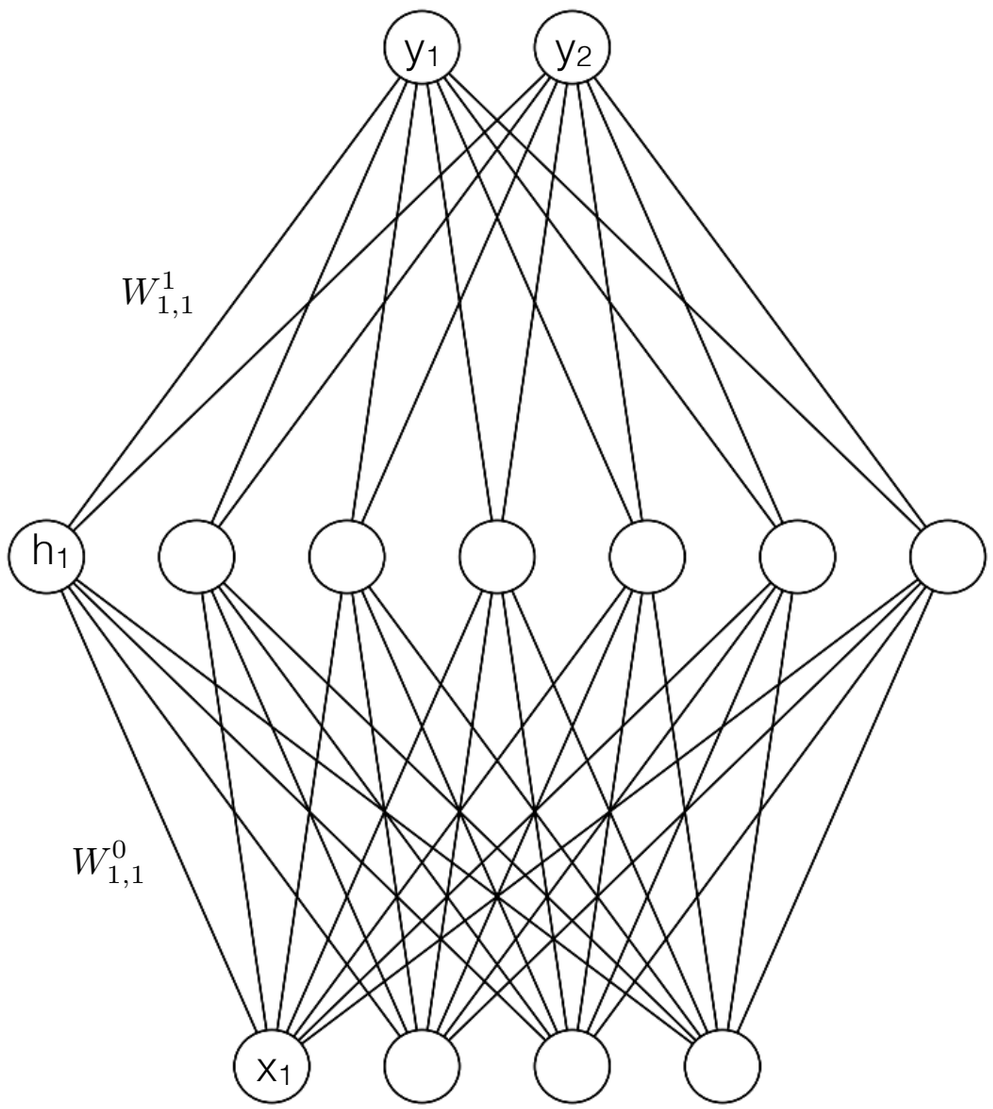
Deep Learning

How to learn DL models?

DL Models



$$L(\mathbf{w}) = \frac{1}{N}\sum_i (\mathbf{y}_i - f(\mathbf{x}_i, \mathbf{w}))^2$$

Output Layer
(y_1, y_2)

Weights W^1

$$y_i = \sigma(\textstyle\sum_{j=1}^{7} h_i * W^1_{i,j})$$

Hidden Layer
$(h_1, h_2, h_3, h_4, h_5, h_6, h_7)$

Weights W^0

$$h_i = \sigma(\textstyle\sum_{j=1}^{7} x_i * W^0_{i,j})$$

Input Layer
(x_1, x_2, x_3, x_4)

Automatic Differentiation

$$L(\mathbf{w}) = \frac{1}{N} \sum_i (\mathbf{y}_i - f(\mathbf{x}_i, \mathbf{w}))^2$$

We have seen that in order to optimize our model f we need to compute the **derivative of the loss function** with respect **to all model parameters**.

In the case of neural networks, $\mathbf{w} = (W^1, W^2, \ldots, W^m)$ and f is the composition of several layers:

$$\begin{aligned} \mathbf{h}_0 &= \mathbf{x} \\ \mathbf{h}_1 &= \sigma(W^1 \mathbf{h}_o + b_1) \\ &\cdots \\ f &= L(\sigma(W^m \mathbf{h}_{m-1} + b_m)) \end{aligned}$$

DL models can have hundreds of millions of parameters!

