Deep Learning

How to learn DL models?

Automatic Differentiation

We have seen that in order to optimize our models 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:

$$h_0 = x$$
$$h_1 = \sigma(W_1 h_o + b_1)$$
$$\cdots$$
$$h_m = \sigma(W_m h_{m-1} + b_m)$$

DL models can have hundreds of millions of parameters!

We need to compute:

$$\frac{\delta L}{\delta w_{ij}^p}, \frac{\delta L}{\delta b_j}$$

where w_{ij}^p is an element of W_p.

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).

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

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

if there are more than one variable g_i in-between y and x (i.e. if f is a two-dimensional functions $f(g_1(x), g_2(x))$, then $\frac{\partial f}{\partial x} = \sum_i \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial x}$)

Now, **let's see how AD allows the accurate evaluation of derivatives at machine precision**, with only a small constant factor of overhead.

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:

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

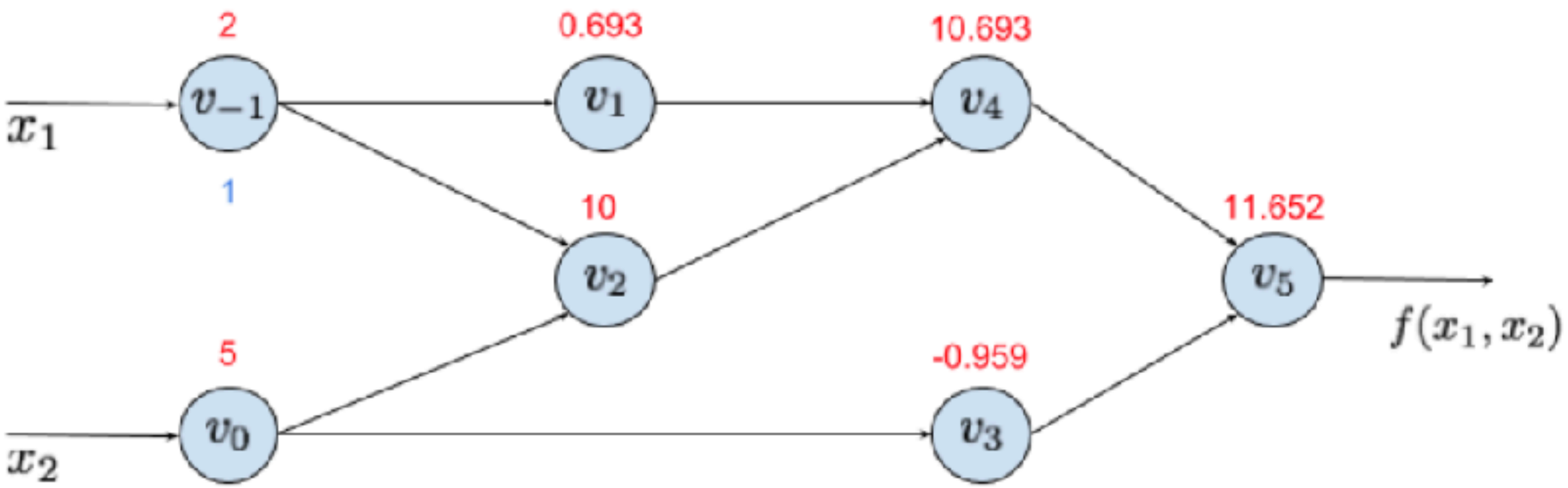
and let's write how to evaluate $f(2,5)$ via a sequence of primitive operations:

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

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

Automatic Differentiation

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



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: $\partial v_i = \frac{\partial v_i}{\partial x_1}$.

Then we apply the chain rule to each elementary operation:

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

At the end we have the derivative of f with respect to x_1 at (2,5). We have seen **forward accumulation AD**. Forward accumulation is efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with $n << m$. For cases $n >> m$ **a different technique is needed.**

Automatic Differentiation: 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:

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

Automatic Differentiation: Reverse Mode

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

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.

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.

Automatic Differentiation

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@PARC.COM
Department of Computer Science
National University of Ireland Maynooth
Maynooth, Co. Kildare, Ireland

Alexey Andreyevich Radul ARCH@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

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])
```

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

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

```
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

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

```
# 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
```

Any complex function that can be decomposed in a set of elementary functions can be derived in an automatic way, at machine precision, by this algorithm!

We no longer need to code complex derivatives to apply SGD!

