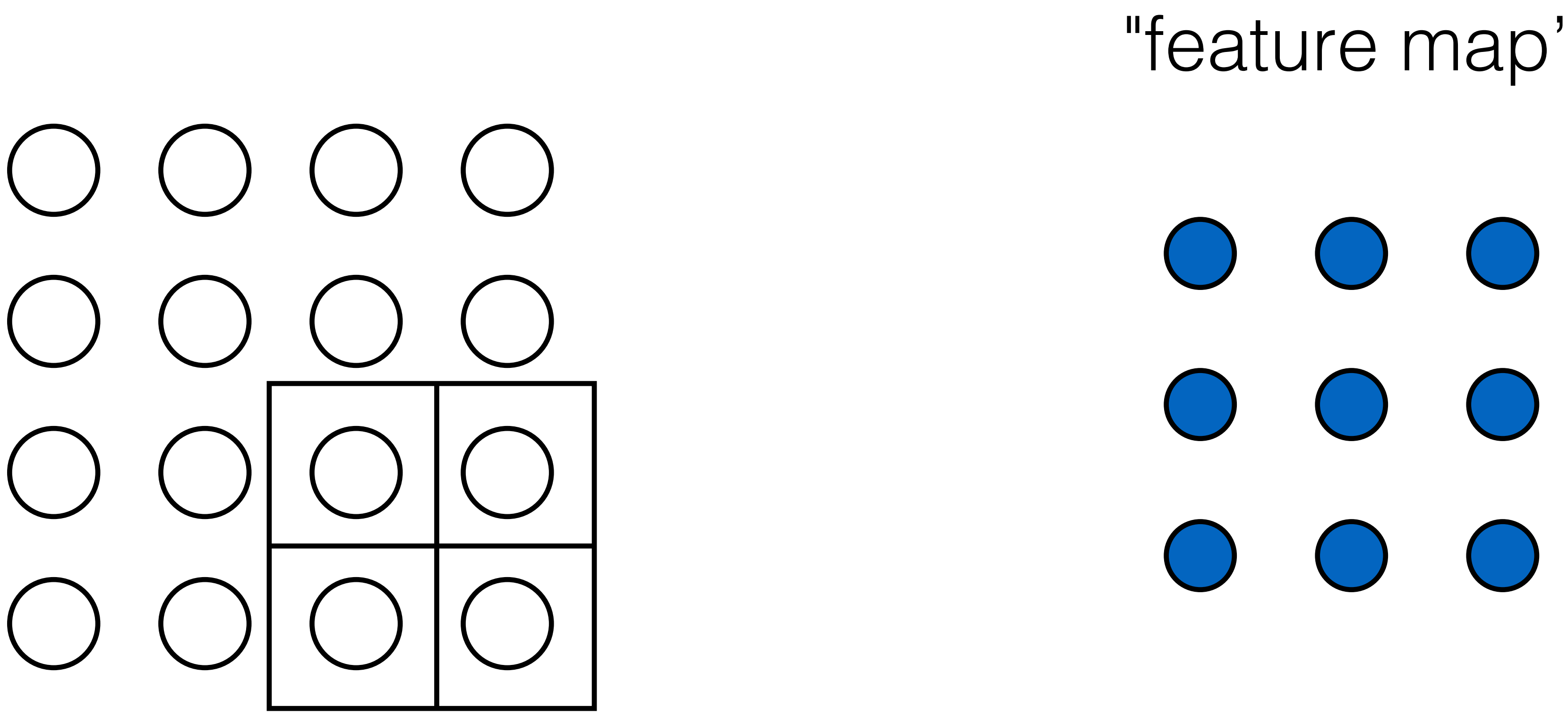
CNNs



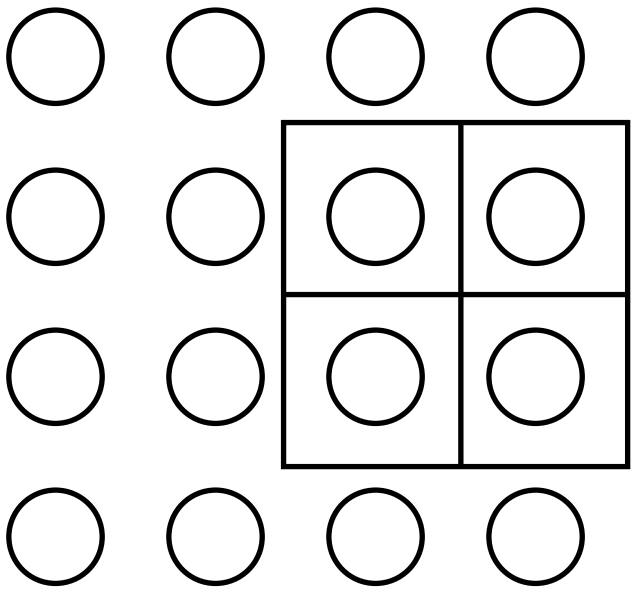
CNNs



CNNs



CNNs



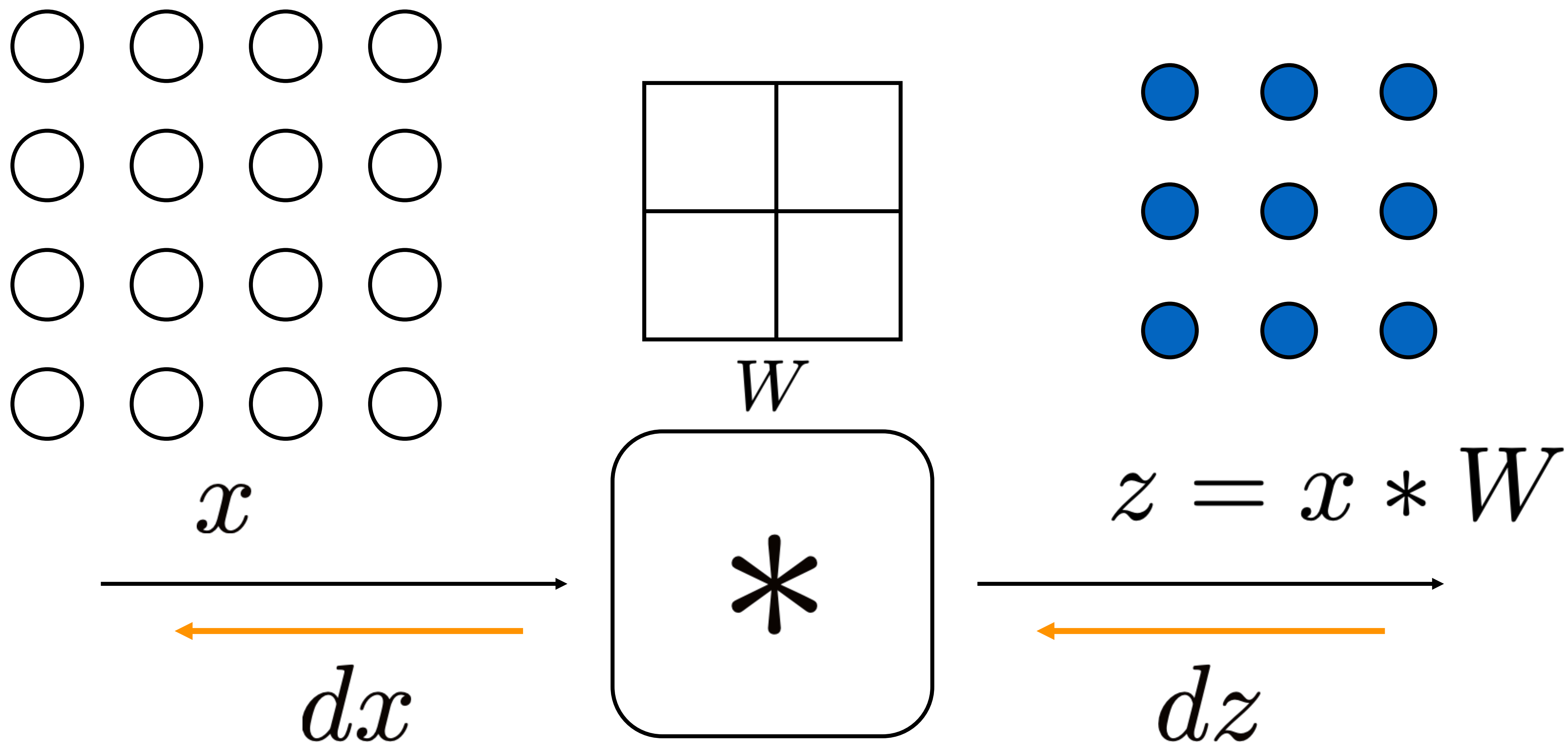
W_{11}	W_{12}
W_{21}	W_{22}

Note that we used the same weight matrix for the entire operation:

The idea that nearby input values are related is **built-in** to CNNs via "weight tying".

