Introduction to machine learning and neural networks

Toby Dylan Hocking
toby.hocking@nau.edu
toby.hocking@r-project.org

March 19, 2021

Fully connected multi-layer Neural Networks

Computing gradients and learning weights

Convolutional networks and pooling

Supervised learning setup

- Have an input $\mathbf{x} \in \mathbb{R}^d$ – a vector of d real numbers.
- And an output y (real number: regression, integer ID: classification).
- Want to learn a prediction function $f(\mathbf{x}) = y$ that will work on a new input.
- In a neural network with $L-1$ hidden layers the function f is defined using composition of L functions, $f(x) = f^{(L)}[\ldots f^{(1)}[x]] \in \mathbb{R}$.

Each function is matrix multiplication and activation

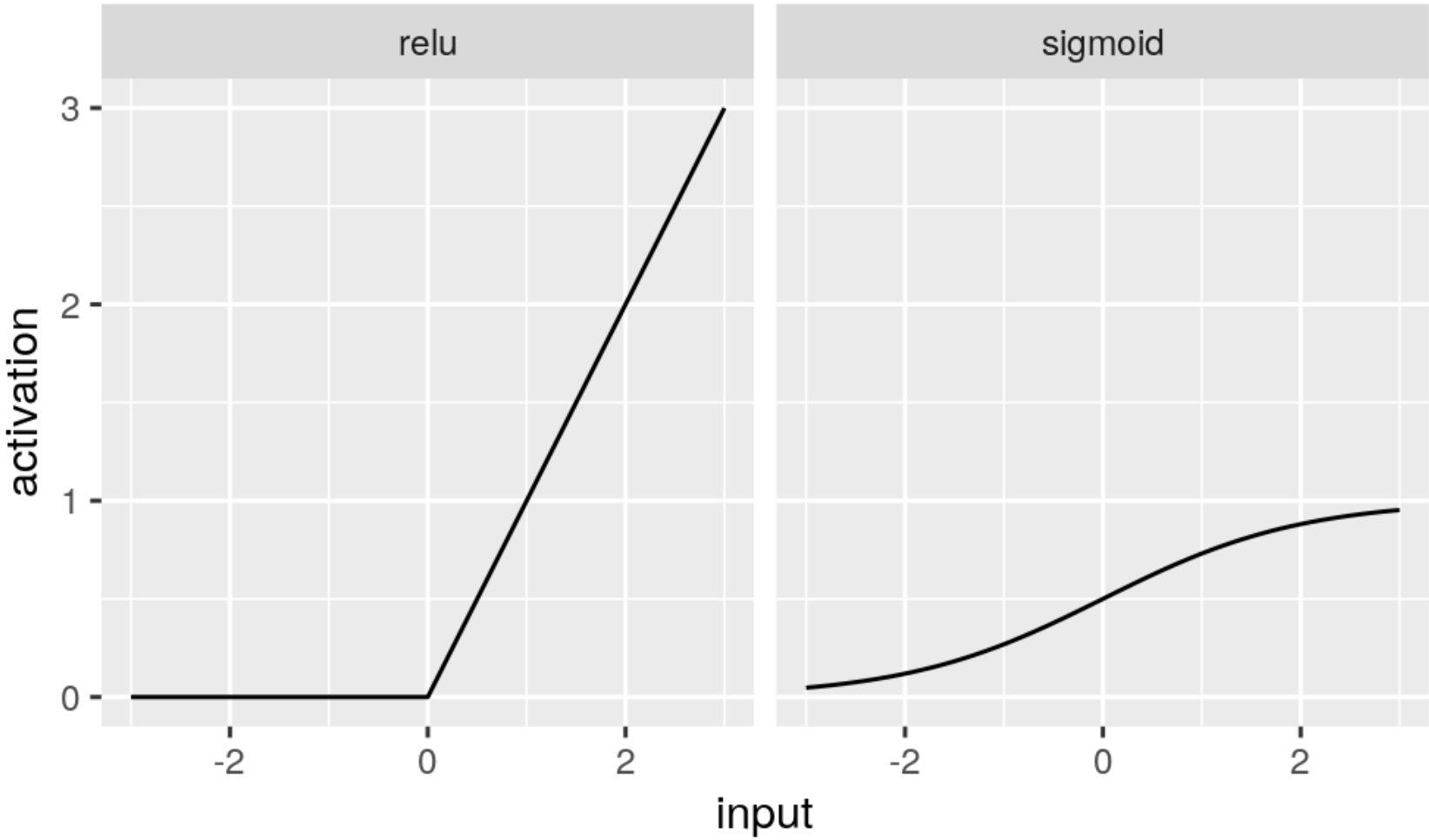
- Prediction function $f(x) = f^{(L)}[\cdots f^{(1)}[x]] \in \mathbb{R}$.
- Each function $l \in \{1, \ldots, L\}$ is a matrix multiplication followed by an activation function: $f^{(l)}[z] = \sigma^{(l)}[W^{(l)}z]$ where $W^{(l)} \in \mathbb{R}^{u^{(l)} \times u^{(l-1)}}$ is a weight matrix to learn, and $z \in \mathbb{R}^{u^{(l-1)}}$ is the input vector to that layer.
- In regression the last activation function must return a real number prediction so it is fixed to the identity: $\sigma^{(L)}[z] = z$.
- The other activation functions must be non-linear, e.g. logistic/sigmoid $\sigma(z) = 1/(1 + \exp(-z))$ or rectified linear units (ReLU)

$$\sigma(z) = \begin{cases} z & \text{if } z > 0, \\ 0 & \text{else.} \end{cases}$$

Non-linear activation functions

$$\sigma(z) = \begin{cases} z & \text{if } z > 0, \\ 0 & \text{else.} \end{cases}$$

