

# Neural network architecture and learning

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

October 13, 2023

# Fully connected multi-layer Neural Networks

Computing gradients and learning weights

Automatic differentiation

Multi-class classification

Torch

# 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, spam filtering, images of digits/clothing, etc).
- ▶ 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,  $f(x) = f^{(L)}[\dots f^{(1)}[x]] \in \mathbb{R}$ .
- ▶ 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  $f(x) = f^{(L)}[\dots f^{(1)}[x]] \in \mathbb{R}$ .
- ▶ Each function  $l \in \{1, \dots, L\}$  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  $(u^{(0)} = d, u^{(1)}, \dots, u^{(L-1)}, u^{(L)} = 1)$ , where  $u^{(1)}, \dots, u^{(L-1)} \in \mathbb{Z}_+$  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  $u^{(1)}, \dots, u^{(L-1)}$  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.

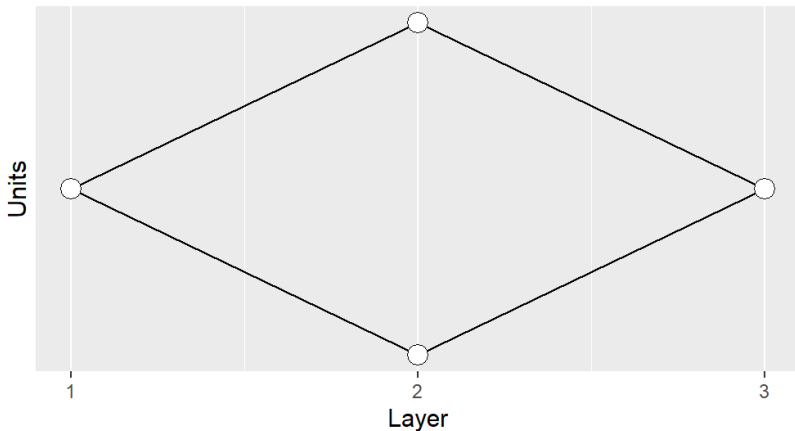
Number of units: 10,1



## 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.

Number of units: 1,2,1

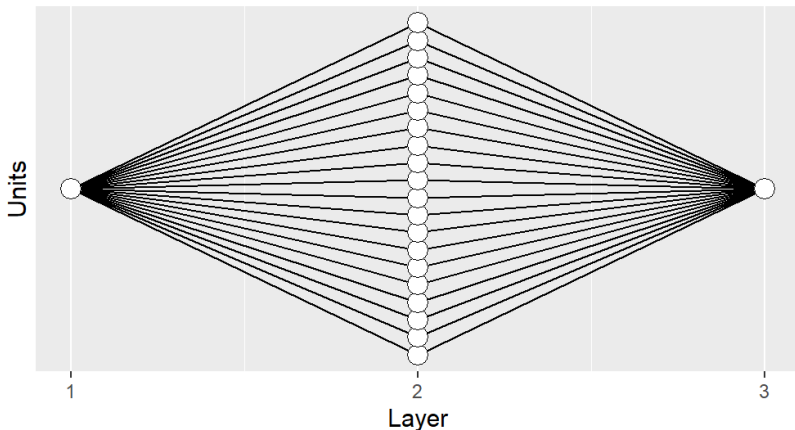




# Network diagrams

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.