CNNs

How do you “take the derivative” of that?



$$z_{ij} = \sum_{kl} W_{kl} x_{i+k,j+l}$$

CNNs

$$z_{ij} = \sum_{kl} W_{kl} x_{i+k,j+l}$$

Note: $$\frac{\partial W_{ij}}{\partial W_{mn}} = \delta_{im}\delta_{jn}$$

(weights are independent variables)

Let's find the values used to update weights W_{ij}:

$$\begin{aligned}
dW_{mn} = \left(\frac{\partial L}{\partial W_{mn}}\right)\epsilon &= \sum_{ij} dz_{ij} \frac{\partial z_{ij}}{\partial W_{nm}} \\
&= \sum_{ij} dz_{ij} \frac{\partial W_{kl}}{\partial W_{nm}} x_{i+k,j+l} \\
&= \sum_{ij} dz_{ij} \delta_{kn}\delta_{lm} x_{i+k,j+l} \\
&= \sum_{ij} dz_{ij} x_{i+n,j+m} \\
&= \sum_{ij} dz_{ij} x_{n+i,m+j} \quad \boxed{= (x * dz)_{nm}}
\end{aligned}$$

CNNs

$$z_{ij} = \sum_{kl} W_{kl} x_{i+k,j+l}$$

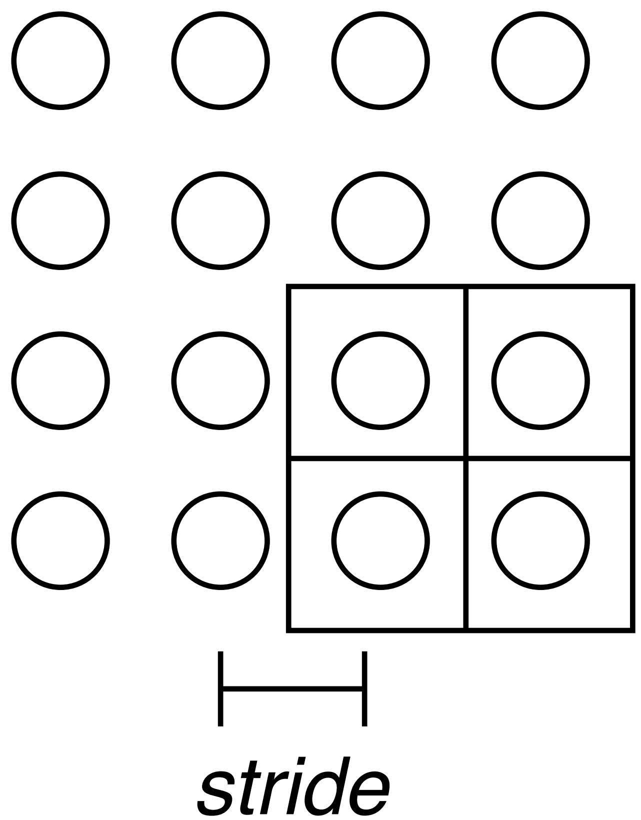
Let's find the values used to update weights W_{ij}:

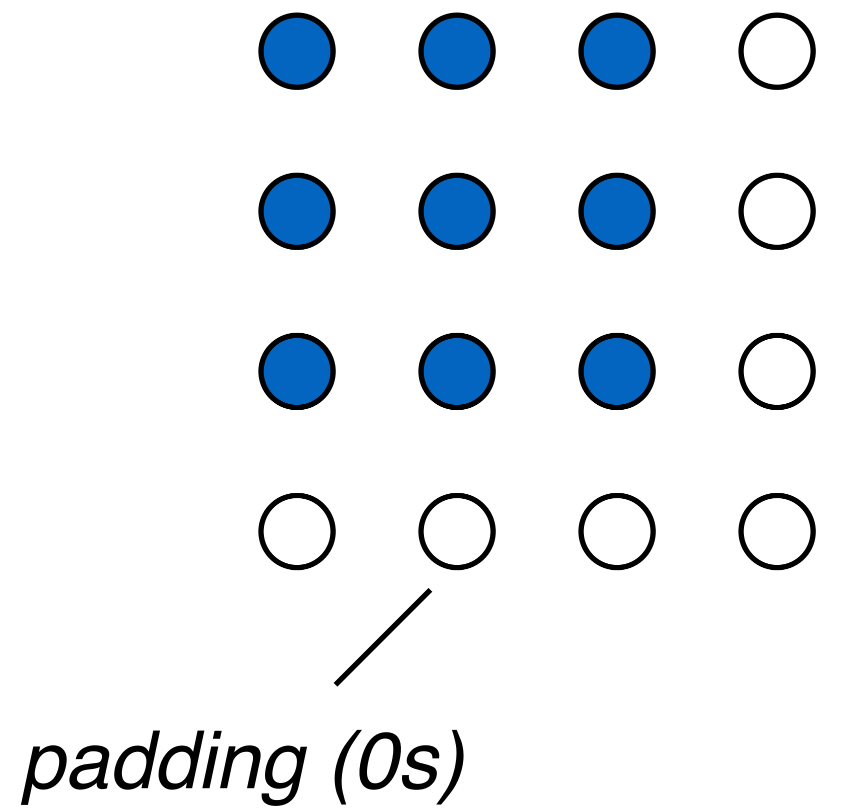
$$\begin{aligned} dW_{mn} = \left(\frac{\partial L}{\partial W_{mn}}\right)\epsilon &= \sum_{ij} dz_{ij} \frac{\partial z_{ij}}{\partial W_{nm}} \\ &= \sum_{ij} dz_{ij} \frac{\partial W_{kl}}{\partial W_{nm}} x_{i+k,j+l} \\ &= \sum_{ij} dz_{ij} \delta_{kn} \delta_{lm} x_{i+k,j+l} \\ &= \sum_{ij} dz_{ij} x_{i+n,j+m} \\ &= \sum_{ij} dz_{ij} x_{n+i,m+j} \quad \boxed{= (x * dz)_{nm}} \end{aligned}$$

The gradients for the weights in a convolution can be found by convolving the output gradients with the input.