$$\sigma(z) = 1/(1 + \exp(-z))$$



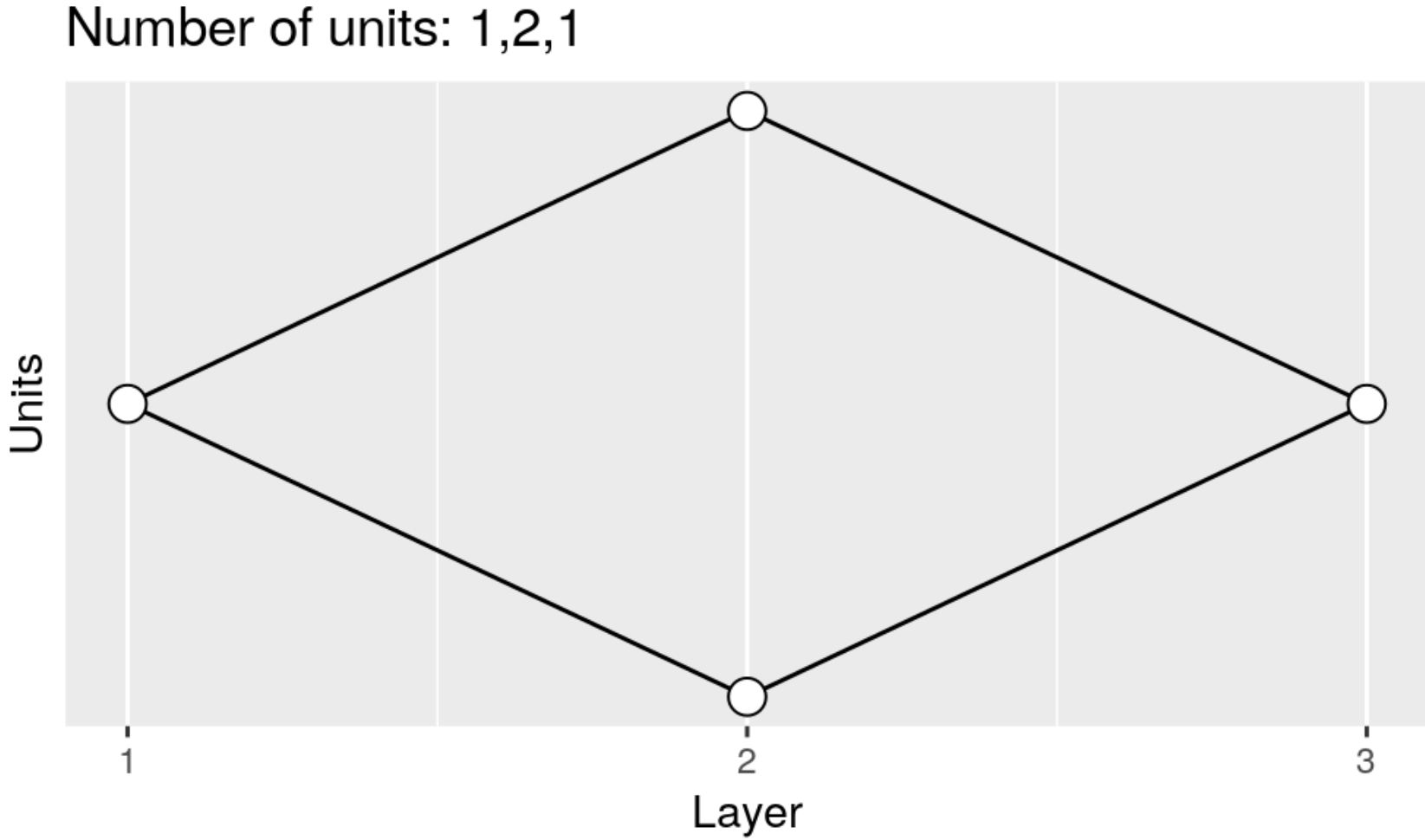
Network size

For binary classification with inputs $x \in \mathbb{R}^d$, the overall neural network architecture is $(u^{(0)} = d, u^{(1)}, \ldots, u^{(L-1)}, u^{(L)} = 1)$, where $u^{(1)}, \ldots, u^{(L-1)} \in \mathbb{Z}_+$ are positive integers (hyper-parameters that control the number of units in each hidden layer, and the size of the parameter matrices $W^{(l)}$).

- First layer size $u^{(0)}$ is fixed to input size.
- Last layer size $u^{(L)}$ is fixed to output size.
- Number of layers and hidden layer sizes $u^{(1)}, \ldots, u^{(L-1)}$ must be chosen (by you).

Network diagrams

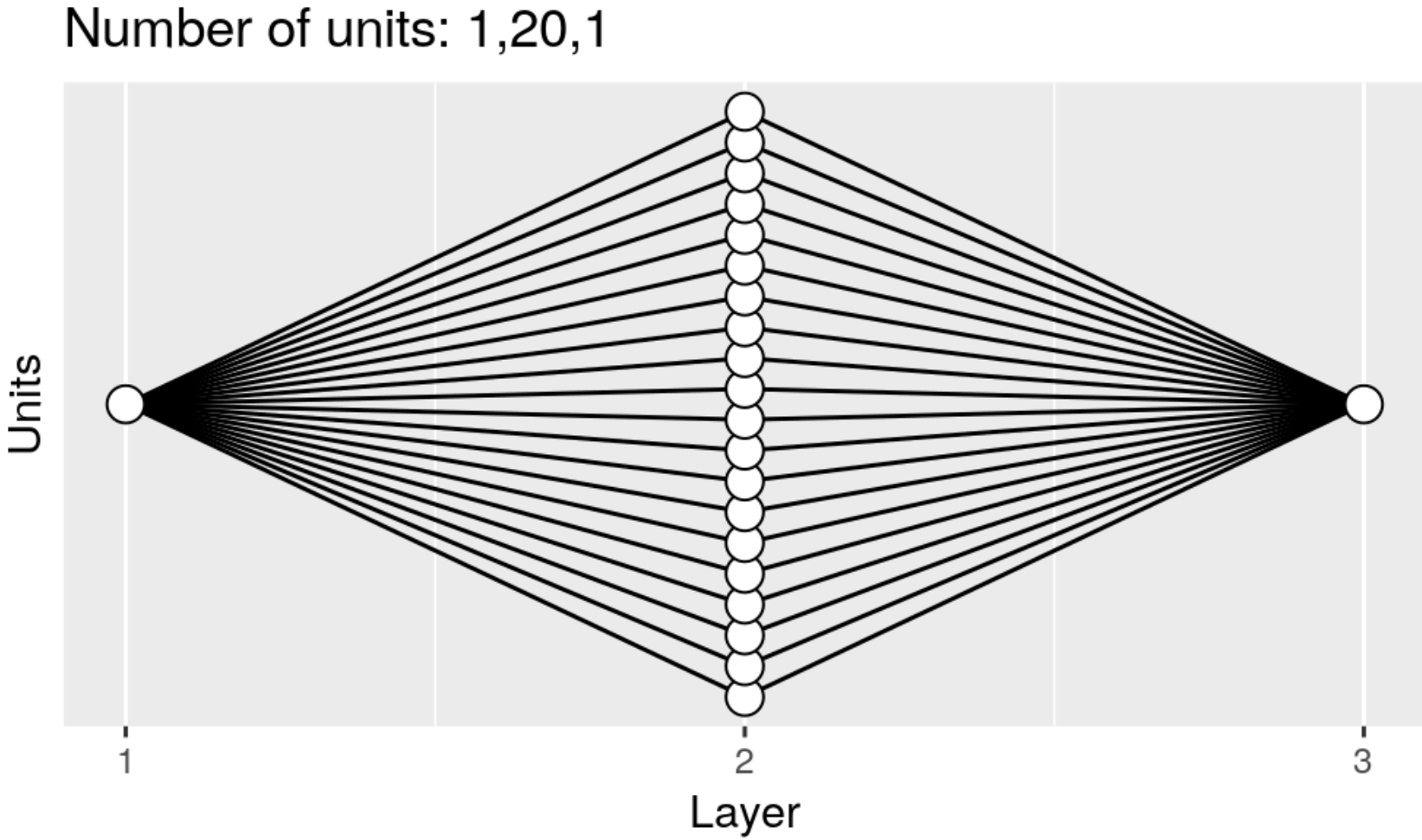
Neural network diagrams show how each hidden unit (node) is computed by applying the weights (edges) to the values of the hidden units at the previous layer.



