Neural network architecture and learning

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

February 16, 2022

Fully connected multi-layer Neural Networks

Computing gradients and learning weights

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 (or multi-layer perceptron) with $L - 1$ hidden layers, the function f is defined using composition of L functions, $f(x) = f^{(L)}[\cdots f^{(1)}[x]] \in \mathbb{R}$.
- Linear model is special case with $L = 1$ function, 0 hidden layers.
- "Deep" learning means $L \geq 3$ functions, at least 2 hidden layers.

Each function is matrix multiplication and activation

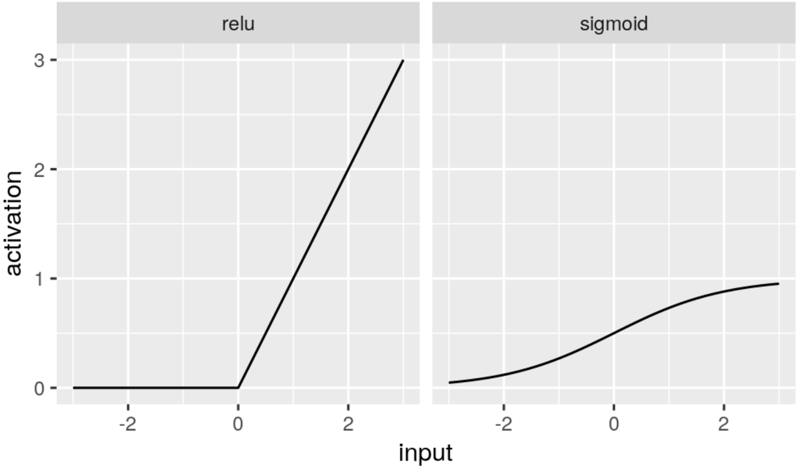
- Prediction function $f(x) = f^{(L)}[\cdots f^{(1)}[x]] \in \mathbb{R}$.
- Each function $l \in \{1, \ldots, L\}$ is a matrix multiplication followed by an activation function: $f^{(l)}[z] = \sigma^{(l)}[W^{(l)}z]$ where $W^{(l)} \in \mathbb{R}^{u^{(l)} \times u^{(l-1)}}$ is a weight matrix to learn, and $z \in \mathbb{R}^{u^{(l-1)}}$ is the input vector to that layer.
- If the loss function is defined in terms of a real-valued predicted score (typical, like we did in linear models), then the last activation function 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

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

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



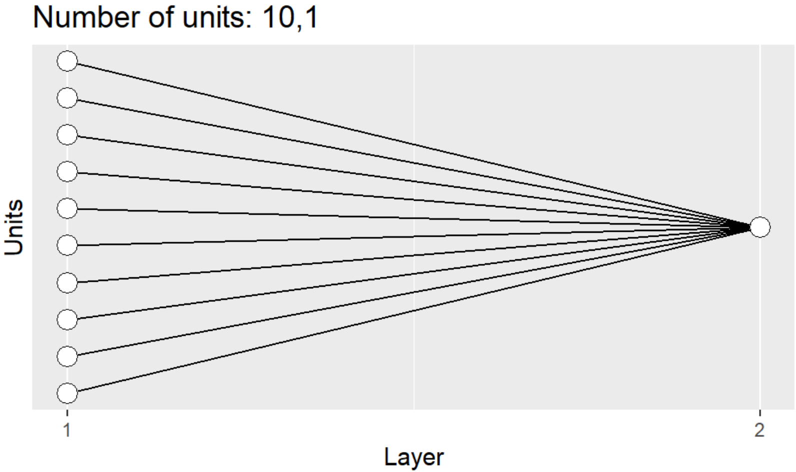
Network size

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 $W^{(l)}$).

- "Units" is a synonym for "features" and "variables."
- First and last layer are "visible" others are "hidden."
- 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).
- No hidden layers/units means $L = 1$, linear model.
- "Deep" learning means $L \geq 3$ functions, at least 2 hidden layers.