CNNs

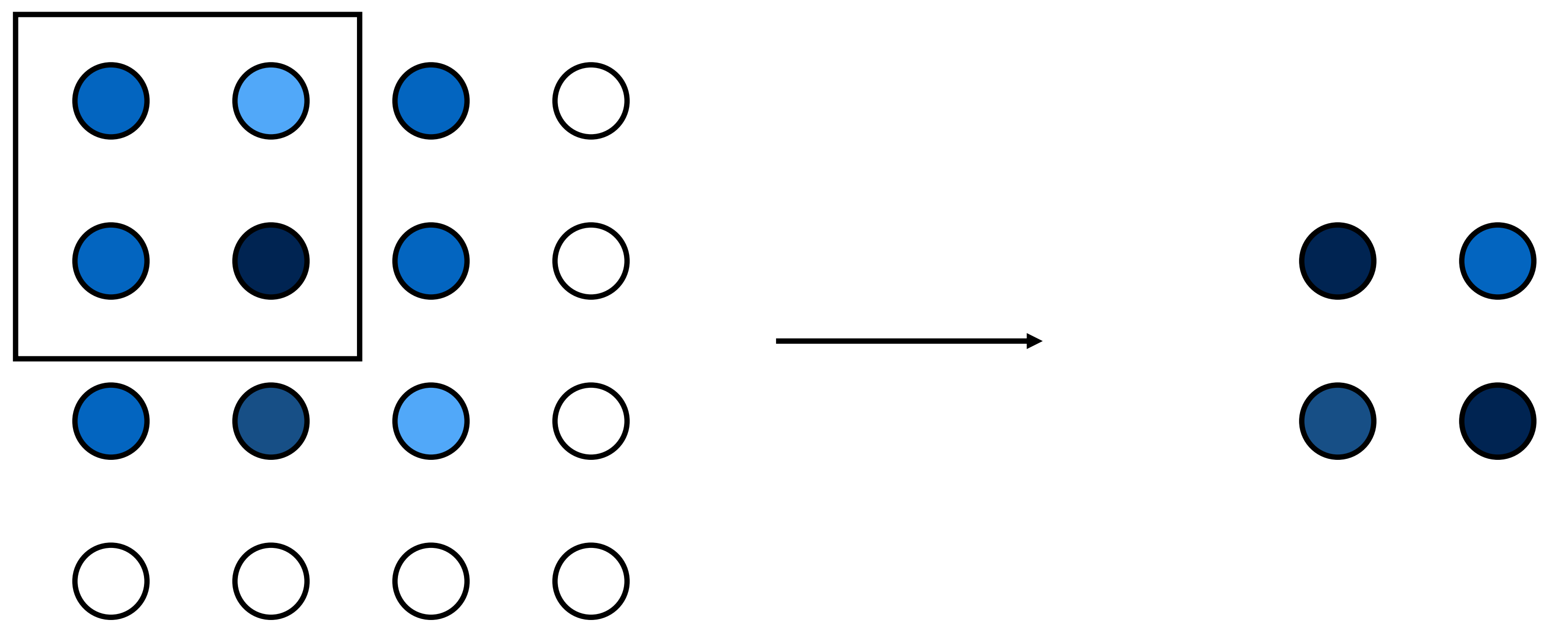
Some additional terminology:





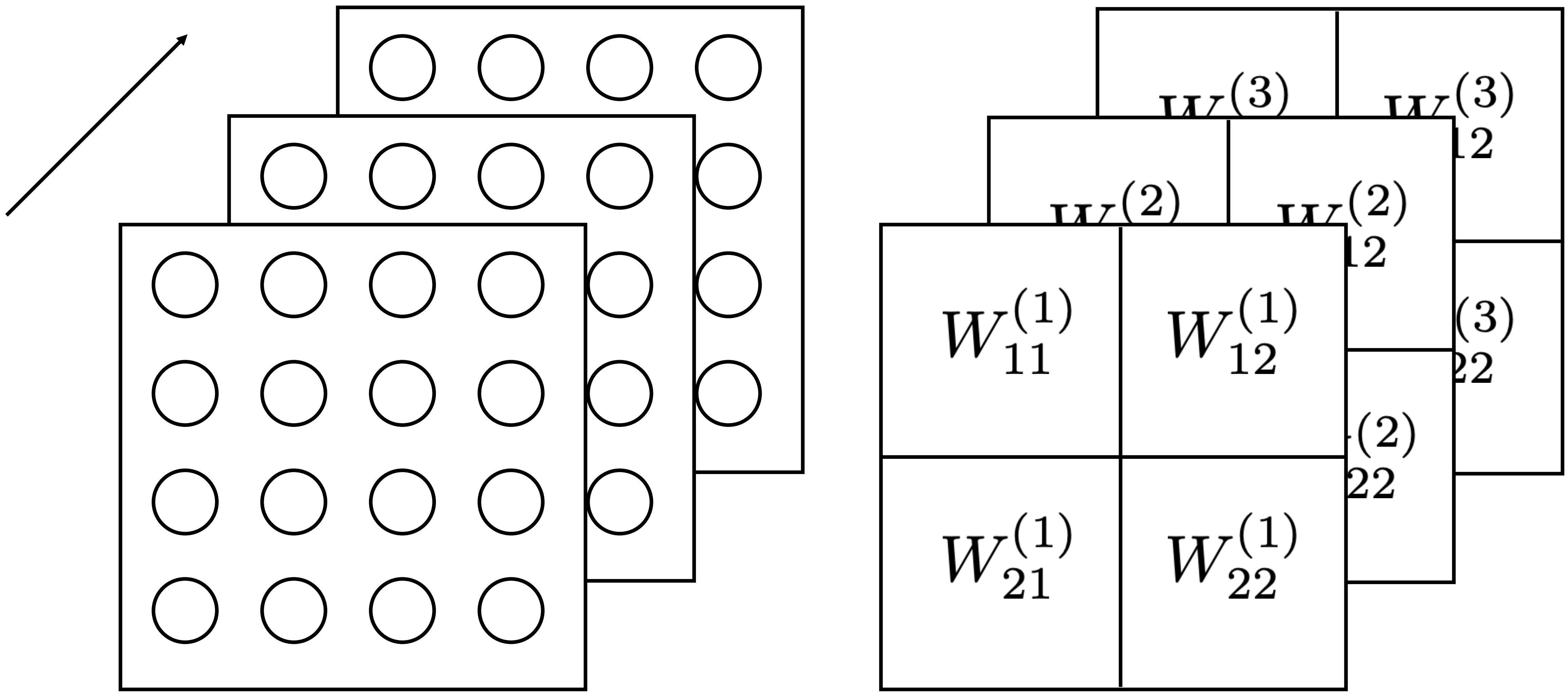
CNNs

(max/average) pooling



Selects only the maximum value, or averages all values over a KxL interval

CNNs

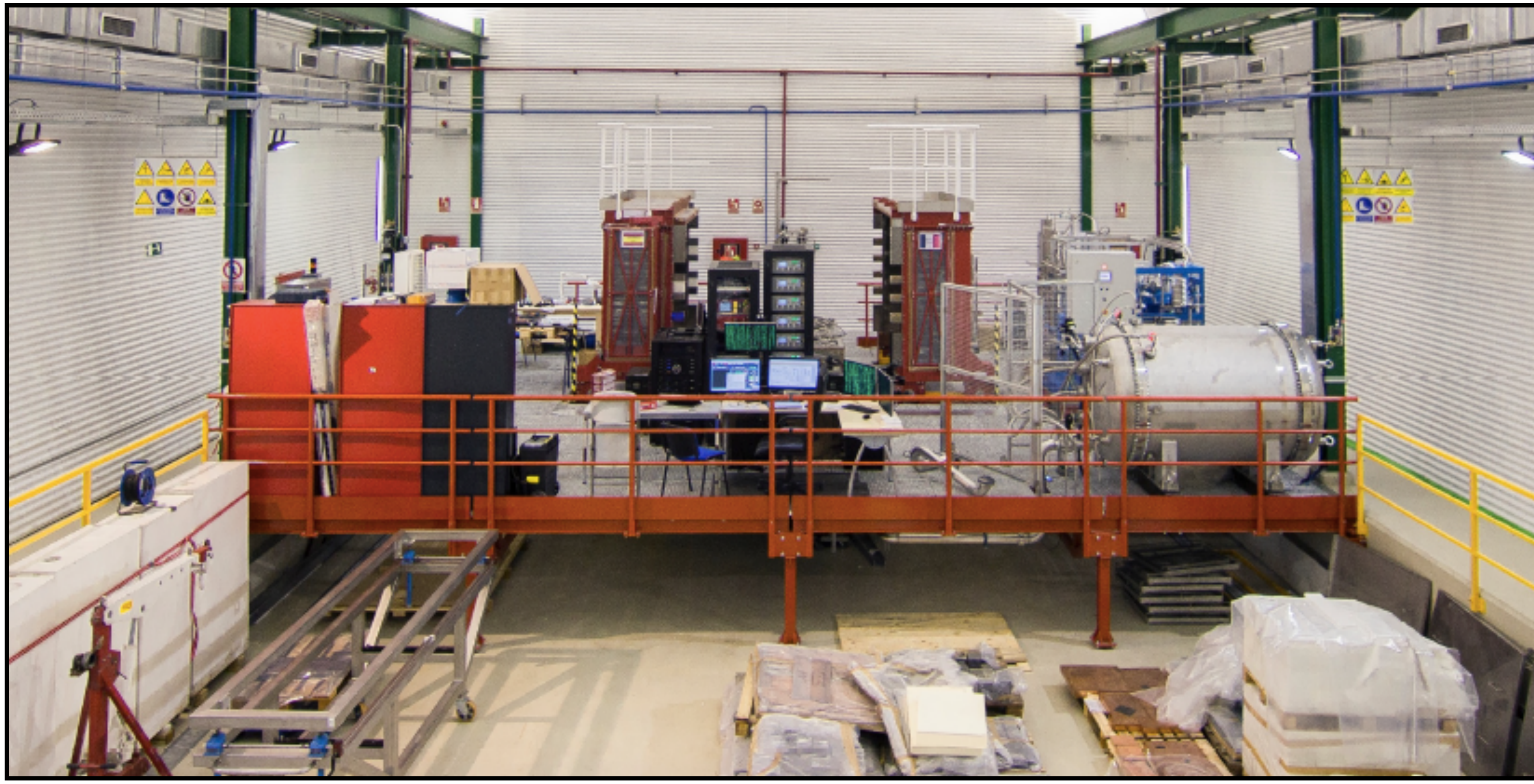


Inputs and kernels can also have multiple *channels* (for example, 3 channels in an RGB image)

NEXT

Neutrino
Experiment with a
Xenon
TPC

LSC
Laboratorio Subterráneo de Canfranc

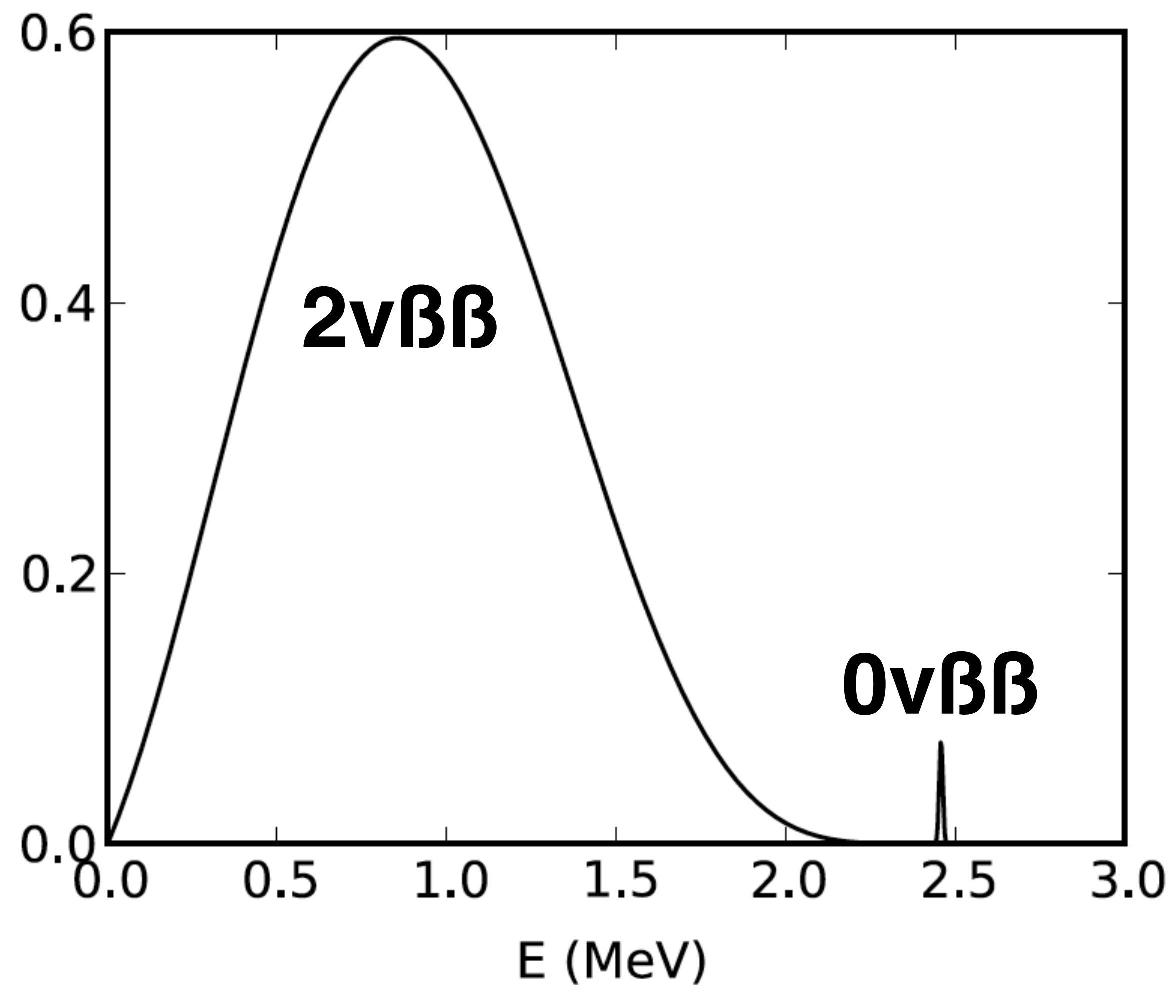


NEXT at Canfranc

- search for neutrinoless double-beta decay ($0\nu\beta\beta$)
- to be commissioned in 2020: 100 kg Xe, enriched to ^{136}Xe (90%)
- high pressure gas, electroluminescent TPC: capable of measuring **energy** of an interaction and reconstructing particle **tracks**

What are we looking for?

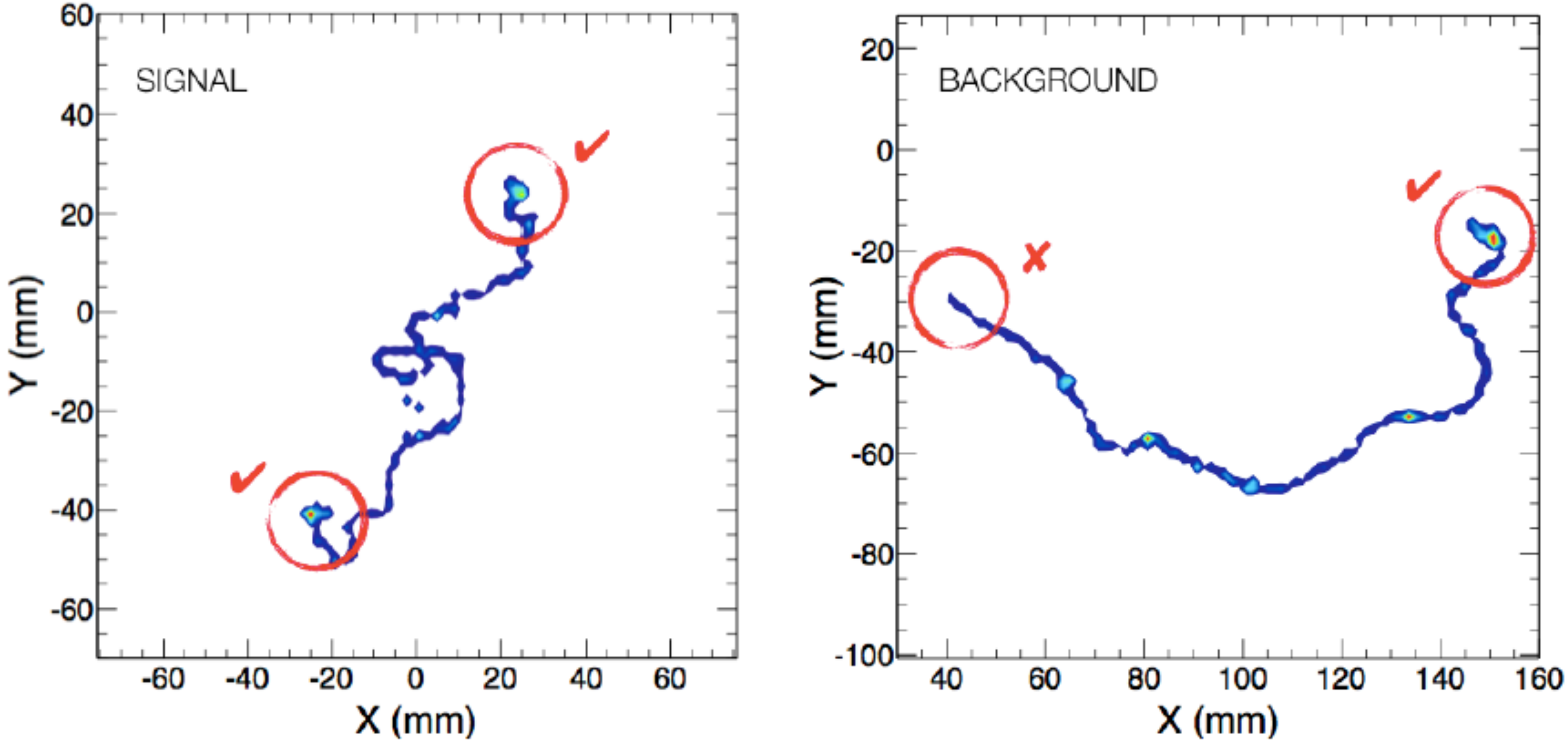
0vßß would yield 2 electrons of total energy = Qßß