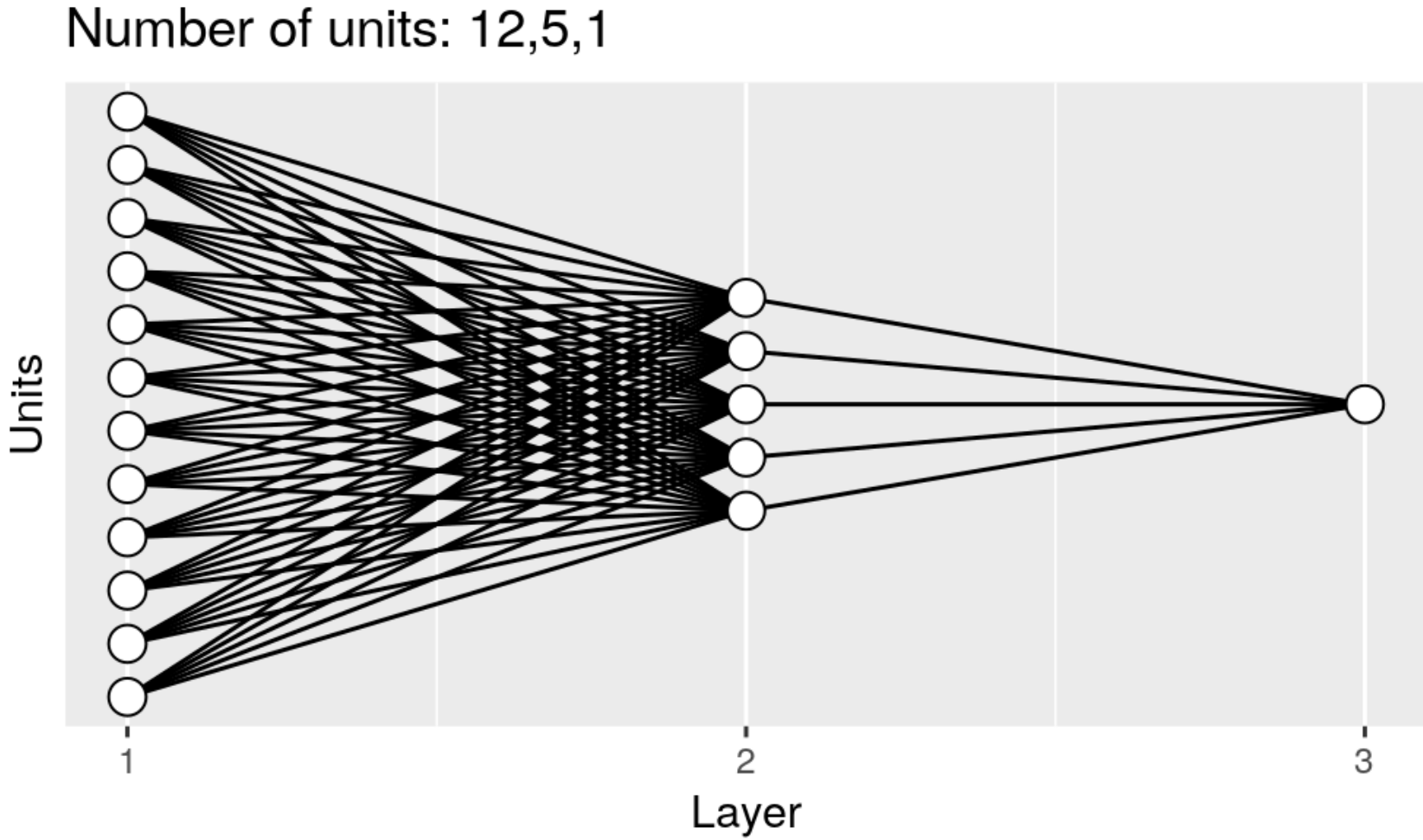
torch code

```
import torch
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        n_hidden = 2
        self.act = torch.nn.Sigmoid()
        self.hidden = torch.nn.Linear(1, n_hidden)
        self.out = torch.nn.Linear(n_hidden, 1)
    def forward(self, x):
        x = self.act(self.hidden(x))
        x = self.out(x)
        return x
```

Network diagrams



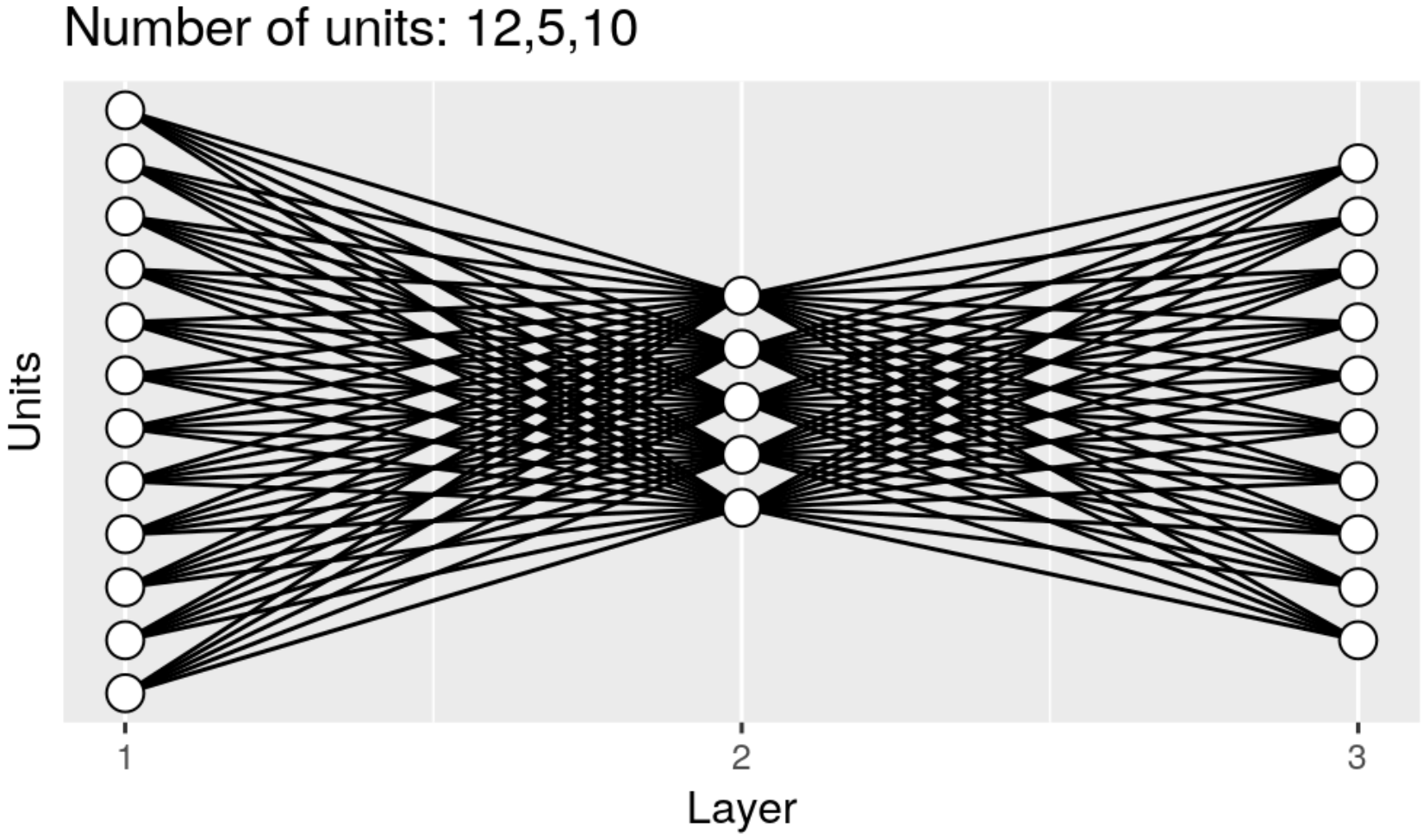
Network diagrams



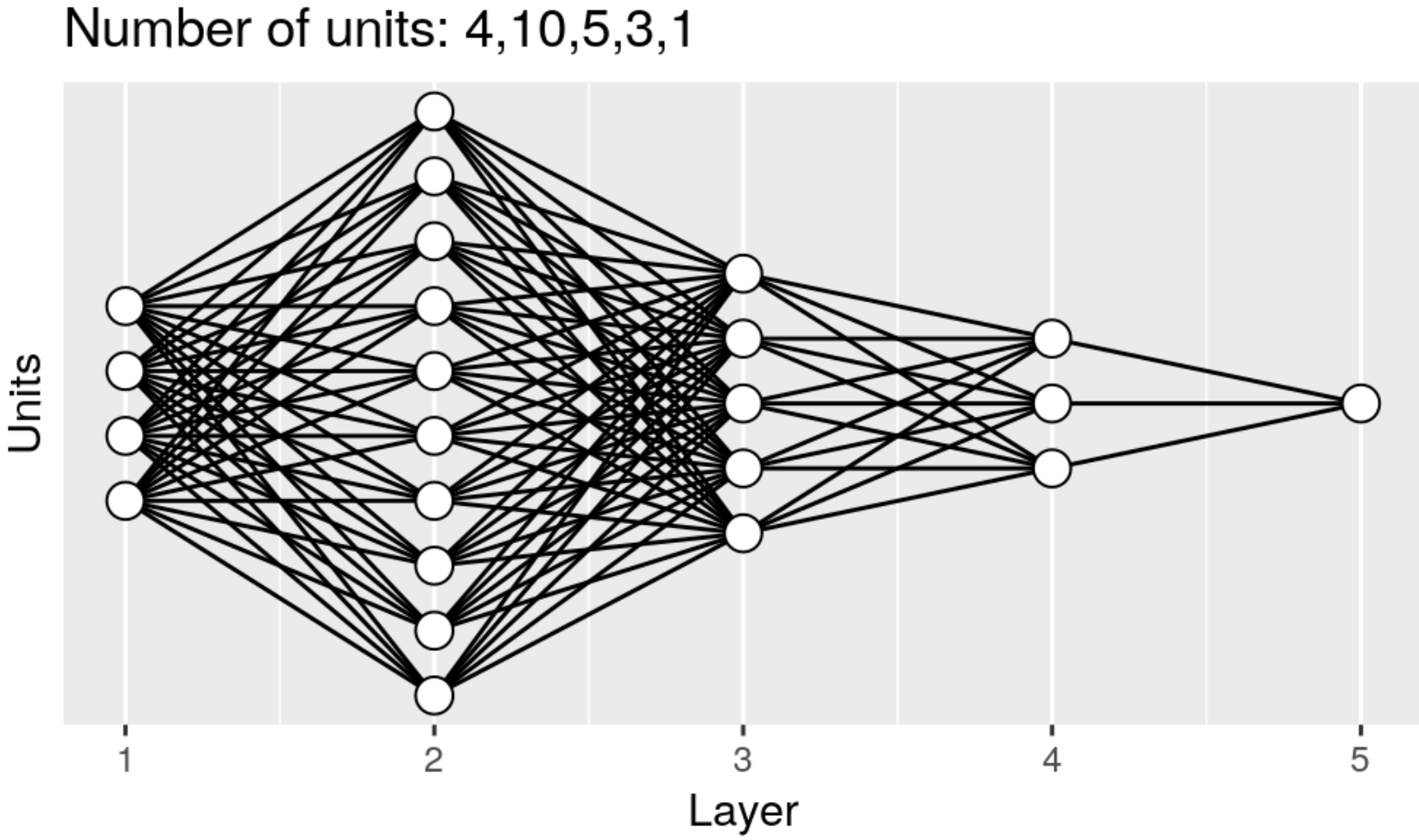
torch code

```
import torch
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        n_hidden = 5
        self.act = torch.nn.Sigmoid()
        self.hidden = torch.nn.Linear(12, n_hidden)
        self.out = torch.nn.Linear(n_hidden, 1)
    def forward(self, x):
        x = self.act(self.hidden(x))
        x = self.out(x)
        return x
```

