CAL POLY

101 202 480

Instructor Info

Projects

Tile Driver I

Tile Driver II

Biogimmickry

Mine Shafted

Board Stupid

Moonlander II

Know Op

Course Details

Description

Assessment

Resources

Calendar

Policies

Enrollment

Code Style

Student Conduct

Accommodation

Guides

# Project VII: Know Op

## Due Mon, Aug 30 by 9PM Pacific Time

To begin, use the following file: http://users.csc.calpoly.edu/~dkauffma/480/knowop.py

- To download locally, right-click link and choose "Save link as...".
- To download and unzip on a CSL server run:
  `wget http://users.csc.calpoly.edu/~dkauffma/480/knowop.py`
  `unzip knowop.py`

# Description

Supervised Learning (a subset of Machine Learning) is primarily concerned with the process of approximating a function that maps some input `x` to an appropriate output `y`. These functions are usually "learned" by constructing some sort of model built iteratively over many observations of known x-to-y mappings, known as a training set. In the simplest case, once the model is "trained" on this set, its accuracy can be evaluated on a testing set, which is composed of other known x-to-y mappings that were not seen during training. In doing so, a data scientist can have some degree of confidence that the model generalizes well to inputs that have yet to be encountered by the model.

While there are many forms of learning models (including Naive Bayes, Decision Trees, and Support Vector Machines, to name a few), the most sophisticated of these come in the form of artificial neural networks (often called "neural nets" or just "networks"). These networks are *very loosely* based on how the human brain functions, and their recent success is largely tied to how well they can scale and be adapted to suit a problem (in addition to the accessibility of very large data sets and massively parallel computing via GPUs). Put another way, neural networks represent one of the most versatile means to approximate a function.

## Layered Model

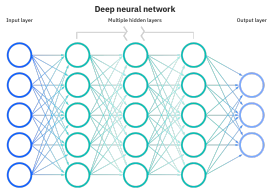
Casey Submission System

Getting Started

Secure Shell (SSH)

Vim Editor

The architecture of a neural network is composed of a sequence of *layers*, with each layer composed of a number of *neurons* (also known as units). Neurons each contain exactly one bias value, which is used in generating a value to send to the next layer. The neurons in one layer are connected to the neurons in another layer by weights, which may be thought of as edges between two vertices. In a *Feedforward Network*, these edges are directed toward the output layer only; because neurons within the same layer are not connected, these networks represent directed Multipartite Graphs.




https://www.ibm.com/cloud/learn/neural-networks

The minimum number of layers in a network is two: an input layer and an output layer. However, these simple networks rarely generalize to meaningful mappings, and therefore additional layers are added in between to increase the sophistication of the network. Because the inputs and outputs of these layers are not directly observed, they are known as *hidden layers*, and any network that uses one or more hidden layers is said to be a deep neural network (and hence engages in "deep learning").

## Network Training

The goal of training a neural network is to find suitable values for the weights and biases (mentioned above). An optimal assignment of these values would allow any never-before-seen input to map to its expected output, providing a highly accurate approximation of the target function.

At the start of the training process, the weights are often initialized to small random values; because the biases are added to the output instead of multiplied, they may be initialized to zero. On each iteration of a forward propagation, a training sample is sent through the network, layer by layer, up until and including the output layer. The output is then compared to the expected output (which is known during supervised training), and the *loss* is computed using a loss function (e.g. squared error).

At each layer, the output of each neuron to be sent to the next layer is computed as

```
a = σ(wx+b)
```

where the input vector `x` is multiplied by the neuron's incoming weights `w` and then added to the bias `b`. The result is then passed to the activation function `σ`, which is typically the ReLU function for hidden layers and the sigmoid function for the output layer.

On each iteration of training, forward propagation processes multiple training samples - but usually not *all* training samples, as that would take too long with the typically large training sets used. Instead, small random subsets of the training set called mini-batches are used for that iteration; on the next iteration, a new random mini-batch is created. The use of mini-batching allows for many iterations of the training process while still making it likely that every training sample is used at least once.

Once all mini-batch training samples have been forward propagated, the losses are aggregated (usually averaged) to a *cost*, which serves as a measure of how well the network did during that iteration of training. The cost is then used to determine how to update the output layer's weights and biases. These updates are in turn used to update the previous layer's weights and biases, which continues up until but not including the input layer (which has no weights or biases). This process is known as backpropagation, and is central to training the network.

## Backpropagation

Once the cost is known, the next step is to determine by how much the weights and biases need to be adjusted in order for the cost to be reduced on subsequent iterations. Doing so requires knowing the *gradient* (essentially a slope in multidimensional space) of certain values, which will point to their direction of steepest descent with respect to the cost and thus provide faster learning. This technique is known as gradient descent - or stochastic gradient descent (SGD) when mini-batching is used.

