Introduction to machine learning and neural networks

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

March 3, 2021

Fully connected multi-layer Neural Networks

Forward and Back-propagation algorithms

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,
  $f(x) = f^{(L)}[\cdots f^{(1)}[x]] \in \mathbb{R}$.

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

$$\sigma(z) = \begin{cases} z & \text{if } z > 0, \\ 0 & \text{else.} \end{cases} \qquad\qquad \sigma(z) = 1/(1 + \exp(-z))$$

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.
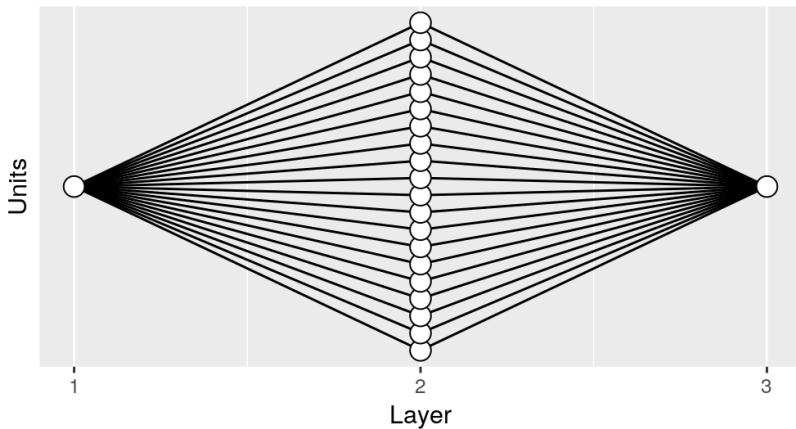
Number of units: 1,2,1
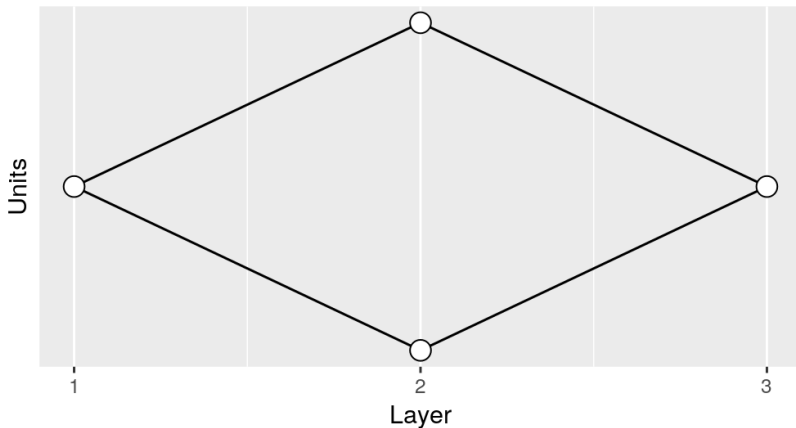
torch code

```python
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
```
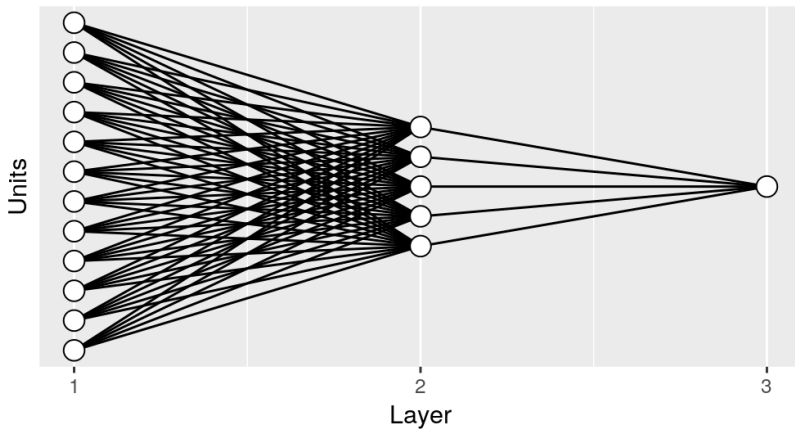
Network diagrams

Number of units: 1,20,1

Network diagrams

Number of units: 12,5,1
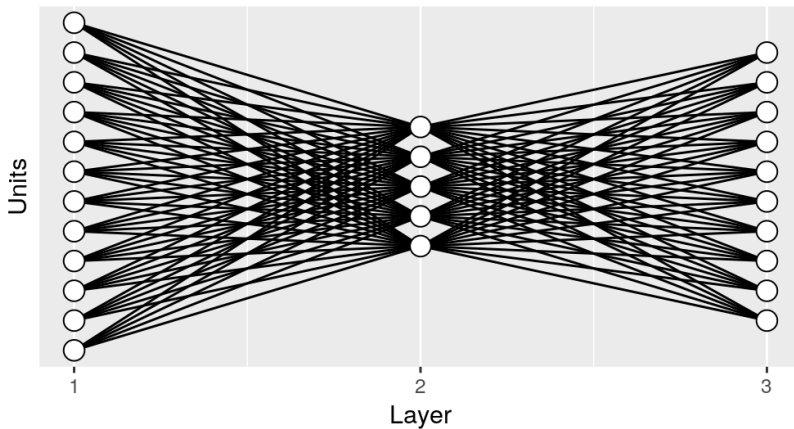
torch code

```python
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
```
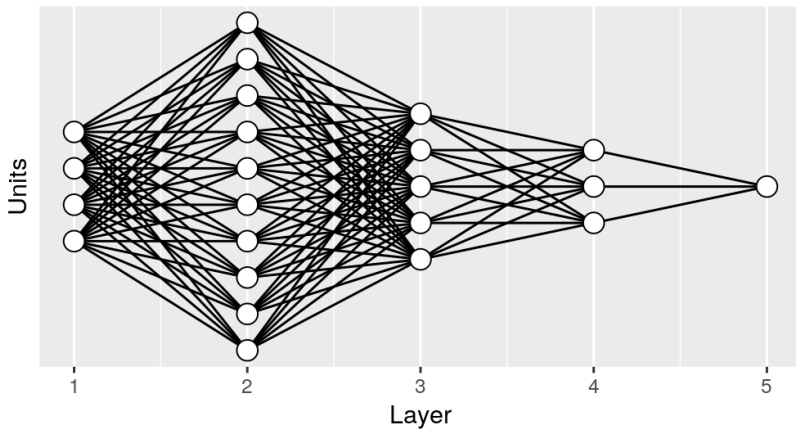
Network diagrams

Number of units: 12,5,10

Network diagrams

Number of units: 4,10,5,3,1

Units in each layer

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

- $h^{(0)} = x \in \mathbb{R}^n$ 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:

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

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 $n = 10$
inputs/features with one hidden layer with $u = 100$ units?

Fully connected multi-layer Neural Networks

Forward and Back-propagation algorithms

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.

Basic 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,

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

- But how to efficiently compute the gradient?

Forward propagation

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

```python
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