- But we could still get background events that are not 0vßß but still fall into the energy peak

Topological signature in NEXT

JHEP 01, 104 (2016) [arXiv:1507.05902]

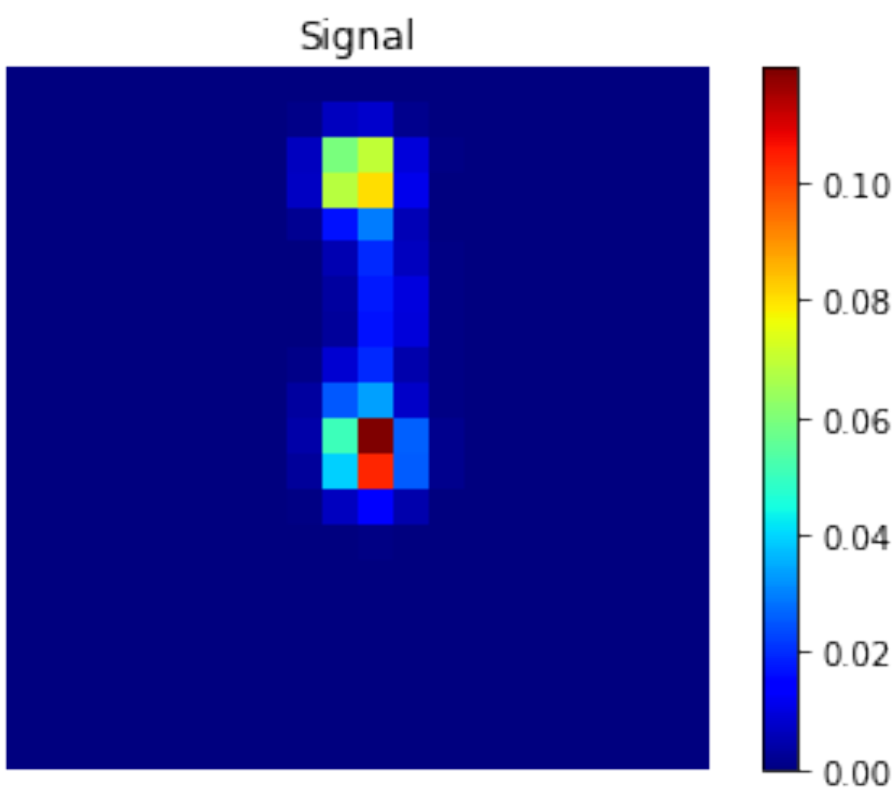


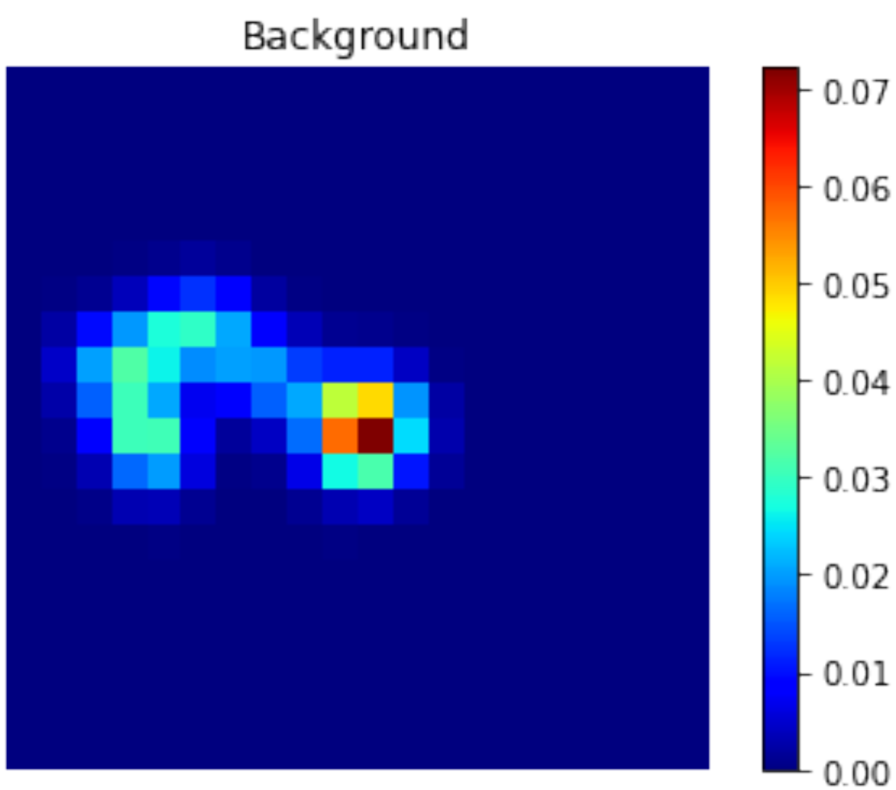
- Energetic electron leaves a high-density deposition at the end of its track (Bragg peak)
- Results in distinct topological signatures for **signal** and **background** events of the same energy

CNNs in Pytorch

NEXT simulation dataset (20x20x60 particle tracks):