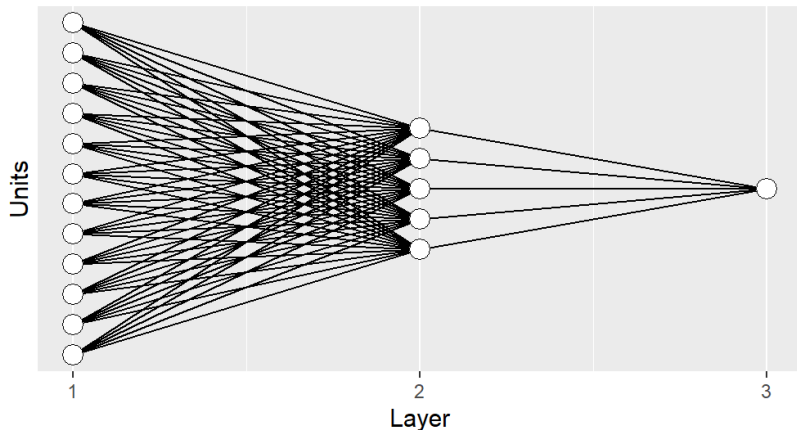
Number of units: 1,20,1



## 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.

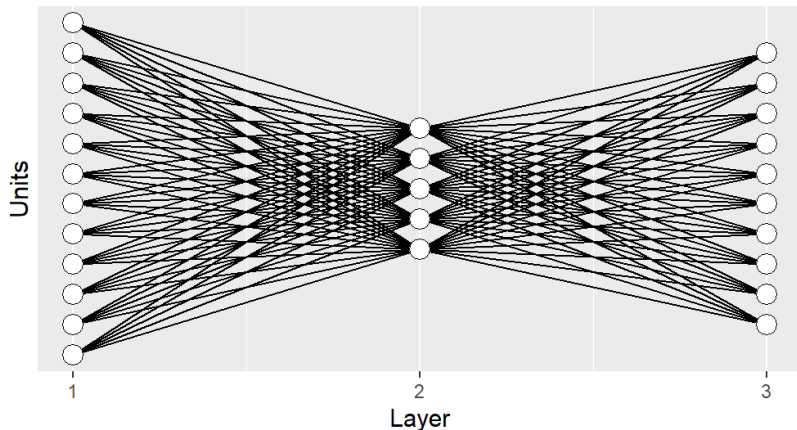
Number of units: 12,5,1



# 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.

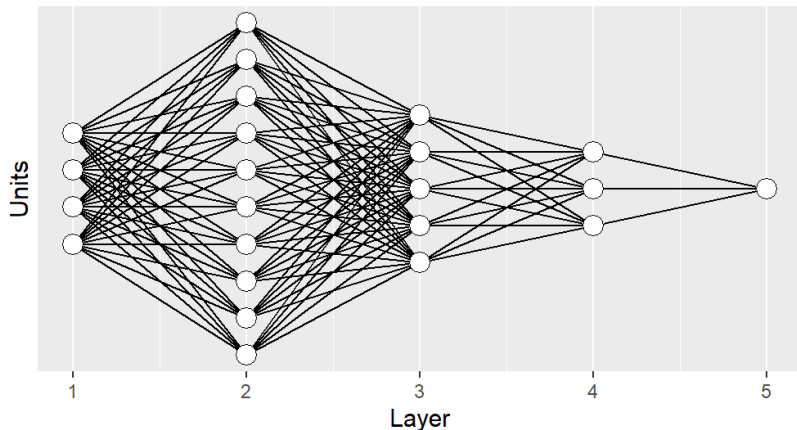
Number of units: 12,5,10



# 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.

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



# Units in each layer

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

Automatic differentiation

Multi-class classification

Torch

# 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/>

$$z = x^2 + 3\sin(y)$$



# 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.
- ▶ An **epoch** is the time/iterations to go through the subtrain set (computing gradient for each observation once).

# 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

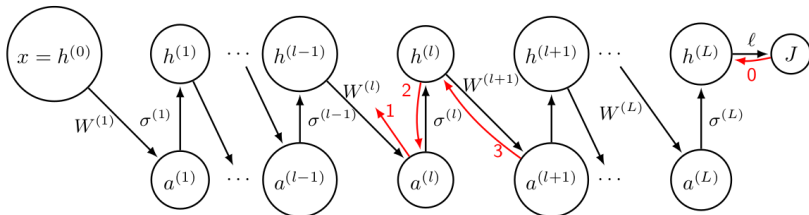
- ▶ Forward propagation is the computation of hidden units  $h^{(1)}, \dots, h^{(L)}$  given the inputs  $x$  and current parameters  $W^{(1)}, \dots, W^{(L)}$ .
- ▶ Start from input, apply weights and activation in each layer until predicted output is computed.
- ▶ In the code this should be a for loop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

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

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  $\ell$ .

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

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

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

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

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

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



# Implementation details

- ▶ Previous slides explained computations for a single observation, here we explain for a batch.
- ▶ Each  $h^{(l)}$ ,  $a^{(l)}$  and their gradients can be stored as a matrix (nrow=batch size, ncol= $u^{(l)}$ =number units in this layer).
- ▶ Each  $W^{(l)}$  and its gradient is a  $u^{(l-1)} \times u^{(l)}$  matrix.
- ▶ You may want to code assertions to make sure each matrix is the correct shape.
- ▶ Matrix multiply features by weights to get next layer,  $a^{(l)} = h^{(l-1)} W^{(l)}$ .
- ▶ Use np.where to implement relu activation (output is non-negative).
- ▶ Make sure last activation is identity — final predicted values should be real numbers (both positive and negative).