We need to compute: $\dfrac{\delta L}{\delta w_{ij}^p}, \dfrac{\delta L}{\delta b_j}$

Automatic Differentiation

The computation of derivatives in computer models is addressed by four main methods:

- **Manually working out derivatives and coding the result** (as in the original paper describing *backpropagation*)
- Numerical differentiation (using finite difference approximations);
- Symbolic differentiation (using expression manipulation in software, such as Sympy);
- and **Automatic Differentiation** (AD).

The output of each neuron is an non-linear function of its total input. The paper uses the logistic function

NATURE VOL. 323 9 OCTOBER 1986 — LETTERS TO NATURE — 533

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input-output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

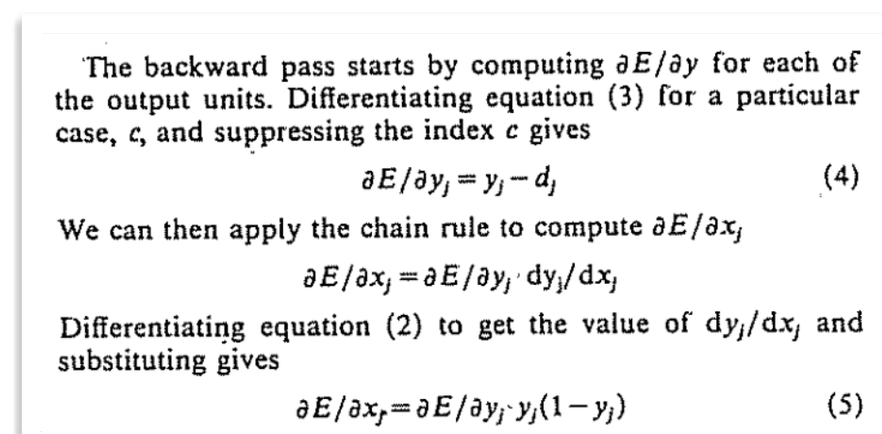
$$x_j = \sum_i y_i w_{ji} \quad (1)$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, y_j, which is a non-linear function of its total input

$$y_j = \frac{1}{1+e^{-x_j}} \quad (2)$$

† To whom correspondence should be addressed



The backward pass starts by computing $\partial E/\partial y$ for each of the output units. Differentiating equation (3) for a particular case, c, and suppressing the index c gives

$$\partial E/\partial y_j = y_j - d_j \quad (4)$$

We can then apply the chain rule to compute $\partial E/\partial x_j$

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot dy_j/dx_j$$

Differentiating equation (2) to get the value of dy_j/dx_j and substituting gives

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot y_j(1-y_j) \quad (5)$$

https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

Automatic Differentiation

Automatic differentiation (AD) works by systematically applying the **chain rule** of differential calculus at the elementary operator level.

$$f(g(x))$$

Chain Rule ↓

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

If there are more than one variable g_i in-between y and x, for example, if f is a two-dimensional function $f(g_1(x), g_2(x))$, then:

$$\frac{\partial f}{\partial x} = \sum_i \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial x}$$

AD allows the accurate evaluation of derivatives at machine precision, with only a <u>small</u> constant factor of overhead.

(FA) Automatic Differentiation

In its most basic description, AD relies on the fact that all **numerical computations are ultimately compositions of a finite set of elementary operations for which derivatives are known.**

Let's consider:

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - sin(x_2)$$

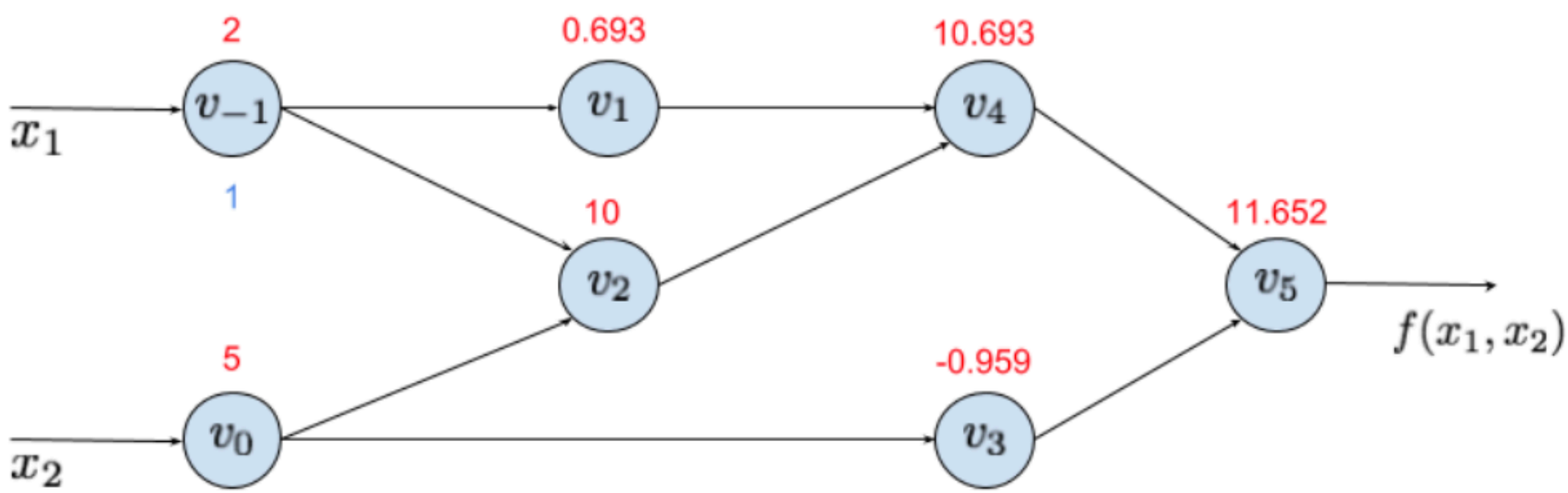
and let's write how to evaluate $f(2,5)$ via a sequence of primitive operations:

- $v_{-1} = x_1 = 2$
- $v_0 = x_2 = 5$
- $v_1 = \ln v_{-1} = \ln 2 = 0.693$
- $v_2 = v_{-1} \times v_0 = 2 \times 5 = 10$
- $v_3 = sin(v_0) = sin(5) = -0.959$
- $v_4 = v_1 + v_2 = 0.693 + 10 = 10.693$
- $v_5 = v_4 - v_3 = 10.693 + 0.959 = 11.652$
- $y = v_5 = 11.652$

This program can compute the value of $f(x)$ and also populate program variables.

(FA) Automatic Differentiation

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - sin(x_2)$$



This is the **computational graph**, which is automatically build by all language interpreters and compilers to efficiently evaluate f.

Observation:
Every node represents an elementary operation **with known analytic derivative**.

(FA) Automatic Differentiation

For computing the derivative of f with respect to x_1 we start by associating with each intermediate variable v_i a derivative: $\partial v_i = \dfrac{\partial v_i}{\partial x_1}$.

Then we apply the chain rule to each elementary operation:

- $\partial v_{-1} = \frac{\partial x_1}{\partial x_1} = 1$
- $\partial v_0 = \frac{\partial x_2}{\partial x_1} = 0$
- $\partial v_1 = \boxed{\frac{\partial \ln(v_{-1})}{\partial v_{-1}} \partial v_{-1}} = 1/2 \times 1 = 0.5$
- $\partial v_2 = \frac{\partial (v_{-1} \times v_0)}{\partial v_{-1}} \partial v_{-1} + \frac{\partial (v_{-1} \times v_0)}{\partial v_0} \partial v_0 = 5 \times 1 + 2 \times 0 = 5$
- $\partial v_3 = \frac{\partial sin(v_0)}{\partial v_0} \partial v_0 = cos(5) \times 0$
- $\partial v_4 = \partial v_1 + \partial v_2 = 0.5 + 5$
- $\partial v_5 = \partial v_4 - \partial v_3 = 5.5 - 0$
- $\partial y = \partial v_5 = 5.5$

- $v_{-1} = x_1 = 2$
- $v_0 = x_2 = 5$
- $v_1 = \boxed{\ln v_{-1}} = \ln 2 = 0.693$
- $v_2 = v_{-1} \times v_0 = 2 \times 5 = 10$
- $v_3 = sin(v_0) = sin(5) = -0.959$
- $v_4 = v_1 + v_2 = 0.693 + 10 = 10.693$
- $v_5 = v_4 - v_3 = 10.693 + 0.959 = 11.652$
- $y = v_5 = 11.652$

(FA) Automatic Differentiation

For computing the derivative of f with respect to x_1 we start by associating with each intermediate variable v_i a derivative: $\partial v_i = \dfrac{\partial v_i}{\partial x_1}$.

Then we apply the chain rule to each elementary operation:

This expressions can be automatically evaluated (**1 function evaluation at machine precision**!) because we know the analytical derivatives of these functions

- $\partial v_{-1} = \frac{\partial x_1}{\partial x_1} = 1$
- $\partial v_0 = \frac{\partial x_2}{\partial x_1} = 0$
- $\partial v_1 = \boxed{\frac{\partial \ln(v_{-1})}{\partial v_{-1}}} \partial v_{-1} = \boxed{1/2} \times 1 = 0.5$
- $\partial v_2 = \frac{\partial(v_{-1} \times v_0)}{\partial v_{-1}} \partial v_{-1} + \frac{\partial(v_{-1} \times v_0)}{\partial v_0} \partial v_0 = 5 \times 1 + 2 \times 0 = 5$
- $\partial v_3 = \frac{\partial sin(v_0)}{\partial v_0} \partial v_0 = cos(5) \times 0$
- $\partial v_4 = \partial v_1 + \partial v_2 = 0.5 + 5$
- $\partial v_5 = \partial v_4 - \partial v_3 = 5.5 - 0$
- $\partial y = \partial v_5 = 5.5$

- $v_{-1} = x_1 = 2$
- $v_0 = x_2 = 5$
- $v_1 = \ln v_{-1} = \ln 2 = 0.693$
- $v_2 = v_{-1} \times v_0 = 2 \times 5 = 10$
- $v_3 = sin(v_0) = sin(5) = -0.959$
- $v_4 = v_1 + v_2 = 0.693 + 10 = 10.693$
- $v_5 = v_4 - v_3 = 10.693 + 0.959 = 11.652$
- $y = v_5 = 11.652$

(FA) Automatic Differentiation

For computing the derivative of f with respect to x_1 we start by associating with each intermediate variable v_i a derivative: $\partial v_i = \dfrac{\partial v_i}{\partial x_1}$.