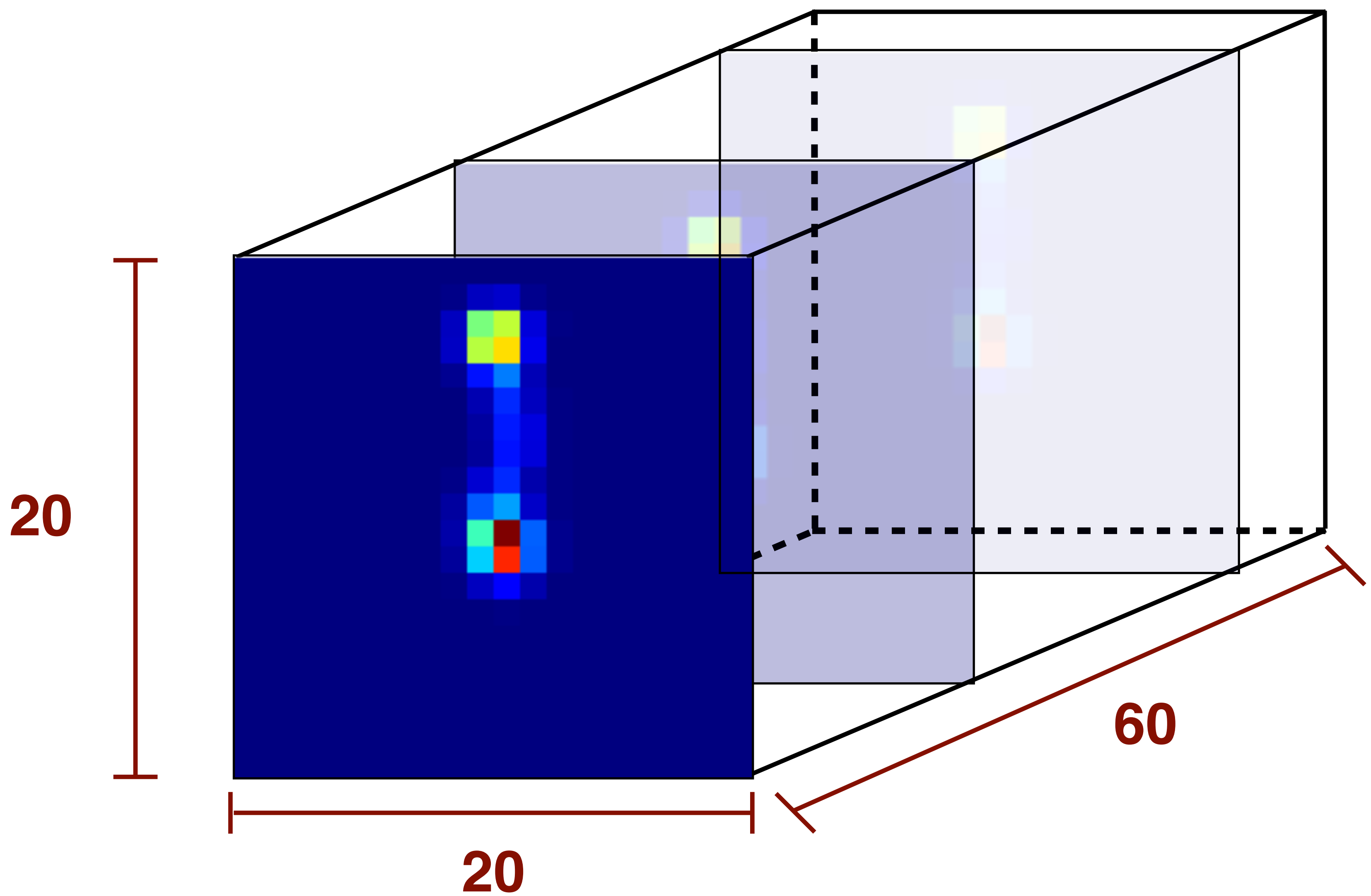
- **Note these are actually e+e- pair events (similar topology and can be produced experimentally)**





CNNs in Pytorch

NEXT simulation dataset (20x20x60 particle tracks):



~ 35000 total events available/class, to be split up into training, validation, and test sets

We'll be dealing with tensors of shape:

`[N, 20, 20, 60]`

"batch size"

Note: Batches

Gradient descent steps are commonly performed with smaller subsets of the full training set called *minibatches* (or “batches”).

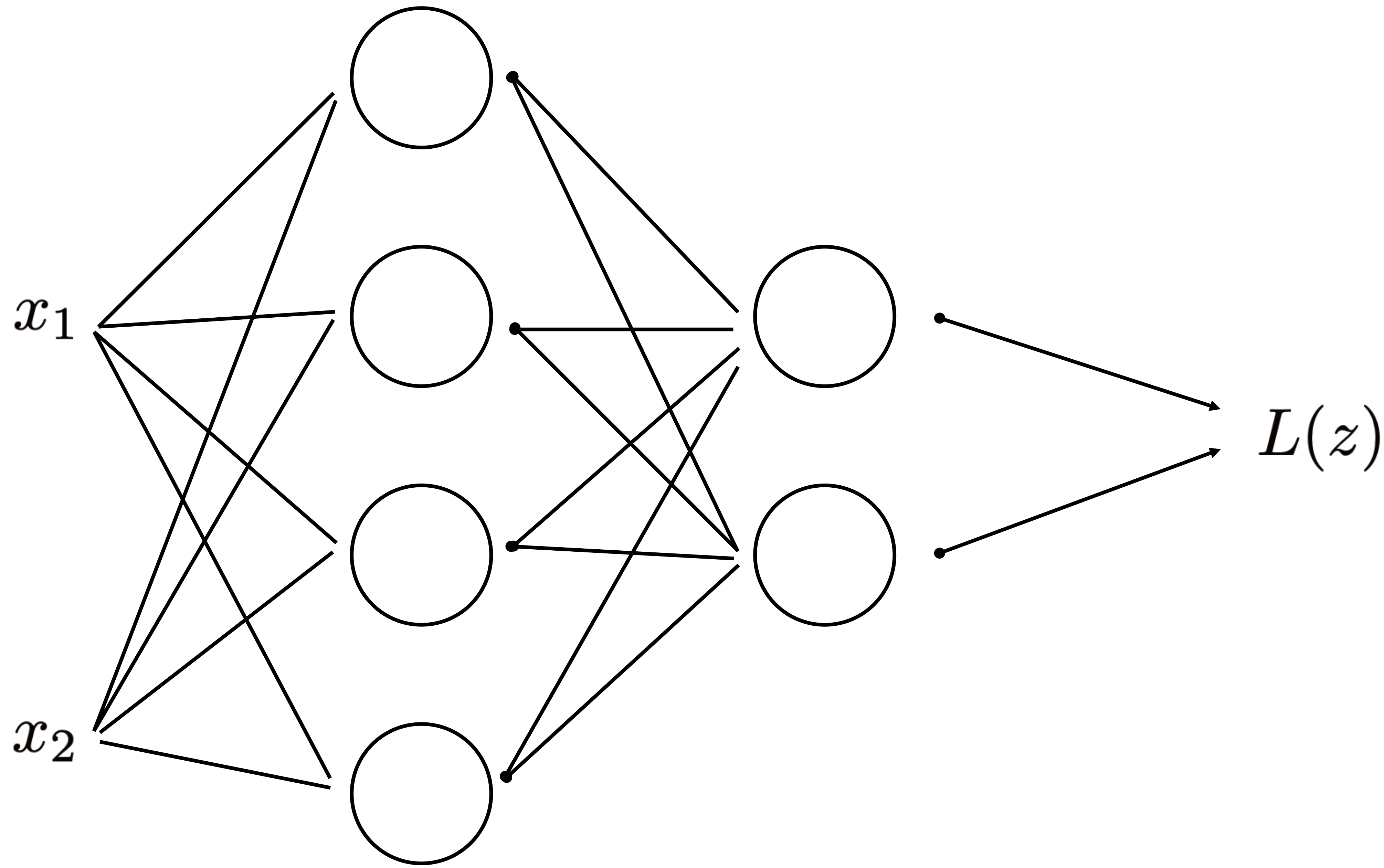
```
# Create a new Dataset
dataset_train = NEXTDataset(datafile_signal,
datafile_background, nstart_train, nend_train)

# Create a new DataLoader
train_loader  = DataLoader(dataset_train,
batch_size=batch_size, shuffle=True)
```

At each step, the loss and gradients will be computed using `batch_size` data samples:

```
for batch_idx, (data, target) in enumerate(train_loader):

    optimizer.zero_grad()
    outputs = model(data)
    loss = criterion(outputs, target)
    loss.backward()
    optimizer.step()
```