## Computation exercises (gradient descent learning)

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

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

Starting at current weights  $\mathbf{v} = [-2, 1]$  and using gradient descent with step size  $\alpha = 0.5$ , ( $\mathcal{L}$  is total loss, 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?

Fully connected multi-layer Neural Networks

Computing gradients and learning weights

Automatic differentiation

Multi-class classification

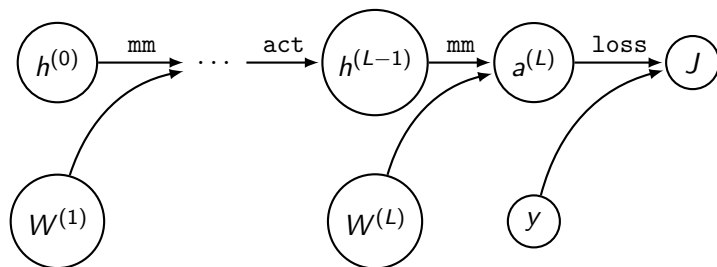
Torch

# Why automatic differentiation?

- ▶ Also called “auto-grad,” short for automatic gradient.
- ▶ People who design new neural network architectures and loss functions (fit method of learner class) do not necessarily have the expertise to compute the gradients.
- ▶ Automatic differentiation allows “separation of concerns.”
- ▶ People who know how to compute gradients can implement classes which encapsulate forward/backward computations for individual operations (matrix multiplication, log, exp, etc).
- ▶ Other people can use these classes to implement their neural network, without having to know about the details of the forward/backward computations (and no worries about coding buggy/incorrect gradients).

# Computation graph for multi-layer perceptron

- ▶ Each node in the computation graph is a tensor (0d=scalar, 1d=vector, 2d=matrix, etc).
- ▶ Each edge in the computation graph is an operation (with methods for forward/back-prop).
- ▶ Only three operations needed: matrix multiply (`mm`), non-linear activation (`act`), and computing loss given labels  $y$  and predicted scores  $a^{(L)}$ .

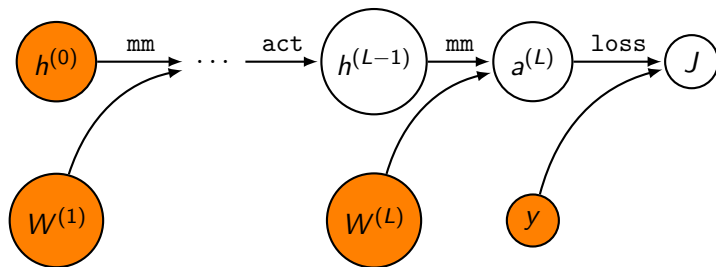


# Nodes in the computation graph

- ▶ Each node in the computation graph can be represented by an instance of a python class.
- ▶ `value` attribute is a numpy array, result of forward propagation, computed during instantiation.
- ▶ `grad` attribute is a numpy array of same size, gradient of loss with respect to this node, result of back-propagation.
- ▶ Main idea is to link these instances together to form a computation graph and `value` (forward pass), then recursively compute `grad` (backward pass).

# Initial nodes in the computation graph

- ▶ Initial node in the computation graph can be represented by an instance of `InitialNode` class.
- ▶ value attribute is a numpy array, stored on instantiation.
- ▶ Main uses are wrapping training data and neural network weights: `InitialNode(weight_mat)`, `InitialNode(feature_mat)`, `InitialNode(label_vec)`.



# Derived nodes in the computation graph

- ▶ `Operation` class represents a node in the computation graph which is computed using other nodes.
- ▶ On instantiation, stores input nodes and does forward propagation.
- ▶ Method `backward()` computes gradient and recursively calls `backward()` on input nodes.
- ▶ `Operation` is virtual so we only instantiate sub-classes:  
`mm(X_node, W_node)`, `relu(A_node)`,  
`logistic_loss(A_node, y_node)`.
- ▶ Sub-classes should define `forward` and `gradient` methods which implement details of forward/back-prop, results are stored as `value/grad` attributes.
- ▶ Don't get confused! Parent class (sub-classes of `Operation` inherit methods/attributes) not the same as parent node (child node is an instance of an `Operation` sub-class, and has parent nodes as attributes).