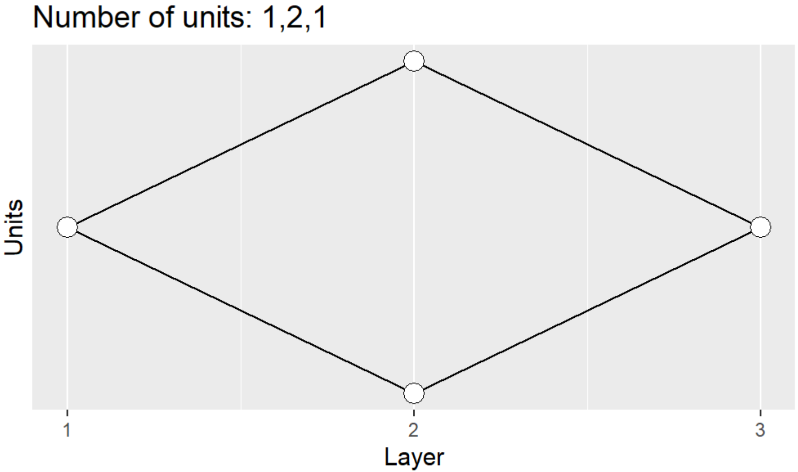
Network diagram for linear model with 10 inputs/features

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



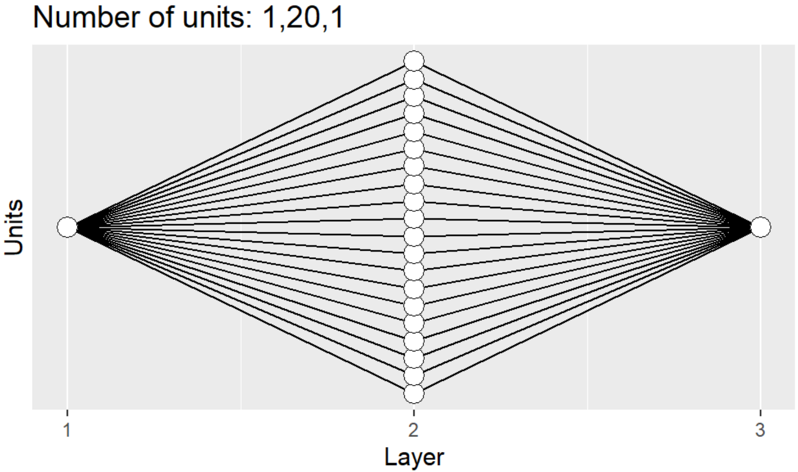
Network diagram for single hidden layer with 2 units