Then we apply the chain rule to each elementary operation:

- $\partial v_{-1} = \frac{\partial x_1}{\partial x_1} = 1$
- $\partial v_0 = \frac{\partial x_2}{\partial x_1} = 0$
- $\partial v_1 = \frac{\partial \ln(v_{-1})}{\partial v_{-1}} \partial v_{-1} = 1/2 \times 1 = 0.5$
- $\partial v_2 = \frac{\partial (v_{-1} \times v_0)}{\partial v_{-1}} \partial v_{-1} + \frac{\partial (v_{-1} \times v_0)}{\partial v_0} \partial v_0 = 5 \times 1 + 2 \times 0 = 5$
- $\partial v_3 = \frac{\partial sin(v_0)}{\partial v_0} \partial v_0 = cos(5) \times 0$
- $\partial v_4 = \partial v_1 + \partial v_2 = 0.5 + 5$
- $\partial v_5 = \partial v_4 - \partial v_3 = 5.5 - 0$
- $\partial y = \partial v_5 = 5.5$

Our space is populated with these variables!

- $v_{-1} = x_1 = 2$
- $v_0 = x_2 = 5$
- $v_1 = \ln v_{-1} = \ln 2 = 0.693$
- $v_2 = v_{-1} \times v_0 = 2 \times 5 = 10$
- $v_3 = sin(v_0) = sin(5) = -0.959$
- $v_4 = v_1 + v_2 = 0.693 + 10 = 10.693$
- $v_5 = v_4 - v_3 = 10.693 + 0.959 = 11.652$
- $y = v_5 = 11.652$

At the end we have the derivative of f with respect to x_1 at (2,5).

(FA) Automatic Differentiation

We have seen **forward accumulation AD**.

To compute the gradient of this example function, which requires the derivatives of f with respect to not only x_1 but also x_2, an **additional sweep** is performed over the computational graph using the seed values $v_{-1} = 0, v_0 = 1$.

Forward accumulation is efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with $n \lll m$ (we need n sweeps).

For cases $n \ggg m$ **a different technique is needed**.

Automatic Differentiation: Forward & Reverse Mode

Given $L(x) = f(g(h(x)))$, the chain rule says that its gradient is:

$$\frac{\partial L}{\partial x} = \frac{\partial f}{\partial g} \times \frac{\partial g}{\partial h} \times \frac{\partial h}{\partial x}$$

If we evaluate this product from right-to-left: $\frac{\partial f}{\partial g} \times (\frac{\partial g}{\partial h} \times \frac{\partial h}{\partial x})$, **the same order as the computations themselves were performed**, this is called **forward-mode differentiation**.

Luckily, **we can also propagate derivatives backward from a given output**:

$$(\frac{\partial f}{\partial g} \times \frac{\partial g}{\partial h}) \times \frac{\partial h}{\partial x}$$

This is called **reverse-mode differentiation**.

Automatic Differentiation: Reverse Mode

- $v_{-1} = x_1 = 2$
- $v_0 = x_2 = 5$
- $v_1 = \ln v_{-1} = \ln 2 = 0.693$
- $v_2 = v_{-1} \times v_0 = 2 \times 5 = 10$
- $v_3 = sin(v_0) = sin(5) = -0.959$
- $v_4 = v_1 + v_2 = 0.693 + 10 = 10.693$
- $v_5 = v_4 - v_3 = 10.693 + 0.959 = 11.652$
- $y = v_5 = 11.652$

- $\partial v_5 = 1$
- $\partial v_4 = \partial v_5 \frac{\partial v_5}{\partial v_4} = 1 \times 1 = 1$
- $\partial v_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3} = \partial v_5 \times -1 = -1$
- $\partial v_1 = \partial v_4 \frac{\partial v_4}{\partial v_1} = \partial v_4 \times 1 = 1$
- $\partial v_2 = \partial v_4 \frac{\partial v_4}{\partial v_2} = \partial v_4 \times 1 = 1$
- $\partial v_0 = \partial v_3 \frac{\partial v_3}{\partial v_0} = \partial v_3 \times \cos v_0 = -0.284$
- $\partial v_{-1} = \partial v_2 \frac{\partial v_2}{\partial v_{-1}} = \partial v_2 \times v_0 = 5$
- $\partial v_0 = \partial v_0 + \partial v_2 \frac{\partial v_2}{\partial v_0} = \partial v_0 + \partial v_2 \times v_{-1} = 1.716$
- $\partial v_{-1} = \partial v_{-1} + \partial v_1 \frac{\partial v_1}{\partial v_{-1}} = \partial v_{-1} + \partial v_1 / v_{-1} = 5.5$
- $\partial x_2 = \partial v_0 = 1.716$
- $\partial x_1 = \partial v_{-1} = 5.5$

This is a two-stage process. In the first stage the original function code is run forward, populating variables. In the second stage, derivatives are calculated by propagating in reverse, from the outputs to the inputs.