Network diagrams



Network diagrams



Units in each layer

We can write the units at each layer as $h^{(0)}, h^{(1)}, \ldots, h^{(L-1)}, h^{(L)}$ where

- $h^{(0)} = x \in \mathbb{R}^d$ is an input feature vector,
- and $h^{(L)} \in \mathbb{R}$ is the predicted output.

For each layer $l \in \{1, \ldots, L\}$ we have:

$$h_l = f^{(l)}\left[h^{(l-1)}\right] = \sigma^{(l)}\left[W^{(l)} h^{(l-1)}\right].$$

Total number of parameters to learn is $\sum_{l=1}^{L} u^{(l)} u^{(l-1)}$.
Quiz: how many parameters in a neural network for $d = 10$ inputs/features with one hidden layer with $u = 100$ units? (one output unit, ten output units)

Fully connected multi-layer Neural Networks

Computing gradients and learning weights

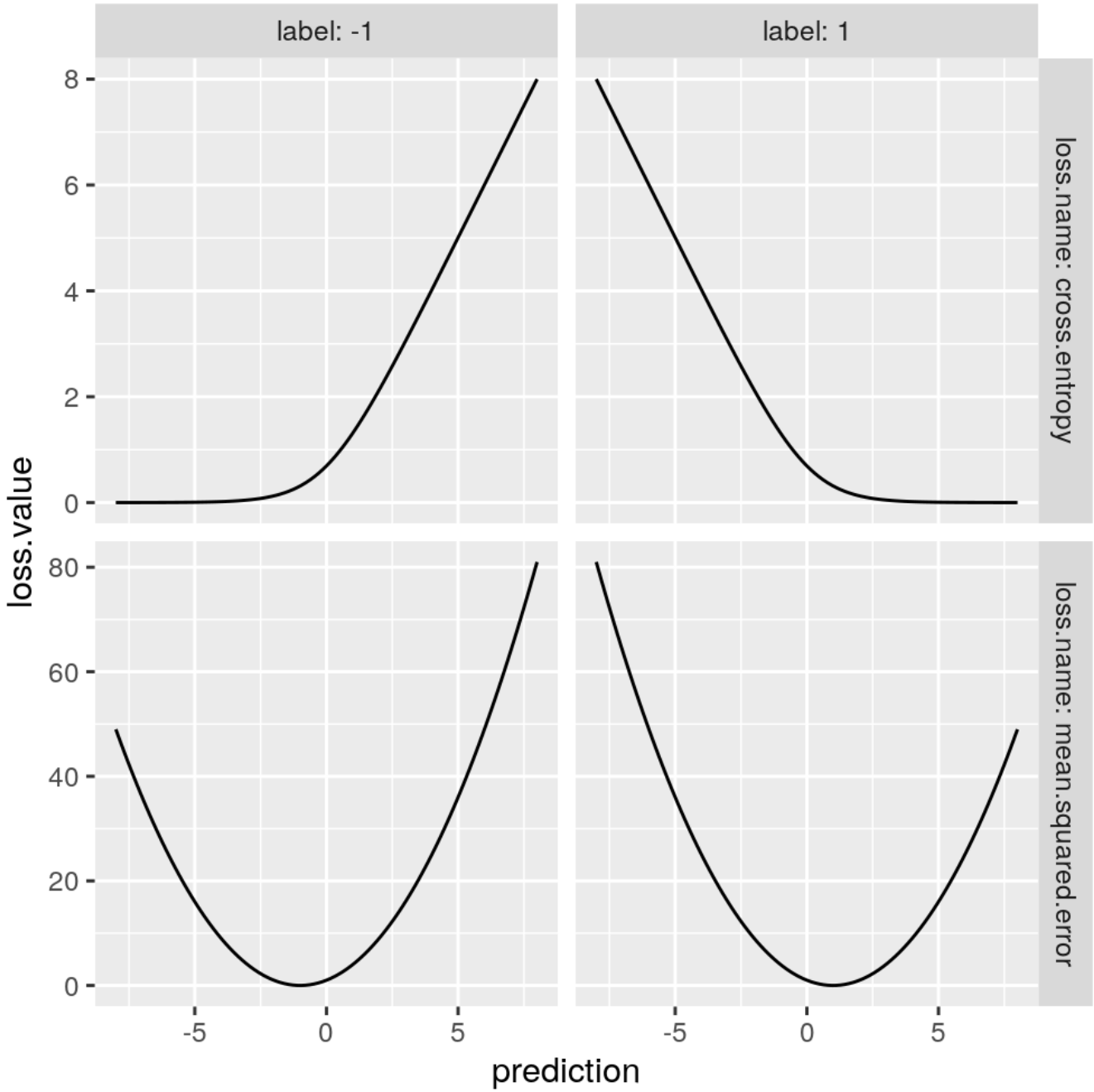
Convolutional networks and pooling

Gradient descent learning

Basic idea of gradient descent learning algorithm is to iteratively update weights $\mathbf{W} = [W^{(1)}, \ldots, W^{(L)}]$ to improve predictions on the subtrain set.