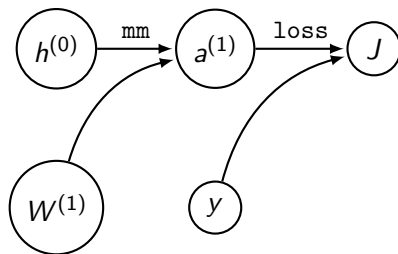


# Functions and gradients

- ▶  $J = \text{logistic\_loss}(A, y) \in \mathbb{R}^{b \times 1}$  is same as in linear models. Values are  $J_i = \log[1 + \exp(-y_i A_i)]$  and gradient is  $(\nabla_A J)_i = -y_i / (1 + \exp(y_i A_i))$ .
- ▶  $A = \text{mm}(X, W) = XW \in \mathbb{R}^{b \times u}$  where  $X$  is a  $b \times p$  matrix ( $b$  = number of samples in batch,  $p$  = number of units/features in this layer), and  $W$  is a  $p \times u$  matrix ( $u$  = number of units/features in next layer). Gradient of linear function is constant:  $\nabla_X J = (\nabla_A J)(W^T)$ ,  $\nabla_W J = X^T (\nabla_A J)$ .
- ▶  $H = \text{relu}(A) \in \mathbb{R}^{b \times u}$  where each element  $H_i = A_i$  if  $A_i > 0$  else 0. Gradient  $\nabla_A J = \nabla_A H \nabla_H J$  is piecewise constant,  $(\nabla_A J)_i = (\nabla_H J)_i$  if  $A_i > 0$  else 0.

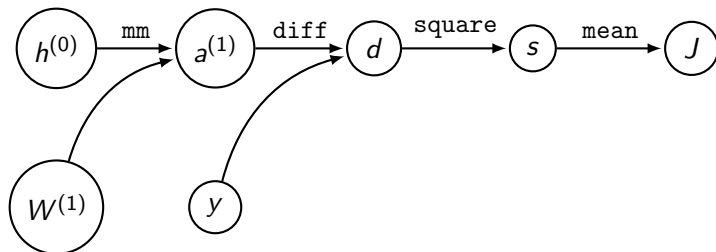
# Simple computation graph for linear model

- ▶ Only two operations needed: matrix multiply (`mm`) and computing loss given predicted scores (`loss`).
- ▶ To implement the `loss` operation/class, you need to know how to compute the gradient of the loss (maybe difficult for complex loss functions).



# Detailed computation graph for linear model for regression

- ▶ Example: square loss  $\ell(a^{(1)}, y) = (a^{(1)} - y)^2 = d^2$ ,
- ▶ Mean squared error:  $J = \sum_{i=1}^n \ell(a_i^{(1)}, y_i) / n = \sum_{i=1}^n s_i / n$ .
- ▶ More operations needed: matrix multiply (`mm`), subtraction (`diff`), square, mean.
- ▶ Each operation has a simple gradient (demo).



## Possible exam questions

- ▶ Given a computation graph, and values for initial nodes, compute value in each derived node by hand (forward propagation), then compute grad (back-prop).
- ▶ In linear models we used scaling inside of the fit method — make sure each feature/column has mean=0 and sd=1 before running gradient descent. What would you have to change in your neural network class to use scaling?
- ▶ In linear models we used gradient descent to learn an intercept parameter (a constant added to each real-valued prediction). We discussed two methods: (1) adding a column of ones to the feature matrix and an entry to the weight vector, or (2) adding a separate node representing the intercept in the computation graph. How would you modify your neural network class, using each method? (each hidden/output unit/feature in the neural network should have its own intercept parameter)

Fully connected multi-layer Neural Networks

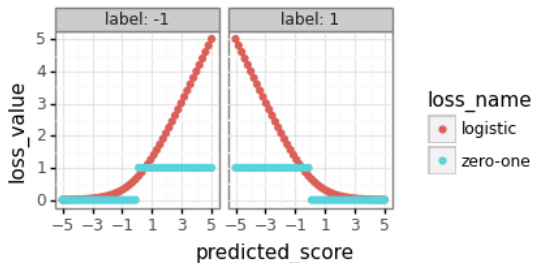
Computing gradients and learning weights