Consider that the output of a network during forward propagation may be thought of as a composite of activation functions (where

`x` is the input vector, `y` is the output vector, `W` is a weight matrix and `b` is a bias vector).

$$y = \sigma_2(W_2\sigma_1(W_1\sigma_0(W_0x+b_0)+b_1)+b_2)$$

Thus, to get the gradients, the Chain Rule from calculus may be used. Gradients are computed starting at the final (output layer) and working toward the input layer, with `da` initialized to the derivative of the loss function. Although the following formulas should be used, understanding why they work is not necessary to complete this assignment.

$$\begin{aligned} dz_n &= da_n \odot g'_n(z_n) \\ dW_n &= dz_n \cdot a^T_{n-1} \\ db_n &= dz_n \\ da_{n-1} &= W^T_n \cdot dz_n \end{aligned}$$

In the above, `n` denotes a value at the current layer and `n - 1` the layer that comes before it. The $\cdot$ operator refers to matrix multiplication whereas the $\odot$ operator refers to elementwise multiplication (or Hadamard product). Variables prefixed with a `d` refer to the gradient (derivative) of those values (e.g. $dz_n$ refers to the gradient vector of $z_n$).

## Updating Parameters

Once backpropagation has been performed on every sample that was forward propagated in a mini-batch, the weights and biases may be updated. Recall that it is these updates that ultimately train the network. Fortunately, unlike backpropagation, updating the network's parameters is more straightforward. In this step, a *learning rate* is used, which typically starts high at the beginning of the training process and decreases over time to keep the network from diverging. For each layer, update each weight and bias value by subtracting it from the negated product of the gradient and the learning rate.

$$w_{pq} = w_{pq} - \alpha\nabla$$

In the above, let $\alpha$ be the learning rate, $\nabla$ be the gradient, and `p` and `q` be the position of a single weight in the weight matrix to be updated. Each bias vector would be updated in a similar way.

# Implementation

**Allowed Modules:** math, itertools, random

You may optionally work in pairs for this assignment. However, the collaboration policy still applies; do not share code outside your pair. You may quit a pair after working together if you like, but you should then complete the assignment individually (i.e. do not join a new pair). Include both names of your pair at the top of your submission - even if you stopped working together - so that your submissions do not get flagged for collaboration.

The following free resources may help you in completing this assignment:

- Text: neuralnetworksanddeeplearning.com
- Videos: www.youtube.com/playlist?list=PLPaWThlrpebULGM5bbaCX6M7bRuzcM8rd (suggested list - not all videos required)

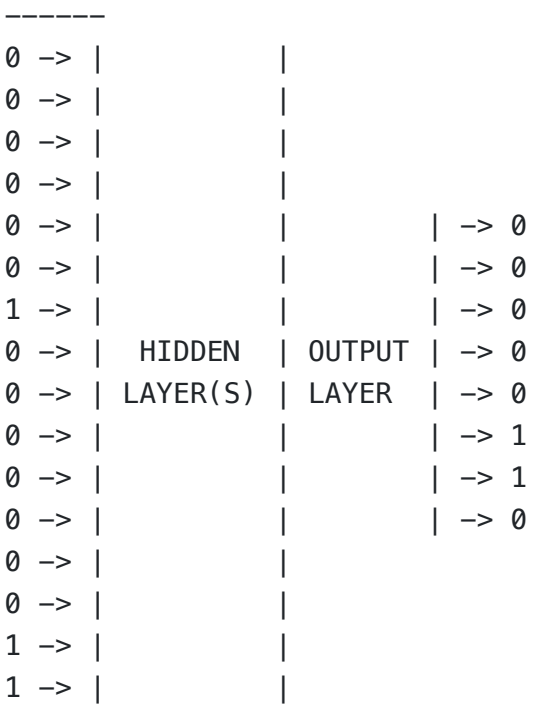
In this assignment, you will train a network to learn how to perform simple arithmetic operations. For example, given the operation `x + y`, the network must be able to accept two operands as input and output their sum. However, to help the network form more complex associations with these operands, they will be represented in 8-bit binary - specifically, as a tuple of `0`s and `1`s; for inputs of multiple operands, the tuples will be concatenated into a single tuple. You may assume inputs and outputs will always be non-negative integers (i.e. standard division will not be used) and that all outputs will be small enough to represent with 8 bits.

To illustrate, suppose the operation to be learned is multiplication (`x * y`) and we have the following sample:

```
Operand1: 2 -> 00000010
Operand2: 3 -> 00000011
  Result: 6 -> 00000110
```

In this example, the inputs and the expected outputs would look like:

```
INPUT
LAYER
```




In practice, the network will not be so accurate as to get exactly a `0` or `1` from its output layer (which uses the sigmoid activation function). Rather, these values will be somewhere between `0.0` and `1.0`; the farther an output is from `0.5` in either direction, the more confident the network is in that decision. Regardless of its confidence, each output value will be rounded to whichever integer is closer during network evaluation (e.g. a `0.49` will be rounded to `0`).

See the starter file for the following provided components.

### Classes

- `Math` provides a collection of static methods for mathematical operations
- `Layer` serves as the foundation for the network's representation

### Functions

- `create_samples` generates a mapping of input-output pairs to be used in training and testing
- `main` contains a simple test driver

```
train_network(
samples: Dict[Tuple[int, ...], Tuple[int,
...]],
```

```
i_size: int, o_size: int) -> List[Layer]
```

Given a training set (with labels) and the sizes of the input and output layers, create and train a network by iteratively propagating inputs (forward) and their losses (backward) to update its weights and biases. Return the resulting trained network, represented as an ordered list of `Layer` objects (with the output layer as the last one in the list).

# Submission

On a CSL server with `knowop.py` in your current directory:

| Instructor | Command |
|---|---|
| Daniel Kauffman | `/home/dkauffma/casey 480 knowop` |