Automatic Differentiation: Reverse Mode

$\partial v_5 = \partial y / \partial y$

- $\partial v_5 = 1$
- $\partial v_4 = \partial v_5 \frac{\partial v_5}{\partial v_4} = 1 \times 1 = 1$
- $\partial v_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3} = \partial v_5 \times -1 = -1$
- $\partial v_1 = \partial v_4 \frac{\partial v_4}{\partial v_1} = \partial v_4 \times 1 = 1$
- $\partial v_2 = \partial v_4 \frac{\partial v_4}{\partial v_2}$
- $\partial v_0 = \partial v_3 \frac{\partial v_3}{\partial v_0}$
- $\partial v_{-1} = \partial v_2 \frac{\partial v_2}{\partial v_{-1}} = \partial v_2 \times v_0 = 5$
- $\partial v_0 = \partial v_0 + \partial v_2 \frac{\partial v_2}{\partial v_0} = \partial v_0 + \partial v_2 \times v_{-1} = 1.716$
- $\partial v_{-1} = \partial v_{-1} + \partial v_1 \frac{\partial v_1}{\partial v_{-1}} = \partial v_{-1} + \partial v_1 / v_{-1} = 5.5$
- $\partial x_2 = \partial v_0 = 1.716$
- $\partial x_1 = \partial v_{-1} = 5.5$

- $v_{-1} = x_1 = 2$
- $v_0 = x_2 = 5$
- $v_1 = \ln v_{-1} = \ln 2 = 0.693$
- $v_2 = v_{-1} \times v_0 = 2 \times 5 = 10$
- $v_3 = sin(v_0) = sin(5) = -0.959$
- $v_4 = v_1 + v_2 = 0.693 + 10 = 10.693$
- $v_5 = v_4 - v_3 = 10.693 + 0.959 = 11.652$
- $y = v_5 = 11.652$

The most important property of reverse accumulation AD is that it is cheaper than forward accumulation AD for funtions with a high number of input variables. In our case, 1 sweep!

This is a two-stage process. In the first stage the original function code is run forward, populating variables. In the second stage, derivatives are calculated by propagating in reverse, from the outputs to the inputs.

Automatic Differentiation

arXiv:1502.05767v4 [cs.SC] 5 Feb 2018

Automatic Differentiation in Machine Learning: a Survey

Atılım Güneş Baydin GUNES@ROBOTS.OX.AC.UK
Department of Engineering Science
University of Oxford
Oxford OX1 3PJ, United Kingdom

Barak A. Pearlmutter BARAK@PEARLMUTTER.NET
Department of Computer Science
National University of Ireland Maynooth
Maynooth, Co. Kildare, Ireland

Alexey Andreyevich Radul AXCH@MIT.EDU
Department of Brain and Cognitive Sciences
Massachusetts Institute of Technology
Cambridge, MA 02139, United States

Jeffrey Mark Siskind QOBI@PURDUE.EDU
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, United States

Abstract

Derivatives, mostly in the form of gradients and Hessians, are ubiquitous in machine learning. Automatic differentiation (AD), also called algorithmic differentiation or simply "autodiff", is a family of techniques similar to but more general than backpropagation for efficiently and accurately evaluating derivatives of numeric functions expressed as computer programs. AD is a small but established field with applications in areas including computational fluid dynamics, atmospheric sciences, and engineering design optimization. Until very recently, the fields of machine learning and AD have largely been unaware of each other and, in some cases, have independently discovered each other's results. Despite its relevance, general-purpose AD has been missing from the machine learning toolbox, a situation slowly changing with its ongoing adoption under the names "dynamic computational graphs" and "differentiable programming". We survey the intersection of AD and machine learning, cover applications where AD has direct relevance, and address the main implementation techniques. By precisely defining the main differentiation techniques and their interrelationships, we aim to bring clarity to the usage of the terms "autodiff", "automatic differentiation", and "symbolic differentiation" as these are encountered more and more in machine learning settings.

Keywords: Backpropagation, Differentiable Programming

Automatic Differentiation

Example: https://hackmd.io/@jordivitria/AD

Autograd

Autograd is a Python module (with only one function) that implements automatic differentiation.

```
!pip install autograd
```

Autograd can automatically differentiate Python and Numpy **code**:

- It can handle most of Python's features, including **loops, if statements, recursion and closures**.
- Autograd allows you to compute gradients of many types of data structures (Any nested combination of lists, tuples, arrays, or dicts).
- It can also compute higher-order derivatives.
- Uses **reverse-mode differentiation** (backpropagation) so it can efficiently take gradients of scalar-valued functions with respect to array-valued or vector-valued arguments.