Automatic differentiation

Multi-class classification

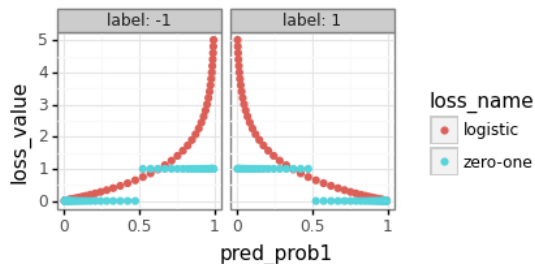
Torch

# Review of logistic loss for binary classification



- ▶ For binary classification we have labels  $\underline{y} \in \{-1, 1\}$  (underline for label coded as -1 or 1).
- ▶ Let  $f(x) = \hat{y} \in \mathbb{R}$  be the predicted score (output of neural network).
- ▶ The logistic loss is  $\ell(\hat{y}, \underline{y}) = \log[1 + \exp(\hat{y}\underline{y})]$ .
- ▶ Minimizing the logistic loss is equivalent to maximizing the binomial likelihood (sometimes called Bernoulli, special case when there is 1 trial), ex: flipping coins or rolling dice.
- ▶ The binomial distribution (two classes) is a special case of the multinomial distribution (two or more classes).

# Probabilistic interpretation of logistic loss



- ▶ Let  $y \in \{0, 1\}$  be the binary class label (non-negative integer).
- ▶ Let  $\hat{p}_1 = 1/(1 + \exp(-\hat{y})) \in (0, 1)$  be the predicted probability of class 1 (positive), output by the neural network.
- ▶ Then  $\hat{p}_0 = 1 - \hat{p}_1$  is the predicted probability of negative class.
- ▶ Then the logistic loss is the negative log probability/likelihood of the label class,

$$\begin{aligned}\ell(\hat{y}, y) &= \begin{cases} -\log[\hat{p}_1] & \text{if label is positive, } y = 1 \\ -\log[\hat{p}_0] & \text{if label is negative, } y = 0 \end{cases} \\ &= -\log[\hat{p}_y].\end{aligned}$$

# Multi-class loss and softmax function

- ▶ We assume there are  $K \geq 2$  classes of labels,  $y \in \{0, 1, \dots, K-1\}$ .
- ▶ The neural network can output a probability for each class,  $\hat{p} = [\hat{p}_0, \hat{p}_1, \dots, \hat{p}_{K-1}]$ , with  $\hat{p}_j \in [0, 1]$  and  $\sum_j \hat{p}_j = 1$ .
- ▶ The loss function to minimize via gradient descent is the negative log likelihood of the label class,  $\ell[\hat{p}, y] = -\log \hat{p}_y$ .
- ▶ The neural network last linear layer outputs  $K$  units/features (real numbers from matrix multiplication, possibly negative), one for each class:  $\hat{y}_0, \hat{y}_1, \dots, \hat{y}_{K-1} \in \mathbb{R}$ .
- ▶ To convert to a vector of probability values, we use the softmax function,

$$\hat{p} = \text{softmax}(\hat{y}_0, \dots, \hat{y}_{K-1}) = \left[ \frac{\exp \hat{y}_0}{\sum_{j=0}^{K-1} \exp \hat{y}_j}, \dots, \frac{\exp \hat{y}_{K-1}}{\sum_{j=0}^{K-1} \exp \hat{y}_j} \right].$$

- ▶ (exp converts to positive, division normalizes to one)



# Visualization of loss functions for 3 class problem



- ▶ Each point in 3-simplex is a probability triple  $(p_0, p_1, p_2)$ .
- ▶ Corners: max prob. for one class  $(1,0,0)$  or  $(0,1,0)$  or  $(0,0,1)$ .
- ▶ Midpoint when classes have equal probability  $(1/3, 1/3, 1/3)$ .
- ▶ Zero-one loss = 0 when label class most probable.

# Note about over-parameterization

- ▶ A neural network for  $K$  classes which outputs  $K$  features is actually over-parameterized (but easy to code).
- ▶ For  $K$  classes, we only need  $K - 1$  outputs (fewer parameters in last weight matrix, more computationally efficient).
- ▶ For example, binary classification,  $K = 2$ , we only need 1 output (higher values mean more likely positive class).
- ▶ For example,  $K = 3$  classes,  $f(x) = [\hat{p}_1, \hat{p}_2]$  only needs parameters for two outputs, and the third class probability can be derived from the other two,  $\hat{p}_0 = 1 - \hat{p}_1 - \hat{p}_2$ .