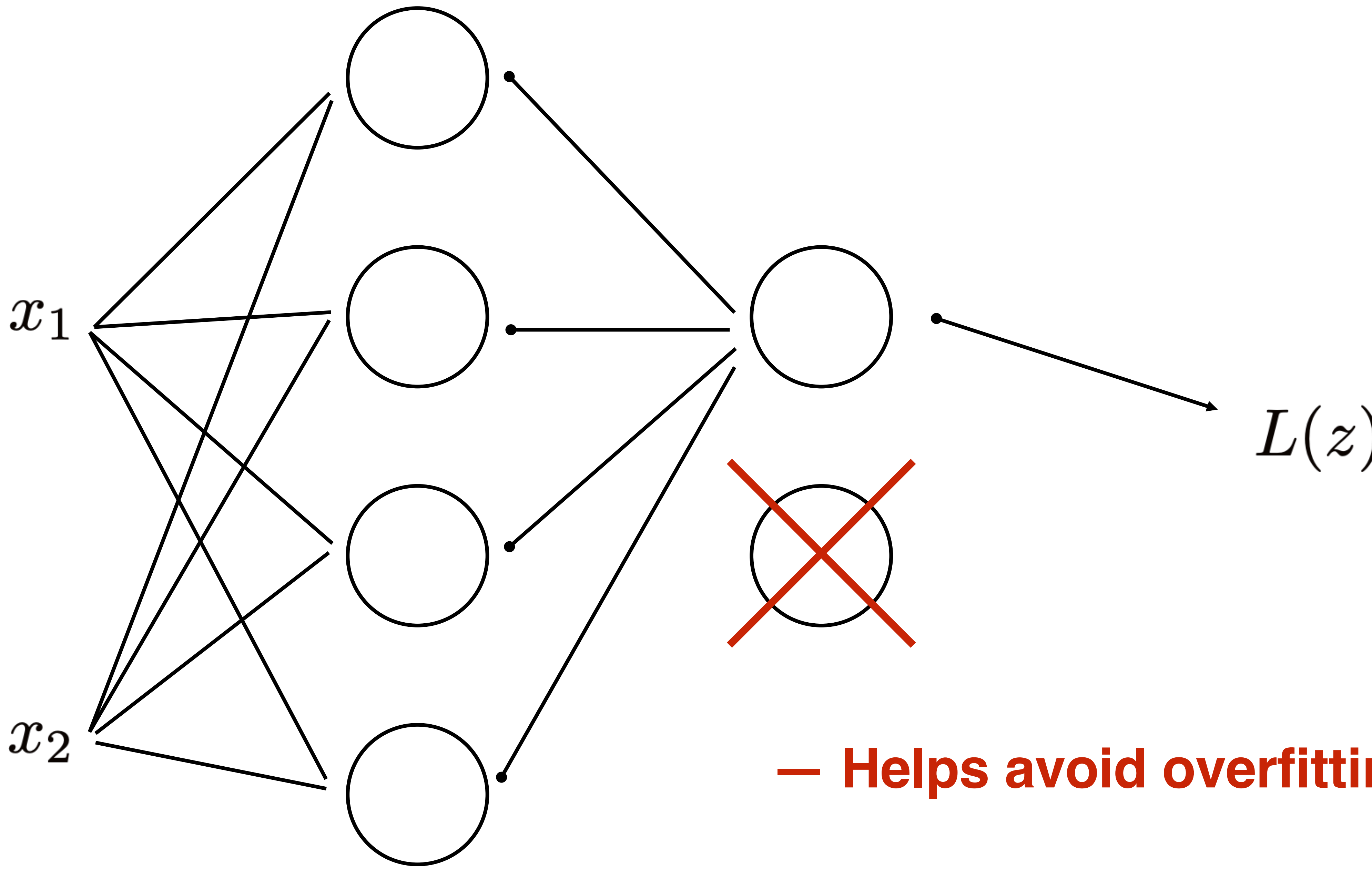
Note: Dropout

During training/backpropagation, some specified probability of removing all connections to a neuron for a step:



Note: Dropout

During training/backpropagation, some specified probability of removing all connections to a neuron for a step:



— Helps avoid overfitting

CNNs in Pytorch

Some relevant layers/commands
(see also https://pytorch.org/docs/stable/nn.html):

```
# Create a new linear (Wx + b) layer
nn.Linear(in_features, out_features)

# Create a new 2D convolutional layer
torch.nn.Conv2d(C_in, C_out, kernel_size, stride, padding)

# Create a new 2D max pooling layer
torch.nn.MaxPool2d(kernel_size, stride)

# Dropout layer with the specified probability
torch.nn.Dropout(prob)


# Apply the sigmoid function
torch.sigmoid(x)

# Apply the ReLU function
torch.relu(x)

# Flatten a tensor (starting after the batch dimension)
torch.flatten(x, start_dim=1)
```

CNNs in Pytorch

Some relevant layers/commands
(see also https://pytorch.org/docs/stable/nn.html):

```
# Create a new linear (Wx + b) layer
nn.Linear(in_features, out_features)

# Create a new 2D convolutional layer
torch.nn.Conv2d(C_in, C_out, kernel_size, stride, padding)

# Create a new 2D max pooling layer
torch.nn.MaxPool2d(kernel_size, stride)

# Dropout layer with the specified probability
torch.nn.Dropout(prob)


# Apply the sigmoid function
torch.sigmoid(x)

# Apply the ReLU function
torch.relu(x)

# Flatten a tensor (starting after the batch dimension)
torch.flatten(x, start_dim=1)
```

ReLU (rectified linear units) is another common activation used in CNNs

relu(x) = max(0,x)

CNNs in Pytorch

An example net (not a CNN):

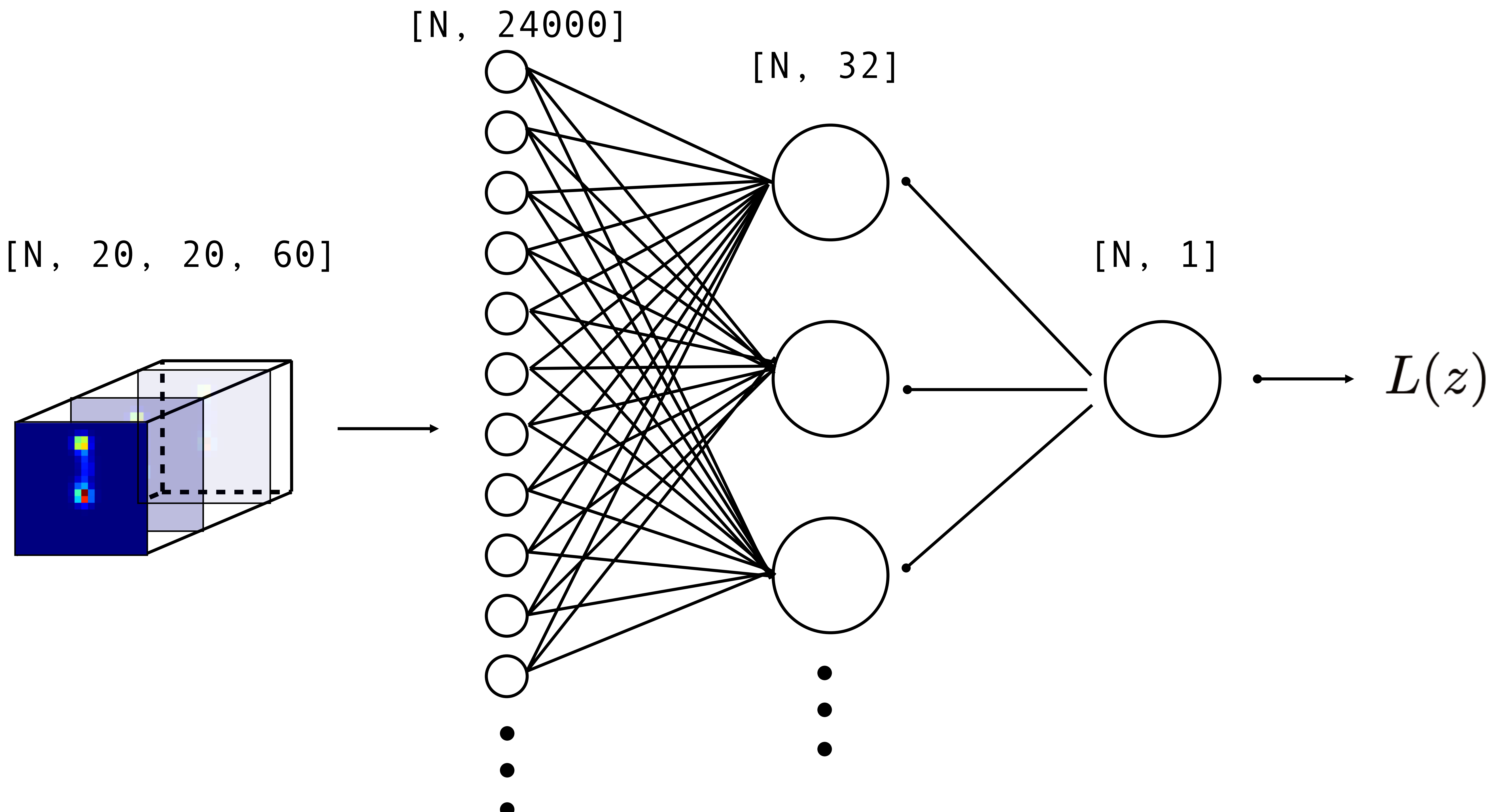
```
class FCNet(nn.Module):

    def __init__(self):
        super(FCNet, self).__init__()
        self.fc1 = nn.Linear(xdim*ydim*zdim, 32)
        self.fc2 = nn.Linear(32, 1)

    def forward(self, x):
        x = torch.flatten(x, start_dim=1)
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        return x
```