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 unbiased estimate, which would involve a single sample from the training set. In practice it has proven much better to use a **block of samples**, on the order of dozens. This has two advantages: the first is less noise, and the second is that data-parallelism can be used.
- **Rectified linear units** 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.
- **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.
- **Early stopping**. When fitting a dynamic system to data, as exact a match as possible is desired, so the true optimum is sought. This is not the case in machine learning, where the optimization is of error on a training set, while the primary concern is generally not performance on the training set, but on as-yet-unseen new data. There is often a tradeoff between the two, encountered after optimization has proceeded for a significant amount of time. 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.

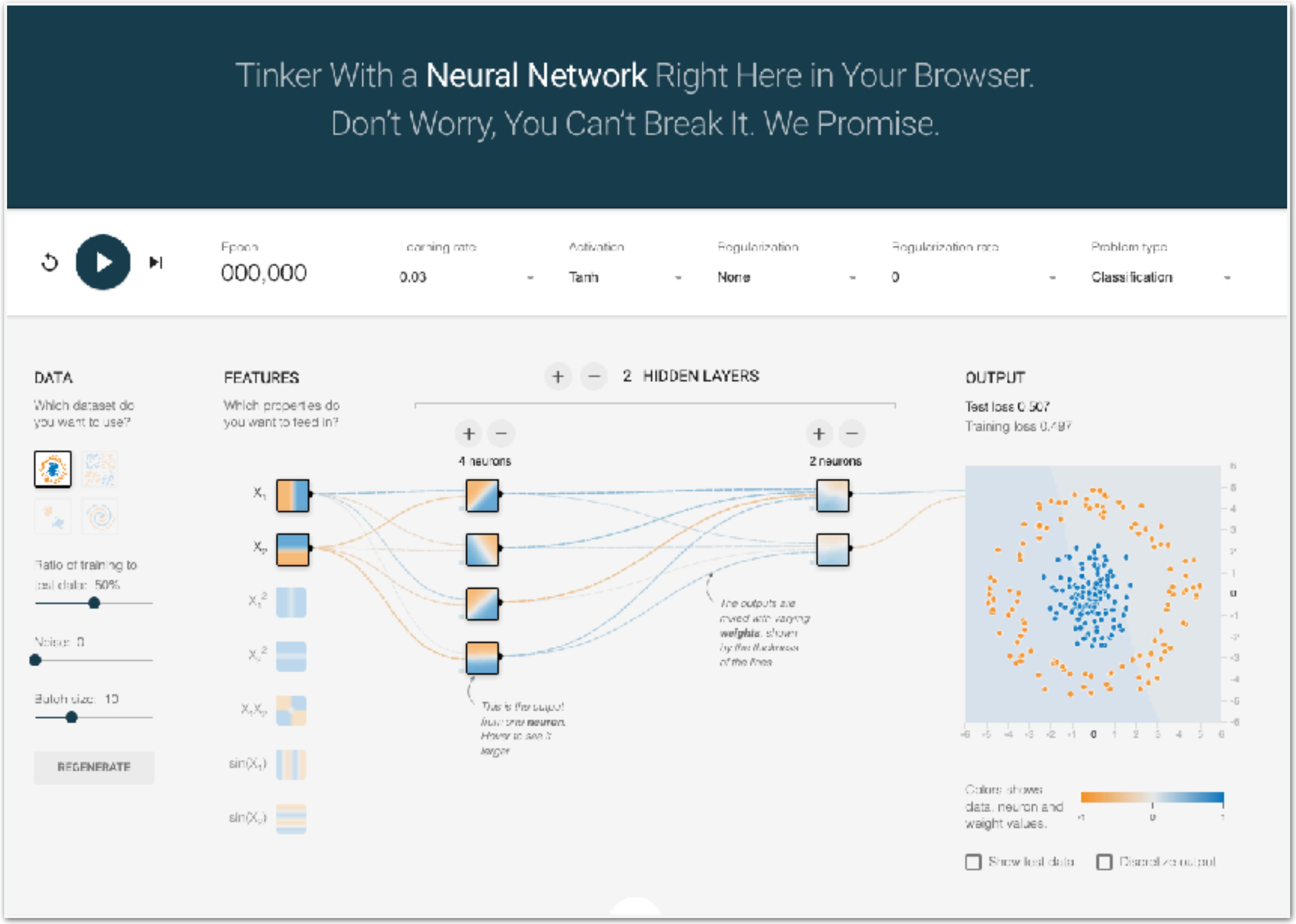
Deep Learning Tricks

- **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:

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

where W_{ijk} represent all network parameters.

Playing with neural networks



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

Delivery (Campus Virtual) : 30/10