## Predicted probability and loss

- ▶ Let  $z \in \mathbb{R}^K$  be a vector of real-valued predictions (inputs to softmax), same as neural network outputs if over-parameterized with  $K$  outputs  $\hat{y}_0, \hat{y}_1, \dots, \hat{y}_{K-1}$ .
- ▶ Otherwise if neural network only has  $K - 1$  outputs  $\hat{y}_1, \dots, \hat{y}_{K-1}$  then  $z = [0, \hat{y}_1, \dots, \hat{y}_{K-1}]$ .
- ▶ Then the predicted probability vector is  $\hat{p} = \text{softmax}(z_0, \dots, z_{K-1})$ .
- ▶ The loss function is the negative log probability of the label class,

$$-\log \hat{p}_y = -\log \left[ \frac{\exp(z_y)}{\sum_{j=0}^{K-1} \exp(z_j)} \right] = \log \left[ \sum_{j=0}^{K-1} \exp(z_j) \right] - z_y.$$

# Binary classification is a special case 1

- ▶ For label  $y$  and real-valued predictions  $z_0, \dots, z_{K-1}$ , loss is  $\log \left[ \sum_{j=0}^{K-1} \exp(z_j) \right] - z_y$ .
- ▶ In binary classification,  $K = 2$  classes, the neural network outputs a single score  $\hat{y} \in \mathbb{R}$ , larger values mean more likely to be positive class.
- ▶ Let  $z = [0, \hat{y}]$ , then  $\sum_{j=0}^{K-1} \exp(z_j) = 1 + \exp(\hat{y})$ .
- ▶ If label is positive,  $y = 1$ , then  $z_y = \hat{y}$  and loss is

$$\log [1 + \exp(\hat{y})] - \hat{y} = \log \left[ \frac{1 + \exp(\hat{y})}{\exp(\hat{y})} \right] = \log[1 + \exp(-\hat{y})].$$

- ▶ If label is negative,  $y = 0$ , then  $z_y = 0$  and loss is  $\log [1 + \exp(\hat{y})]$ .
- ▶ Both cases agree with our previous logistic loss formula,  $\log[1 + \exp(-\underline{y}\hat{y})]$ , with  $\underline{y} \in \{-1, 1\}$ .

## Note about numerical stability

- ▶ For label  $y$  and real-valued predictions  $z_0, \dots, z_{K-1}$ , loss is

$$\log \left[ \sum_{j=0}^{K-1} \exp(z_j) \right] - z_y = L - z_y.$$

- ▶  $\exp$  may overflow to  $\infty$  if  $z_j$  values are too large (bad).
- ▶ Numerical issues can be limited by using the log-sum-exp trick: let  $M = \max_j z_j$ , then

$$L = \log \left[ \sum_{j=0}^{K-1} \exp(z_j) \right] = M + \log \left[ \sum_{j=0}^{K-1} \exp(z_j - M) \right].$$

# Gradient computation

- ▶ For label  $y$  and real-valued predictions  $z_0, \dots, z_{K-1}$ , loss is  $\log \left[ \sum_{j=0}^{K-1} \exp(z_j) \right] - z_y = L - z_y$ .
- ▶ Partial derivative with respect to prediction for class  $k \in \{0, \dots, K-1\}$  can be written in terms of indicator function  $I$  (1 if argument is true, otherwise 0):

$$\frac{\partial}{\partial z_k} (L - z_y) = \frac{\exp(z_k)}{\sum_{j=0}^{K-1} \exp(z_j)} - I[y = k].$$

- ▶ We compute  $L$  during forward pass to get loss and then use it again in backward pass, since  $\exp(L) = \sum_{j=0}^{K-1} \exp(z_j)$ ,

$$\frac{\partial}{\partial z_k} (L - z_y) = \frac{\exp(z_k)}{\exp(L)} - I[y = k] = \underbrace{\exp(z_k - L)}_{\hat{p}_k} - I[y = k].$$

## Binary classification is a special case 2

- ▶ For label  $y$  and real-valued predictions  $z_0, \dots, z_{K-1}$ , partial derivative for class  $k \in \{0, \dots, K-1\}$  is

$$\frac{\partial}{\partial z_k} (L - z_y) = \frac{\exp(z_k)}{\sum_{j=0}^{K-1} \exp(z_j)} - I[y = k].$$