define layers

apply layers

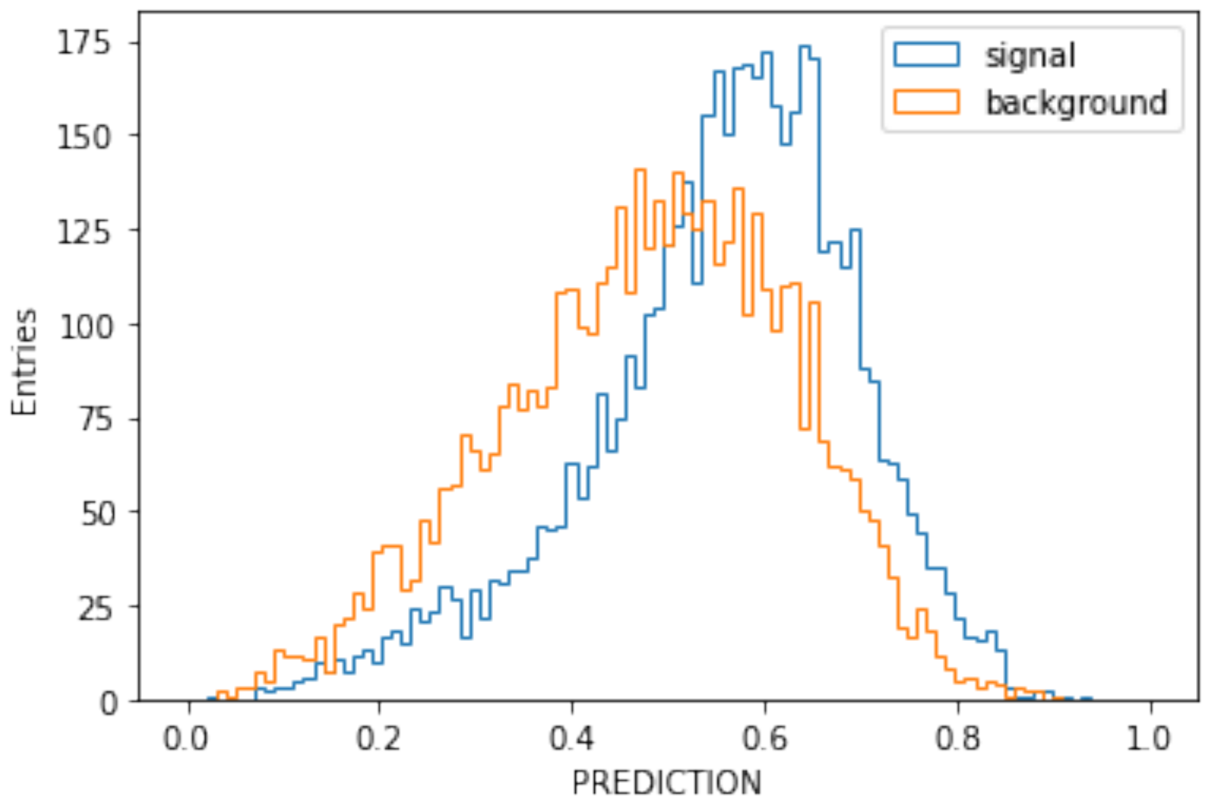
CNNs in Pytorch

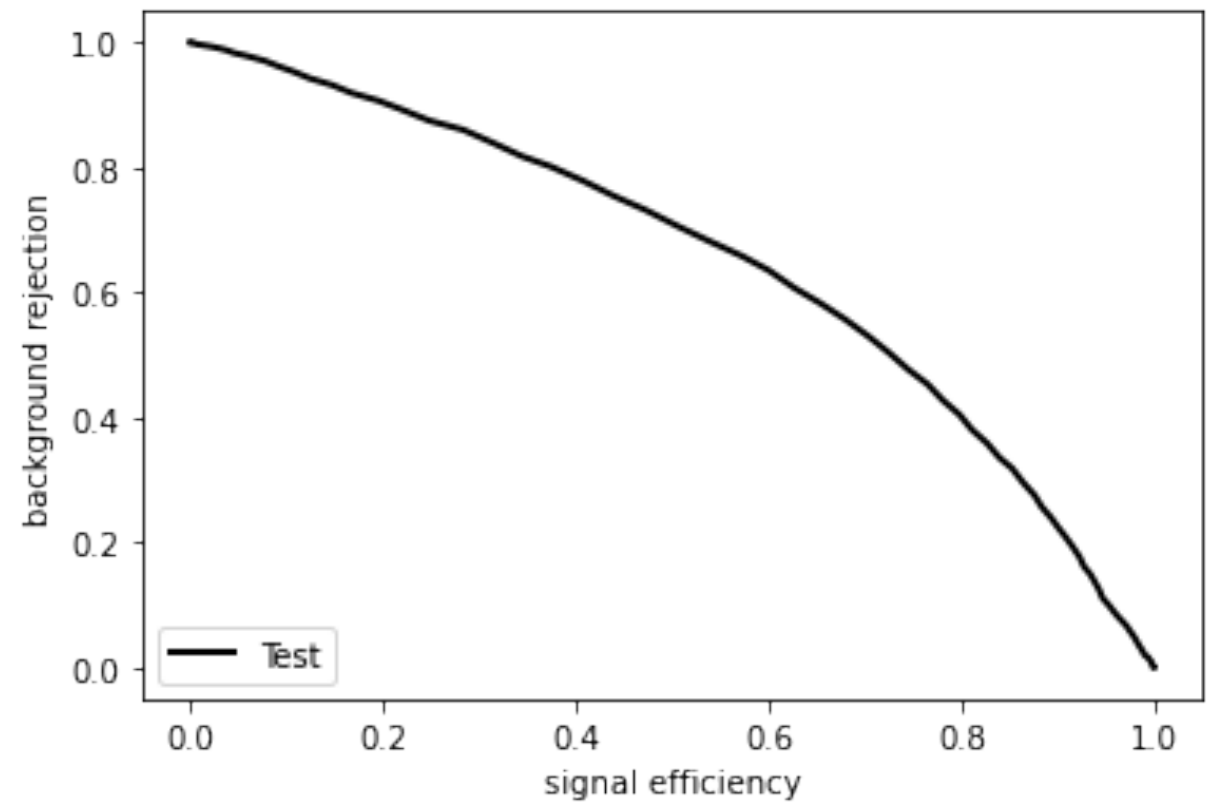
An example net (not convolutional):



CNNs in Pytorch

Performance: **can this be improved with a CNN?**





https://colab.research.google.com/github/jerenner/uscnn course/blob/master/next/NEXT_classification.ipynb

EXTRA SLIDES

Neural networks

Backpropagation on a graph is essentially the chain rule:

$$\frac{\partial L}{\partial W_i} = \sum_j \frac{\partial L}{\partial z_j} \sum_k \frac{\partial z_j}{\partial y_k} \frac{\partial y_k}{\partial W_i} = \sum_{j,k} \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial y_k} \frac{\partial y_k}{\partial W_i} = \sum_{j,k} \frac{\partial y_k}{\partial W_i} \frac{\partial z_j}{\partial y_k} \frac{\partial L}{\partial z_j}$$

$= \sum_{j,k} \frac{\partial y_k}{\partial W_i} \left[\frac{\partial z_j}{\partial y_k} \left(\frac{\partial L}{\partial z_j} \right) \right] = \sum_k \frac{\partial y_k}{\partial W_i} \sum_j \left[\frac{\partial z_j}{\partial y_k} dz_j \right]$, where $dz_j = \frac{\partial L}{\partial z_j}$

$= \sum_k \frac{\partial y_k}{\partial W_i} dy_k$, where $dy_k = \sum_j \left[\frac{\partial z_j}{\partial y_k} dz_j \right]$