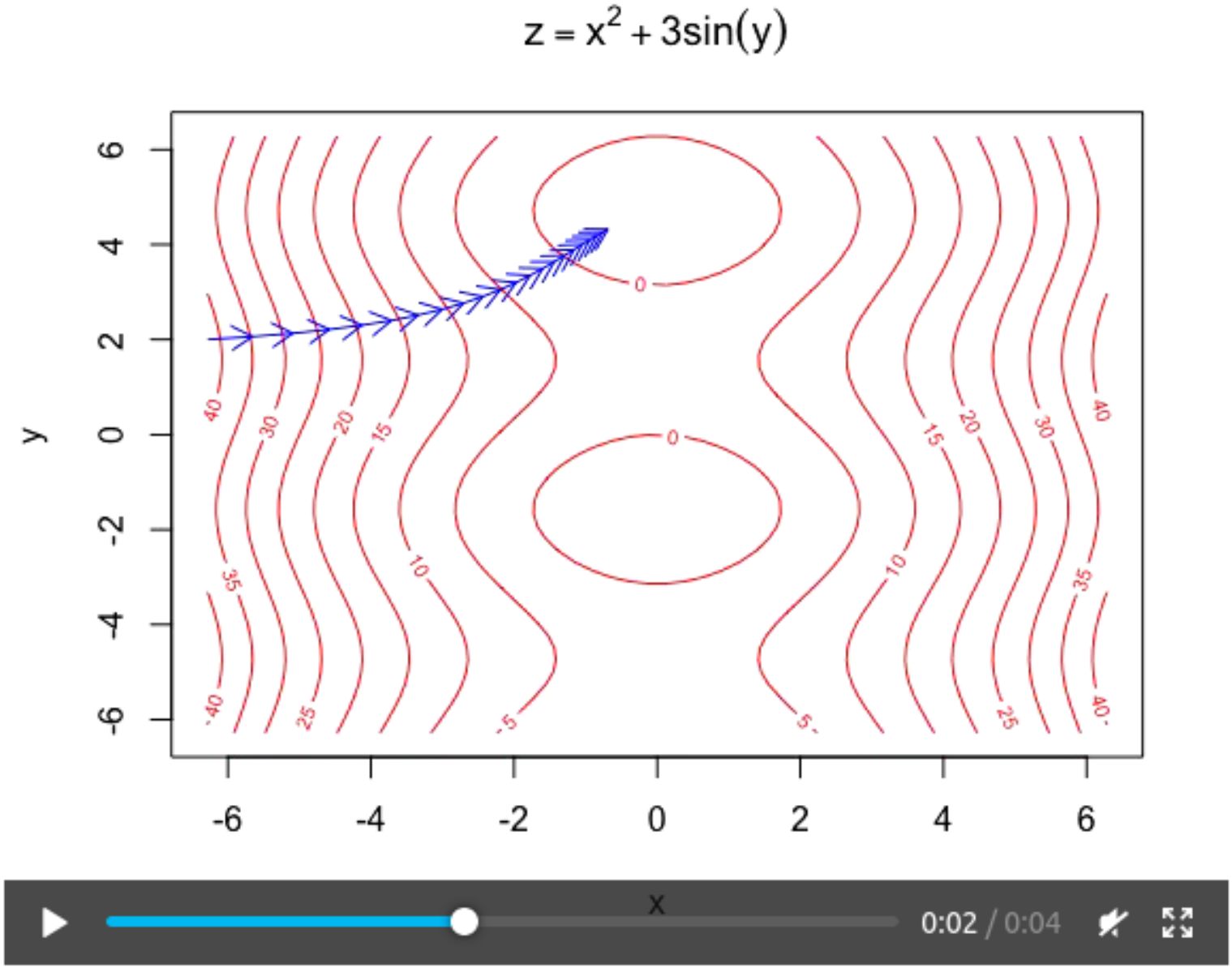
- Need to define a loss function $\mathcal{L}(\mathbf{W})$ which is differentiable, and takes small values for good predictions.
- Typically for regression we use the mean squared error, and for binary classification we use the logistic (cross entropy) loss.
- The gradient $\nabla\mathcal{L}(\mathbf{W})$ is a function which tells us the local direction where the loss is most increasing.

Loss functions



Gradient descent animations

https://yihui.org/animation/example/grad-desc/



Basic full gradient descent algorithm

- Initialize weights $\mathbf{W}_0$ at some random values near zero (more complicated initializations possible).
- Since we want to decrease the loss, we take a step α in the opposite direction of the gradient,

$$\mathbf{W}_t = \mathbf{W}_{t-1} - \alpha \nabla \mathcal{L}(\mathbf{W}_{t-1})$$

- This is the **full** gradient method: batch size $= n =$ subtrain set size, so 1 step per epoch/iteration.

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.03)
optimizer.zero_grad()
predictions = net(subtrain_inputs)
subtrain_loss = criterion(predictions, subtrain_outputs)
subtrain_loss.backward()
optimizer.step()
```

Stochastic gradient descent algorithm

- Initialize weights $\mathbf{W}$ at some random values near zero (more complicated initializations possible).
- for each epoch t from 1 to max epochs:
- for each batch i from 1 to n:
- Let $\mathcal{L}(\mathbf{W}, \mathbf{X}_i, \mathbf{y}_i)$ be the loss with respect to the single observation in batch i.

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla \mathcal{L}(\mathbf{W}, \mathbf{X}_i, \mathbf{y}_i)$$

- This is the **stochastic** gradient method: batch size $= 1$, so there are n steps per epoch.

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.03)
optimizer.zero_grad()
prediction = net(one_input)
one_loss = criterion(prediction, one_output)
one_loss.backward()
optimizer.step()
```

Batch (stochastic) gradient descent algorithm

- Input: batch size b.
- Initialize weights $\mathbf{W}$ at some random values near zero (more complicated initializations possible).
- for each epoch t from 1 to max epochs:
- for each batch i from 1 to $\lceil n/b \rceil$:
- Let $\mathcal{L}(\mathbf{W}, \mathbf{X}_i, \mathbf{y}_i)$ be the loss with respect to the b observations in batch i.

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla \mathcal{L}(\mathbf{W}, \mathbf{X}_i, \mathbf{y}_i)$$

- This is the **(mini)batch** stochastic gradient method: batch size $= b$, so there are $\lceil n/b \rceil$ steps per epoch.

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.03)
optimizer.zero_grad()
prediction = net(batch_inputs)
batch_loss = criterion(prediction, batch_outputs)
batch_loss.backward()
optimizer.step()
```

Forward propagation

Forward propagation is the computation of hidden units $h^{(1)}, \ldots, h^{(L)}$ given the inputs x and current parameters $W^{(1)}, \ldots, W^{(L)}$.

```
def forward(self, x):
    x = self.act(self.hidden(x))
    x = self.out(x)
    return x
```

(start from input, apply weights and activation in each layer until predicted output is computed)

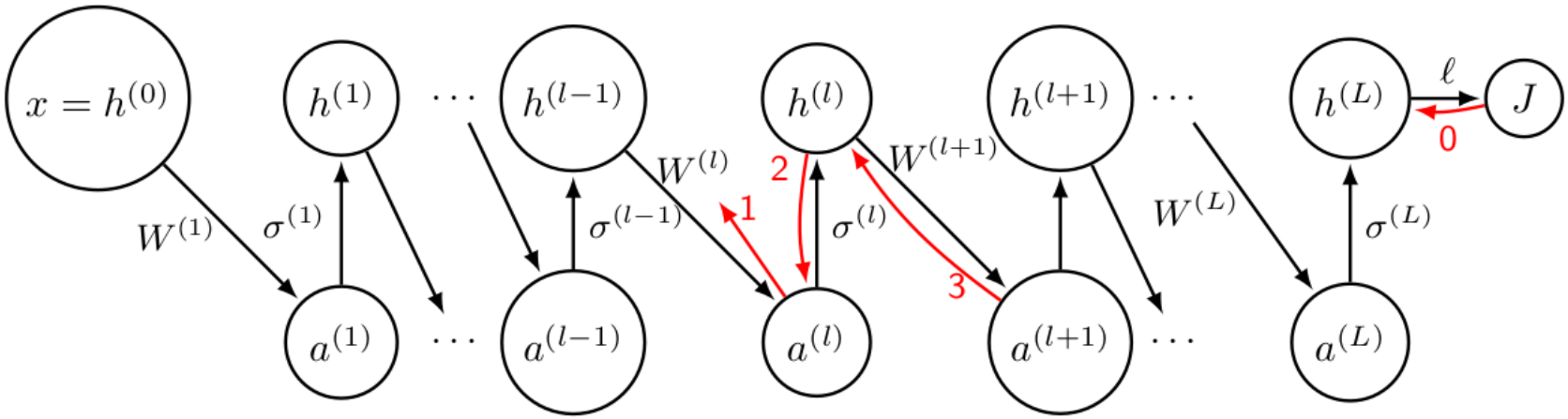
Back propagation

Back propagation is the computation of gradients given current parameters and hidden units.

- Start from loss function, compute gradient, send it to last layer, use chain rule, send gradient to previous layer, finally end up at first layer.
- Result is gradients with respect to all weights in all layers.
- Modern frameworks like torch do this using automatic differentiation based on your definition of the forward method and the loss function.