Autograd

```
import autograd.numpy as np
from autograd import grad

x = np.array([2, 5], dtype=float)

def test(x):
    return np.log(x[0]) + x[0]*x[1] - np.sin(x[1])

grad_test = grad(test)
print "({:.2f},{:.2f})".format(grad_test(x)[0],grad_test(x)[1])
```

$f : \mathbb{R}^2 \to \mathbb{R}$

$\nabla f : \mathbb{R}^2$

$\partial f / \partial x_1$ $\partial f / \partial x_2$

How to train a 1-D 1-layer neural network

```
# Build a toy dataset.
inputs = np.array([[0.52, 1.12,  0.77],
                   [0.88, -1.08, 0.15],
                   [0.52, 0.06, -1.30],
                   [0.74, -2.49, 1.39]])
targets = np.array([True, True, False, True])

weights = optimize(inputs, targets, training_loss)
print "Weights:", weights
```

How to train a 1-D 1-layer neural network

$$\hat{y} = \frac{1}{1 + \exp^{-(w_0 + w_1 x)}}$$

$$L = -\sum_{1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

Binary Cross Entropy

```
import autograd.numpy as np
from autograd import grad

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def logistic_predictions(weights, inputs):
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights, inputs, targets):
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))

def optimize(inputs, targets, training_loss):
    # Optimize weights using gradient descent.
    gradient_loss = grad(training_loss)
    weights = np.zeros(inputs.shape[1])
    print "Initial loss:", training_loss(weights, inputs, targets)
    for i in xrange(100):
        weights -= gradient_loss(weights, inputs, targets) * 0.01
    print "Final loss:", training_loss(weights, inputs, targets)
    return weights
```

How to train a 1-D 1-layer neural network

```
w = optimize(inputs, targets, training_loss)
print("Weights:",w)
```

```
Initial Loss 1.2158416689265426
Intermediate Loss 1.1166005097926057
Intermediate Loss 0.014461377073052091
Intermediate Loss 0.007250436160028394
Intermediate Loss 0.004836042805300988
Intermediate Loss 0.0036274714515086293
Intermediate Loss 0.0029020220592145167
Intermediate Loss 0.002418303261147632
Intermediate Loss 0.002072764326614986
Intermediate Loss 0.0018136037829715383
Intermediate Loss 0.0016120346046981056
Final Loss 0.0014509266460410865
Weights: [ 2.76196259 -1.66165016  8.3258685   2.05455772]
```

https://colab.research.google.com/drive/1DQlZnIsbrtsGvojIX_i3Gi2uVmufxTaj?usp=sharing

Deep Learning Tricks

AD is a critical component when developing deep models because the use of SGD is much more easy and robust (f.e. derivative computation is free of bugs!), but in spite of this fact, optimization of deep models is **not yet an easy task**.

Gradient-based optimization still suffers from some problems. For example, the system can be **poorly conditioned** (changing one parameter requires precise compensatory changes to other parameters to avoid large increases in the optimization criterion).

In order to address each issues, deep learning community has developed some **tricks**.

- First, **gradient tricks**, namely methods to make the **gradient either easier to calculate or to give it more desirable properties**.
- And second, **optimization tricks**, namely new methods related to **stochastic optimization**.

Deep Learning Tricks

- In calculating the stochastic gradient, it is tempting to do the minimal amount of computation necessary to obtain an estimate, which would involve a single sample from the training set.

  In practice it has proven much better to use a **block of samples (minibatch)**, on the order of dozens. This has two advantages: the first is **less noise**, and the second is that **data-parallelism** can be used.

Deep Learning Tricks

- **Rectified linear units** (ReLU) instead of sigmoids.

  Classic multi-layer perceptrons use the sigmoid activation function, but **this has a derivative which goes to zero when its input is too strong**.

  That means that when a unit in the network receives a very strong signal, it becomes difficult to change. Using a rectified linear unit (ReLU) function, overcomes this problem, making the system more plastic even when strong signals are present.

Deep Learning Tricks

- **Gradient clipping**. In the domain of deep learning, there are often **outliers** in the training set: exemplars that are being classified incorrectly, for example, or improper images in a visual classification task, or mislabeled examples, and the like.

  These can cause a **large gradient inside a single mini-batch**, which washes out the more appropriate signals. For this reason a technique called gradient clipping is often used, in which components of the gradient exceeding a threshold (in absolute value) are pushed down to that threshold.

Deep Learning Tricks

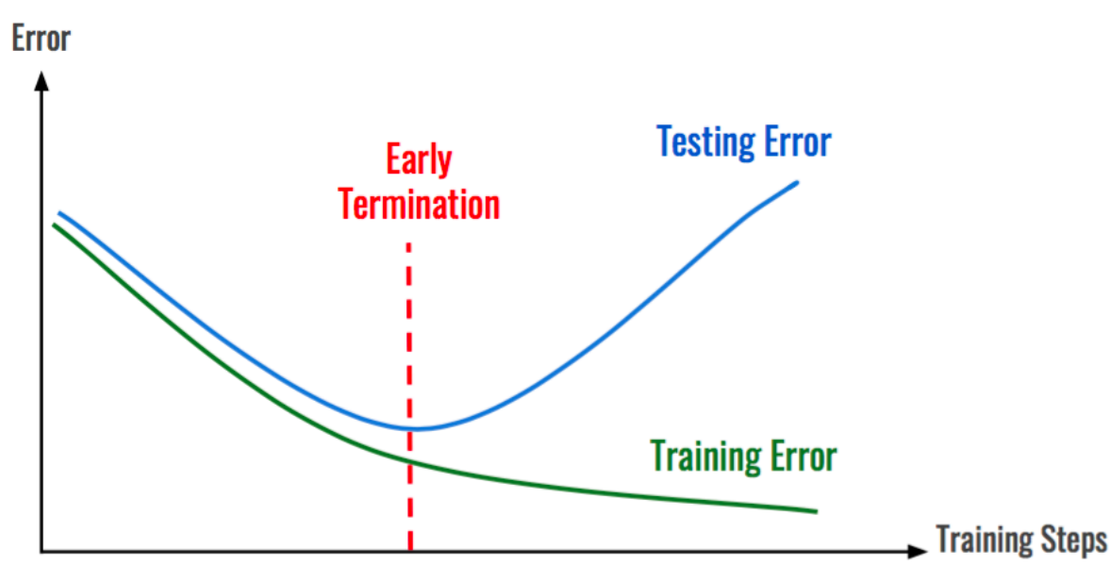
- **Batch normalization**. Batch Normalization is a technique to provide any layer in a neural network with inputs that are zero mean/unit variance.