- ▶ In binary classification,  $K = 2$  classes, the neural network outputs a single score  $\hat{y} \in \mathbb{R}$ , larger values mean more likely to be positive class.
- ▶ Let  $z = [0, \hat{y}]$ , then  $\sum_{j=0}^{K-1} \exp(z_j) = 1 + \exp(\hat{y})$ .
- ▶ Partial derivative with respect to predictions  $\hat{y}$  is

$$\frac{\partial}{\partial \hat{y}} (L - z_y) = \frac{\exp(\hat{y})}{1 + \exp(\hat{y})} - I[y = 1] = \frac{1}{1 + \exp(-\hat{y})} - I[y = 1].$$

- ▶ If label is positive,  $y = 1$ , then gradient is  $-1/[1 + \exp(\hat{y})]$ .
- ▶ If label is negative,  $y = 0$ , then gradient is  $1/[1 + \exp(-\hat{y})]$ .
- ▶ Both cases agree with our previous logistic loss gradient formula,  $(-\underline{y})/[1 + \exp(\underline{y}\hat{y})]$ , with  $\underline{y} \in \{-1, 1\}$ .

## Gradient example, very bad prediction

label=9

	pred_prob	neg_grad
0	0.0	-0.0
1	0.0	-0.0
2	0.0	-0.0
3	0.0	-0.0
4	0.0	-0.0
5	0.0	-0.0
6	1.0	-1.0
7	0.0	-0.0
8	0.0	-0.0
9	0.0	1.0

Predicted 100% class 6, label class 9, gradient says to decrease prob for class 6 and increase prob for class 9.



## Gradient example, very bad prediction

```
label=9
  pred_prob neg_grad
0         0.0    -0.0
1         0.0    -0.0
2         0.0    -0.0
3         0.7    -0.7
4         0.0    -0.0
5         0.0    -0.0
6         0.3    -0.3
7         0.0    -0.0
8         0.0    -0.0
9         0.0     1.0
```

Predicted 70% class 3, 30% class 6, label class 9, gradient says to decrease prob for classes 3 and 6 (but by different amounts), and increase prob for class 9.

## Gradient example, bad prediction

label=3

	pred_prob	neg_grad
0	0.0	-0.0
1	0.0	-0.0
2	0.0	-0.0
3	0.1	0.9
4	0.0	-0.0
5	0.0	-0.0
6	0.9	-0.9
7	0.0	-0.0
8	0.0	-0.0
9	0.0	-0.0

Predicted 10% class 3 and 90% class 6, label class 3, gradient says to increase prob for class 3 and decrease prob for class 6 (note gradient smaller than for predicting 0% label class).

## Gradient example, decent but incorrect prediction

label=3

	pred_prob	neg_grad
0	0.0	-0.0
1	0.3	-0.3
2	0.0	-0.0
3	0.3	0.7
4	0.0	-0.0
5	0.4	-0.4
6	0.0	-0.0
7	0.0	-0.0
8	0.0	-0.0
9	0.0	-0.0

Predicted 30% class 1, 30% class 3, 40% class 5, label class 3,  
gradient says to decrease prob for class 1, increase prob for class 3,  
and decrease prob for class 5.

## Gradient example, good prediction

label=1

	pred_prob	neg_grad
0	0.0	-0.0
1	0.6	0.4
2	0.0	-0.0
3	0.0	-0.0
4	0.0	-0.0
5	0.0	-0.0
6	0.0	-0.0
7	0.4	-0.4
8	0.0	-0.0
9	0.0	-0.0

Predicted 60% class 1, 40% class 7, label class 1, gradient says to decrease prob for class 7 and increase prob for class 1.

## Gradient example, very good prediction

label=6

	pred_prob	neg_grad
0	0.0	-0.0
1	0.0	-0.0
2	0.0	-0.0
3	0.0	-0.0
4	0.0	-0.0
5	0.0	-0.0
6	1.0	0.0
7	0.0	-0.0
8	0.0	-0.0
9	0.0	-0.0

Predicted 100% class 6, label class 6, gradient zero indicates that predictions should not be updated.

Fully connected multi-layer Neural Networks

Computing gradients and learning weights

Automatic differentiation

Multi-class classification

Torch

- ▶ Machine learning library/module for python (and R, C++).
- ▶ Lots of standard neural network architectures and loss functions supported out of the box.
- ▶ Supports automatic differentiation! (easy to experiment with new models and loss functions without having to worry about gradient computations)

## torch code for linear model for binary classification

```
import torch
class LinearModel(torch.nn.Module):
    def __init__(self):
        super(LinearModel, self).__init__()
        self.weight_vec = torch.nn.Linear(n_features, 1)
    def forward(self, features):
        return self.weight_vec(features)
```

- ▶ Linear is a matrix multiplication, with optional intercept (present by default, bias=False to suppress).
- ▶ One output because we are doing binary classification (want to predict a single real-valued score).
- ▶ Linear and other objects with weight parameters to learn must be assigned to an attribute of self (so torch.optim algorithms can find weights to update).