```
net = Net()
criterion = torch.nn.MSELoss()
optimizer = torch.optim.LBFGS(net.parameters(), lr=0.03)
optimizer.zero_grad()
pred = net(input_X_features)
loss = criterion(pred, output_y_labels)
loss.backward()
```

Computation graph



For each layer $l \in \{1, \ldots, L\}$ we have:

$$\begin{aligned} a^{(l)} &= W^{(l)} h^{(l-1)}, \\ h^{(l)} &= \sigma^{(l)} \left[a^{(l)} \right]. \end{aligned}$$

There are essentially four rules for computing gradients during backpropagation (0-3).

Backprop rules

The rules 0–3 for backprop (from loss backwards):

Rule 0 computes $\nabla_{h^{(L)}} J$, which depends on the choice of the loss function ℓ.

Rule 1 computes $\nabla_{W^{(l)}} J$ using $\nabla_{a^{(l)}} J$, for any $l \in \{1, \ldots, L\}$

$$\nabla_{w_k^{(l)}} J = (\nabla_{a^{(l)}} J) \left(h^{(l-1)}\right)^T \tag{1}$$

Rule 2 computes $\nabla_{a^{(l)}} J$ using $\nabla_{h^{(l)}} J$, for any $l \in \{1, \ldots, L\}$.

$$\nabla_{a^{(l)}} J = (\nabla_{h^{(l)}} J) \odot \left(\nabla_{a^{(l)}} h^{(l)}\right) \tag{2}$$

Rule 3 computes $\nabla_{h^{(l)}} J$ using $\nabla_{a^{(l+1)}} J$, for any $l \in \{1, \ldots, L-1\}$.

$$\nabla_{h^{(l)}} J = (\nabla_{a^{(l+1)}} J) \left(W^{(l+1)}\right)^T \tag{3}$$

Fully connected multi-layer Neural Networks

Computing gradients and learning weights

Convolutional networks and pooling