- **Careful initialization**. Considering how the variances of activation values and gradients can be maintained between the layers in a network leads to intelligent normalized initialization schemes, which enable substantially faster optimization convergence.

Deep Learning Tricks

- **Early stopping**. Our **primary concern is generally not performance on the training set, but on as-yet-unseen new data**.

  This is addressed by **early stopping**, in which an estimate of performance on unseen data is maintained, and optimization is halted early when this estimated generalization performance stops improving, even if performance on the training set is continuing to improve.



Deep Learning Tricks

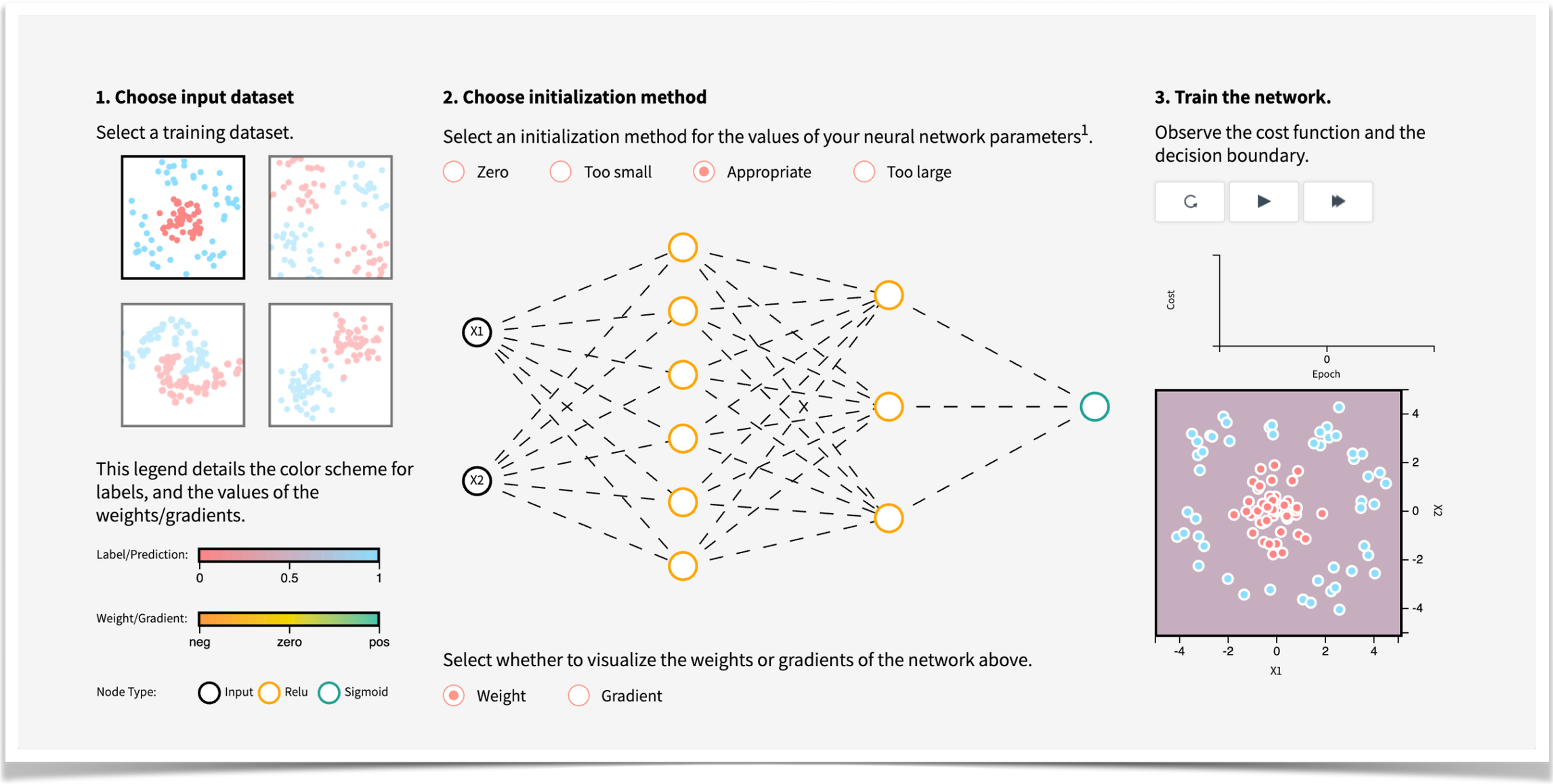
- **Regularization**. Norm regularization is **a penalty imposed on the model's objective function for using weights that are too large**. This is done by adding an extra term onto the function. For example, if we are training a neural network that uses the categorical cross entropy loss, we get:

$$L(y, \hat{y}) = -\frac{1}{N}\sum_{n \in N}\sum_{i \in C} y_{n,i} \log \hat{y}_{n,i} + \boxed{\lambda \frac{1}{N}\sum_{i,j,k} W_{i,j,k}^2}$$

where W_{ijk} represent all network parameters.

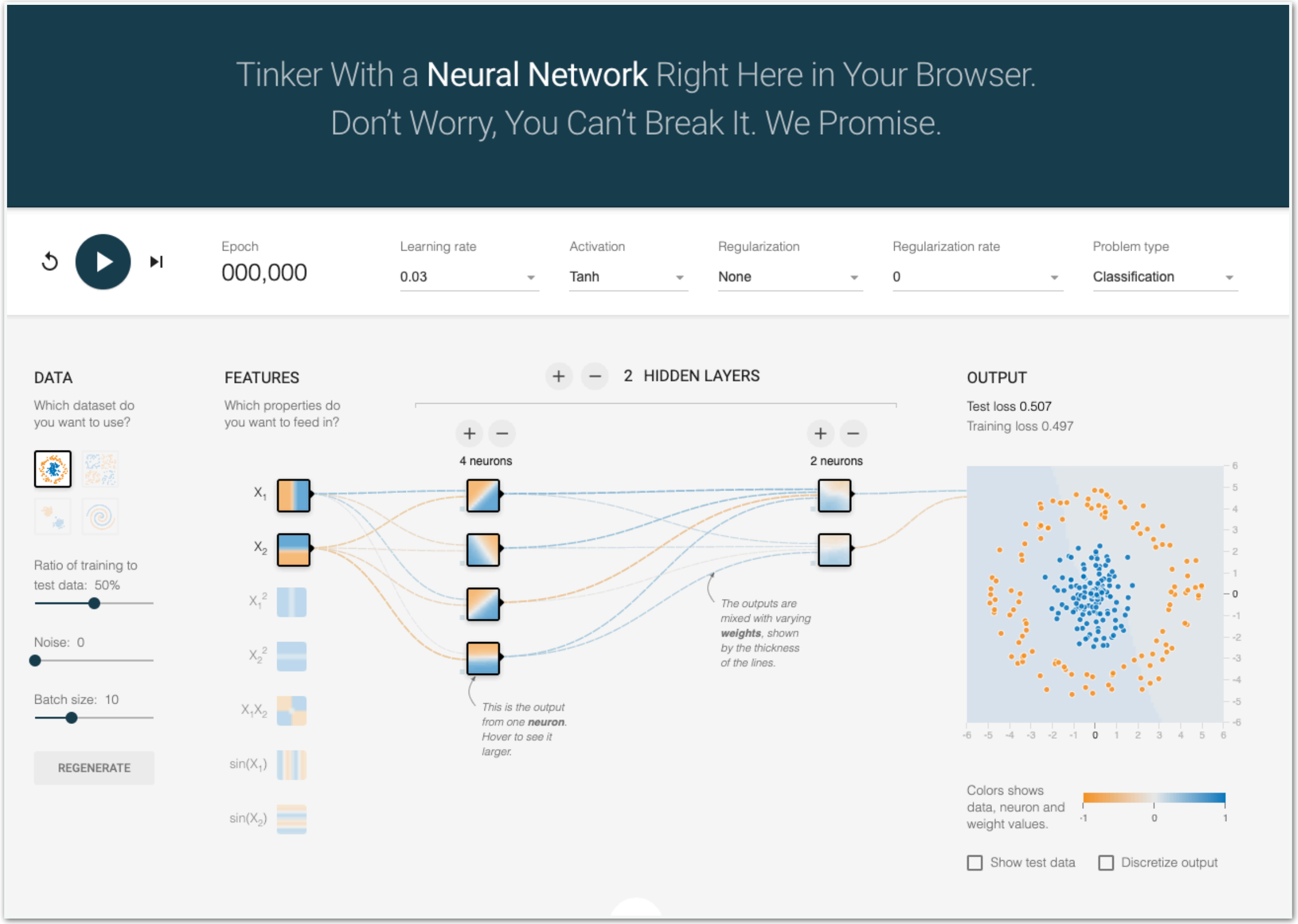
Initialization

The initialization step can be critical to the model's ultimate performance, and it requires the right method.



https://www.deeplearning.ai/ai-notes/initialization/

Playing with neural networks



http://playground.tensorflow.org/

1st Assignment: Building and Training a NN from Scratch (with AD)

Delivery (Campus Virtual) : 15/10