## torch code for neural network with one hidden layer

```
class OneHiddenLayer(torch.nn.Module):
    def __init__(self, n_features, n_hidden):
        super(OneHiddenLayer, self).__init__()
        self.hidden_weights = torch.nn.Linear(
            n_features, n_hidden)
        self.activation = torch.nn.ReLU()
        self.out_weights = torch.nn.Linear(n_hidden, 1)
    def forward(self, feature_mat):
        a_mat = self.hidden_weights(feature_mat)
        h_mat = self.activation(a_mat)
        return self.out_weights(h_mat)
```

Use activation after each Linear layer except for last.

## torch code for stack of hidden layers

```
class Net(torch.nn.Module):
    def __init__(self, n_features, n_hidden):
        super(Net, self).__init__()
        self.stack = torch.nn.Sequential(
            torch.nn.Linear(n_features, n_hidden),
            torch.nn.ReLU(),
            torch.nn.Linear(n_hidden, 1))
    def forward(self, feature_mat):
        return self.stack(feature_mat)
```

## Differences between gradient descent algorithm variants

```
net = Net()
loss_fun = torch.nn.BCEWithLogitsLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.03)

# to do in a loop:
optimizer.zero_grad()
loss_value = loss_fun(net(inputs), outputs) # last node
loss_value.backward() # auto grad
optimizer.step() # learning

# stochastic, batch, full gradient descent variants:
loss_one = loss_fun(
    net(one_input), one_output)
loss_batch = loss_fun(
    net(batch_inputs), batch_outputs)
loss_subtrain = loss_fun(
    net(subtrain_inputs), subtrain_outputs)
```

## Neural network for regression: square loss

```
net = Net()
loss_fun = torch.nn.MSELoss() #The only change!
optimizer = torch.optim.SGD(net.parameters(), lr=0.03)

# to do in a loop:
optimizer.zero_grad()
loss_value = loss_fun(net(inputs), outputs)
# outputs are real numbers, not 0/1 binary classes.
loss_value.backward() # auto grad
optimizer.step() # learning
```

## Neural network for multiple classes, cross entropy loss

```
class Net(torch.nn.Module):
    def __init__(self, n_features, n_hidden, n_classes):
        super(Net, self).__init__()
        self.stack = torch.nn.Sequential(
            torch.nn.Linear(n_features, n_hidden),
            torch.nn.ReLU(),
            torch.nn.Linear(n_hidden, n_classes))
    def forward(self, feature_mat):
        return self.stack(feature_mat)

net = Net()
loss_fun = torch.nn.CrossEntropyLoss()
loss_value = loss_fun(net(inputs), outputs)
```

No need to explicitly compute predicted probability.

## Copy data from numpy and onto GPU

- ▶ Torch is very similar to numpy (tensor data types, vectorized functions and methods).
- ▶ Can easily run on GPU for speedups, but need to copy neural network weights and data to GPU memory.
- ▶ Request a GPU on Monsoon via `-gres=gpu:tesla:1` flag to `srun/sbatch`.

```
device = "cuda" if torch.cuda.is_available() else "cpu"
# features and labels to torch and GPU
cpu_features = torch.from_numpy(numpy_features).float()
dev_features = cpu_features.to(device)
cpu_labels = torch.from_numpy(numpy_labels).float()
dev_labels = cpu_labels.to(device)
# weights to GPU
cpu_net = Net()
dev_net = cpu_net.to(device)
```

## torch Dataset/DataLoader helpers for batching

```
class CSV(torch.utils.data.Dataset):
    def __init__(self, features, labels):
        self.features = features
        self.labels = labels
    def __getitem__(self, item):
        return self.features[item,:], self.labels[item]
    def __len__(self):
        return len(self.labels)

ds = CSV(feature_mat, set_labels["subtrain"])
dl = torch.utils.data.DataLoader(
    ds, batch_size=1000, shuffle=True)
for batch_features, batch_labels in dl:
    # gradient descent code here.
```

Not necessary (you can do your own batching), but can be useful.