Convolution is a linear operator for spatial data

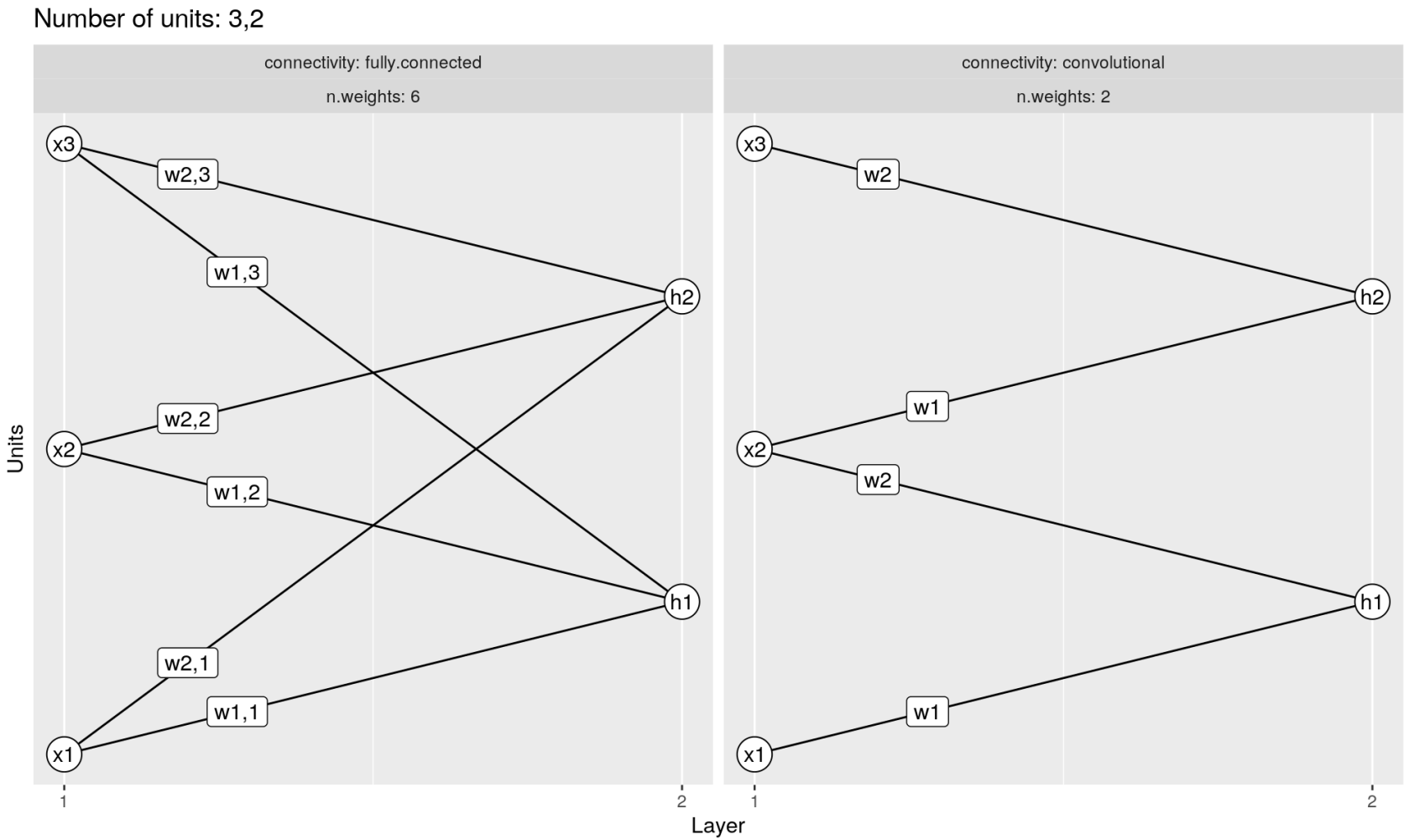
Useful for data which have spatial dimension(s) such as time series (1 dim) or images (2 dim). Simple example with 1 dim:

- $x = [x_1, \ldots, x_D]$ is an input vector (array of D data).
- $w = [w_1, \ldots, w_P]$ is a kernel (array of P parameters / weights to learn), $P < D$.
- $h = [h_1, \ldots, h_U]$ is an output vector of $U = d - p + 1$ hidden units. Convolution (actually cross-correlation) is used to define each hidden unit: $\forall u \in \{1, \ldots, U\}, h_u = \sum_{p=1}^{P} w_p x_{u+p}$.
- EX: $D = 3$ inputs, $P = 2$ parameters $\Rightarrow U = 2$ output units:

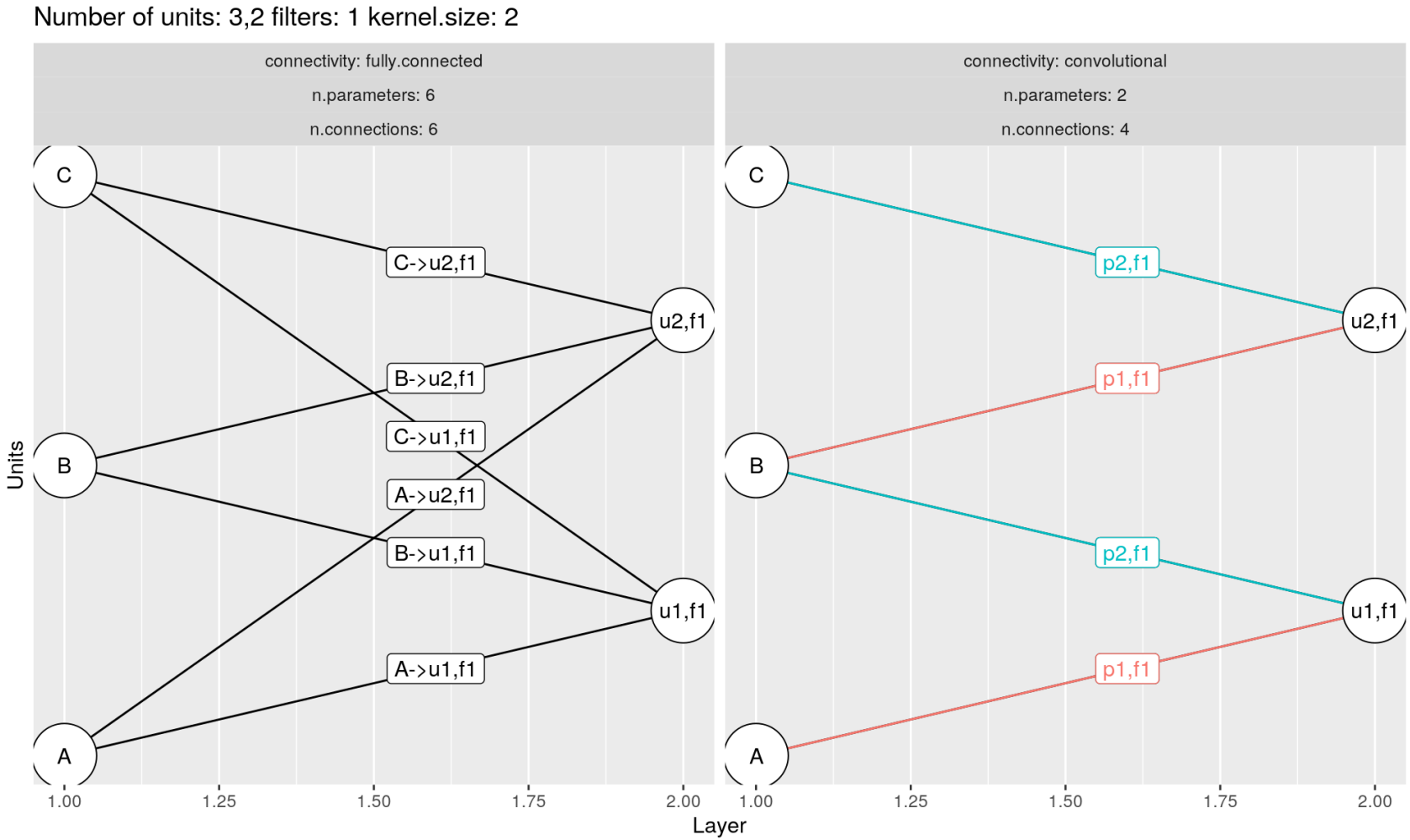
$$
\begin{aligned}
h_1 &= w_1 x_1 + w_2 x_2 \text{ (convolutional=sparse+shared)} \\
h_2 &= w_1 x_2 + w_2 x_3 \text{ (convolutional=sparse+shared)} \\
h_1 &= w_{1,1} x_1 + w_{1,2} x_2 + w_{1,3} x_3 \text{ (fully connected/dense)} \\
h_2 &= w_{2,1} x_1 + w_{2,2} x_2 + w_{2,3} x_3 \text{ (fully connected/dense)}
\end{aligned}
$$

- Compare with fully connected – convolutional means weights are shared among outputs, and some are zero/sparse.

Difference in connectivity and weight sharing

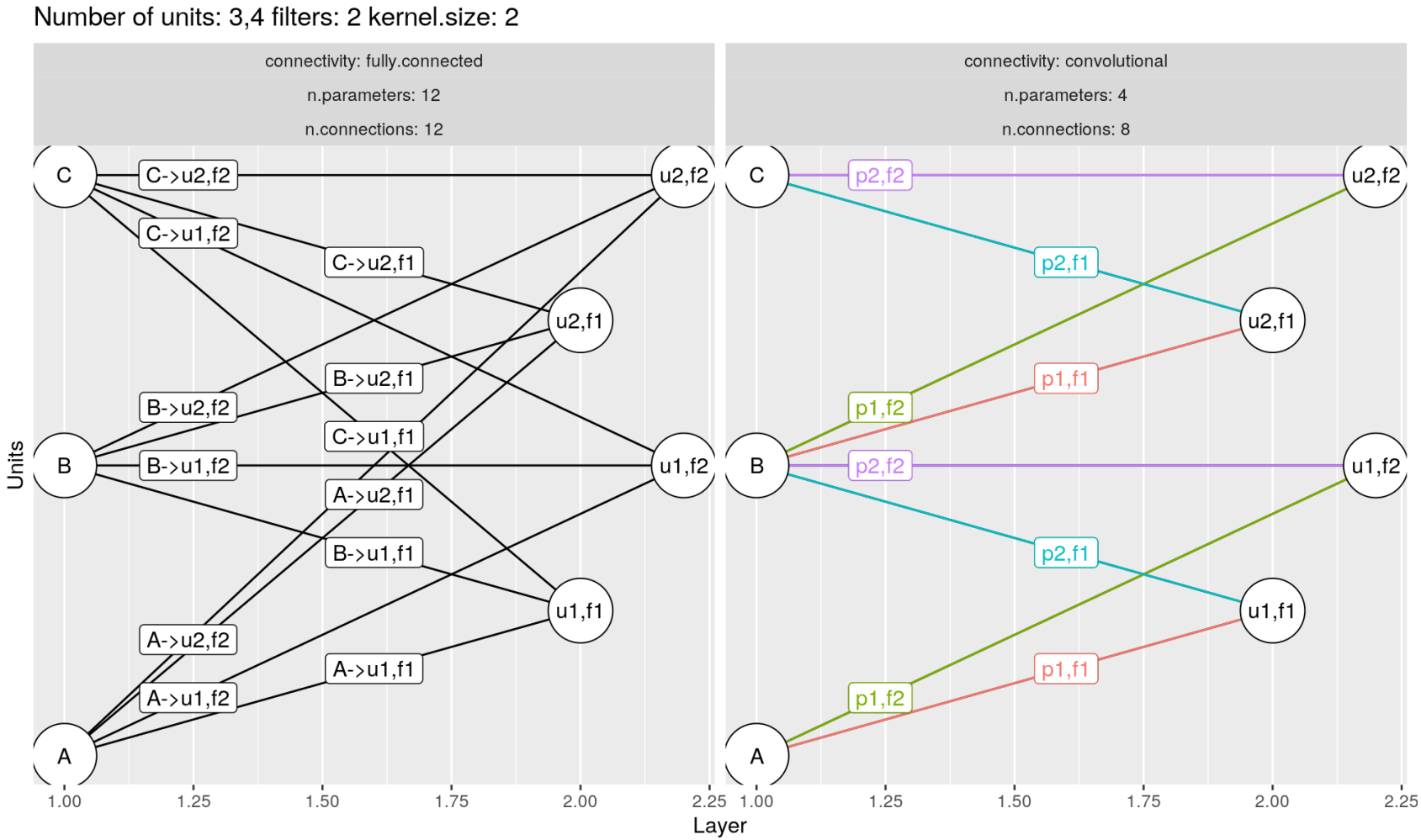


Difference in connectivity and weight sharing



Abbreviations: p=parameter, f=filter, u=unit. e.g., p2,f1=parameter 2 of filter 1. u2,f1=unit 2 using filter 1.