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



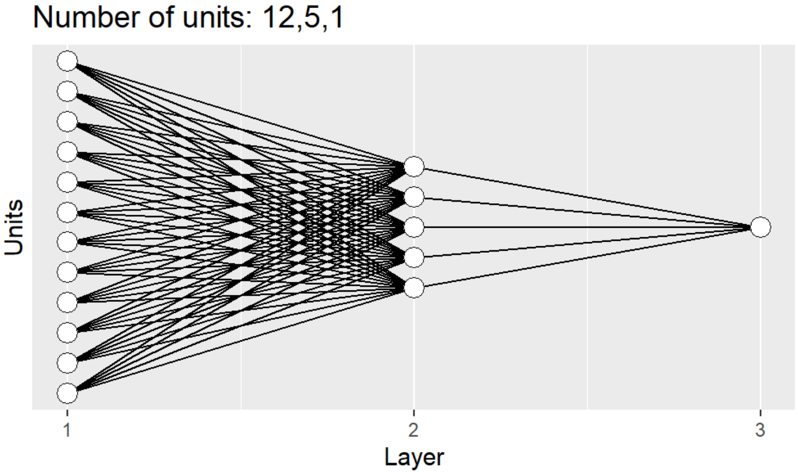
Network diagrams

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



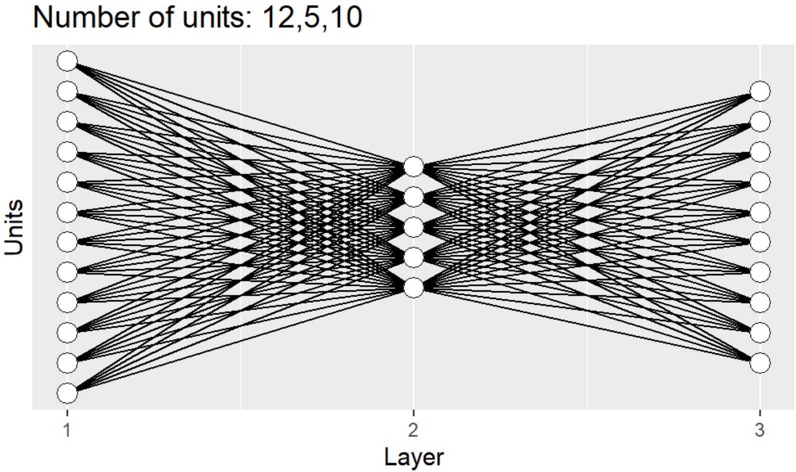
Network diagrams

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



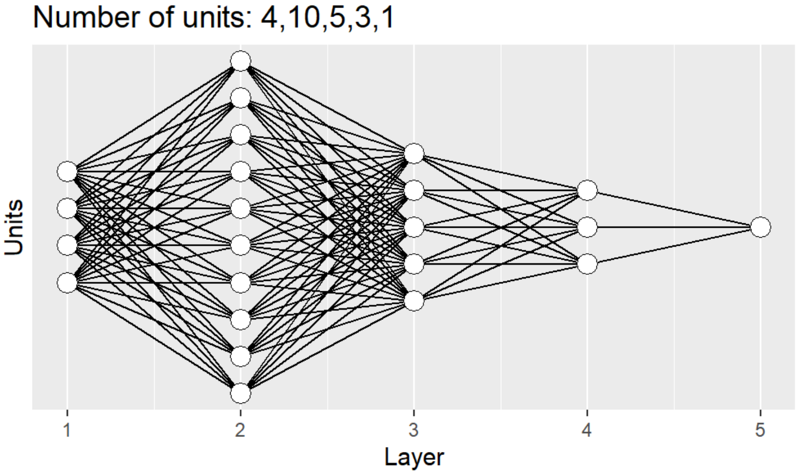
Network diagrams

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



Network diagrams

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



Units in each layer

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

Gradient descent learning

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

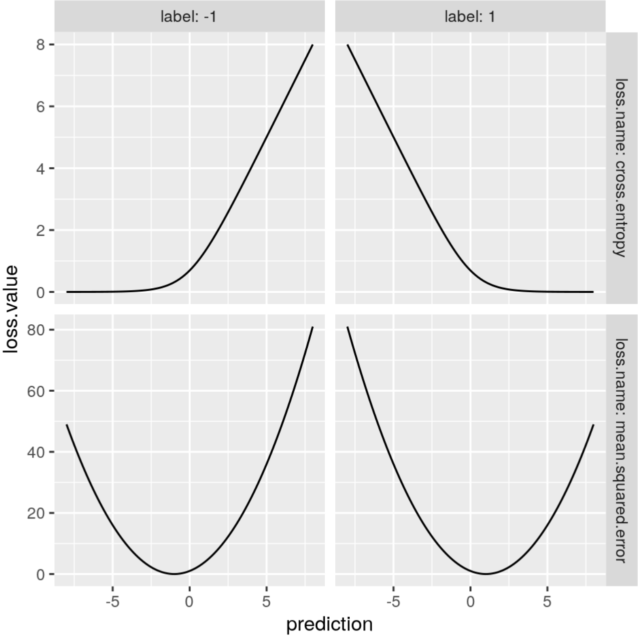
- Need to define a loss function $\mathcal{L}(\mathbf{W})$ which is differentiable, and takes small values for good predictions.
- Typically for regression we use the mean squared error, and for binary classification we use the mean logistic loss (sometimes called cross entropy).
- The mean loss $\mathcal{L}(\mathbf{W})$ is averaged over all N observations or batches i:

$$\mathcal{L}(\mathbf{W}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\mathbf{W}, \mathbf{x}_i, \mathbf{y}_i)$$

- The mean full gradient $\nabla \mathcal{L}(\mathbf{W})$ is a function which tells us the local direction where the loss is most increasing:

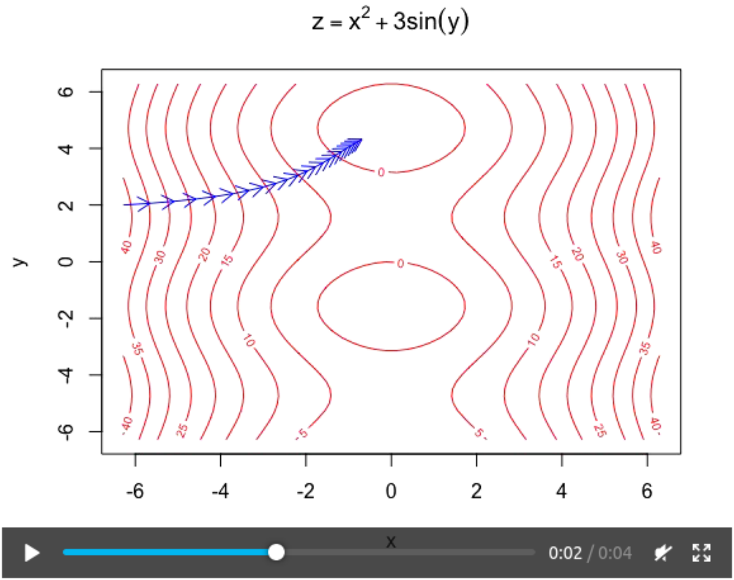
$$\nabla \mathcal{L}(\mathbf{W}) = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{L}(\mathbf{W}, \mathbf{x}_i, \mathbf{y}_i)$$

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 $\alpha > 0$ in the opposite direction of the mean full gradient,

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

- This is the **full** gradient method (same as we did for linear models): batch size $= n =$ subtrain set size, so 1 step per epoch/iteration.

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.

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 mean 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.

Forward propagation

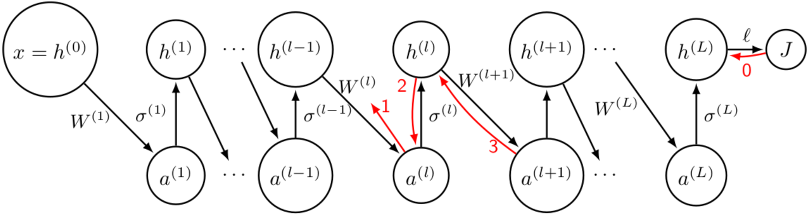
- Forward propagation is the computation of hidden units $h^{(1)}, \dots, h^{(L)}$ given the inputs x and current parameters $W^{(1)}, \dots, W^{(L)}$.
- Start from input, apply weights and activation in each layer until predicted output is computed.
- In the code this should be a for loop from first to last layer.

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.
- Deep learning libraries like torch/keras do this using automatic differentiation based on your definition of the forward method and the loss function.
- This week in class we will code the gradient computation from scratch to see how it works.
- In the code this should be a for loop from last layer to first layer.

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 $l \in \{1, \ldots, L\}$

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

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

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

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

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

Computation exercises (gradient descent learning)

Now assume we have used backpropagation to compute gradients with respect to four observations i:

$$\nabla_{\mathbf{v}} \mathcal{L}(\mathbf{v}, \mathbf{X}_i, \mathbf{y}_i) \quad = \quad \begin{cases} [-1, 1] & i = 1 \\ [-2, 2] & i = 2 \\ [-3, 2] & i = 3 \\ [-1, 2] & i = 4 \end{cases}$$

Starting at current weights $\mathbf{v} = [-2, 1]$ and using gradient descent with step size $\alpha = 0.5$, (show your work!)

1. For the full gradient method, there is one step. What is the new weight vector $\mathbf{v}$ after that step?
2. For a batch size of 2, there are two steps. Assume batch 1 is observations $i = 1, 2$ and batch 2 is observations $i = 3, 4$. What is the new weight vector $\mathbf{v}$ after the batch 1 step? After the batch 2 step?
3. For the stochastic gradient method, there are four steps $i = 1, 2, 3, 4$. What is $\mathbf{v}$ after each of those steps?