Two filters



Abbreviations: p=parameter, f=filter, u=unit. e.g., p2,f1=parameter 2 of filter 1. u2,f1=unit 2 using filter 1.

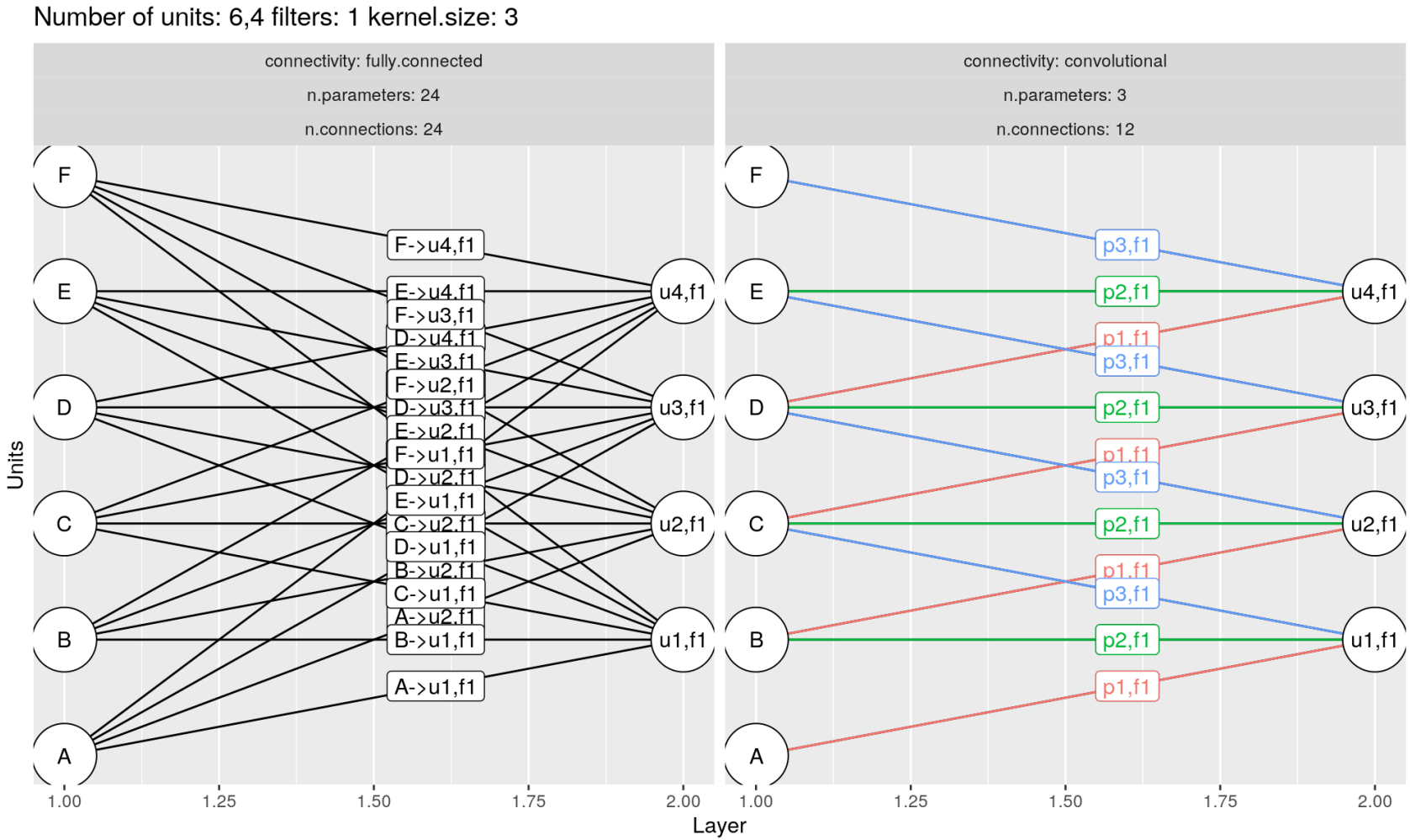
Multiple kernels/filters or sets of weights

- $x = [x_1, \ldots, x_D]$ is an input vector (array of D data).
- $w = \begin{bmatrix} w_{1,1} & \cdots & w_{1,P} \\ \vdots & \ddots & \vdots \\ w_{K,1} & \cdots & w_{K,P} \end{bmatrix}$ is a matrix of K kernels, each row is an array of P parameters / weights to learn, $P < D$.
- $h = \begin{bmatrix} h_{1,1} & \ldots & h_{1,U} \\ \vdots & \ddots & \vdots \\ h_{K,1} & \cdots & h_{K,U} \end{bmatrix}$ is an output matrix of hidden units.
  Each row is computing by applying a kernel to the input:

$$\forall u \in \{1, \ldots, U\}, \forall k \in \{1, \ldots, K\}, h_{k,u} = \sum_{p=1}^{P} w_{k,p} x_{u+p}$$

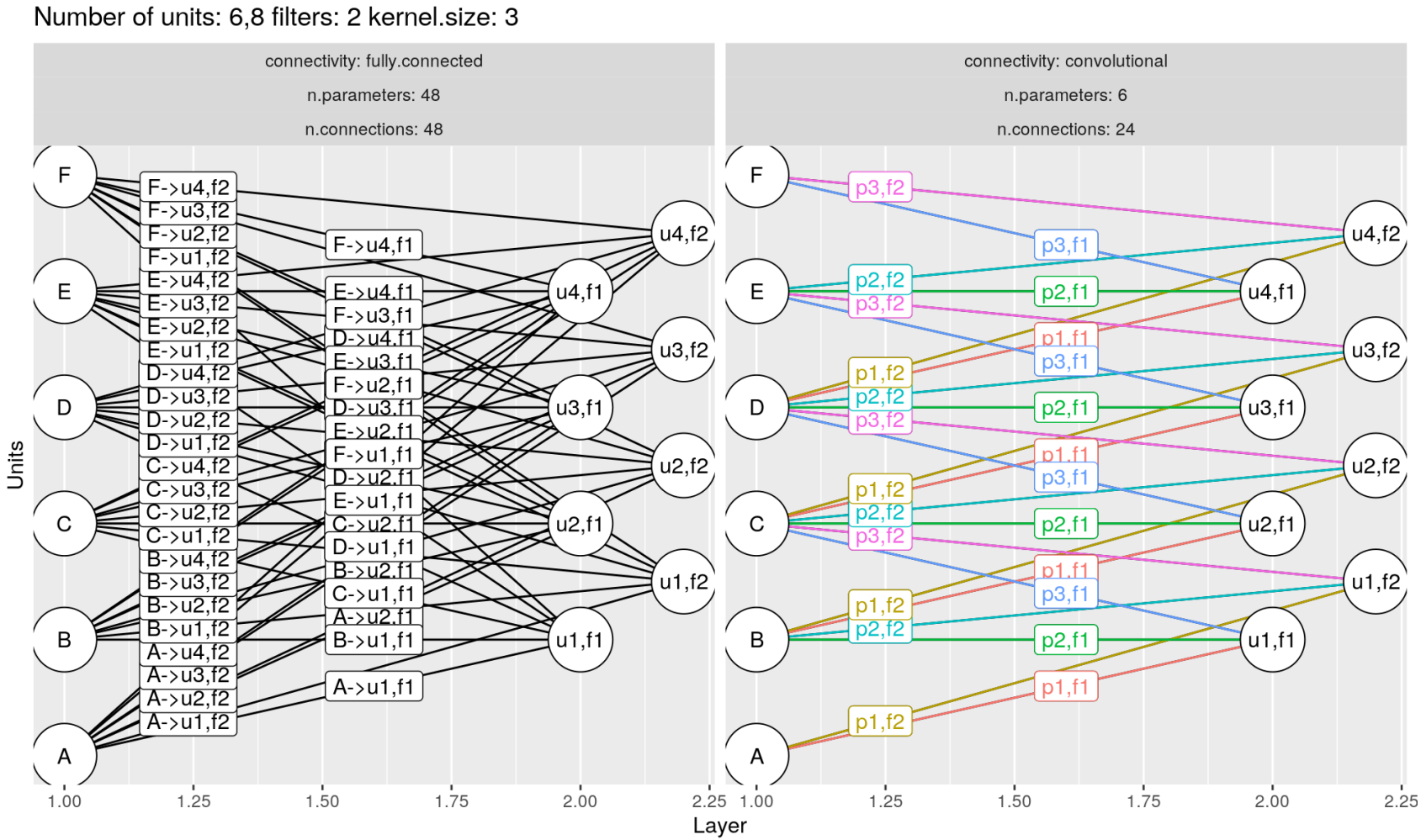
- EX in previous slide: $D = 3$ inputs, $P = 2$ parameters per kernel, $K = 2$ kernels $\Rightarrow U = 2$ output units per kernel, 4 output units total.

A more complex example (one filter)



Abbreviations: p=parameter, f=filter, u=unit. e.g., p2,f1=parameter 2 of filter 1. u2,f1=unit 2 using filter 1.

A more complex example (two filters)



Abbreviations: p=parameter, f=filter, u=unit. e.g., p2,f1=parameter 2 of filter 1. u2,f1=unit 2 using filter 1.