Introduction to Machine Learning (by Implementation)

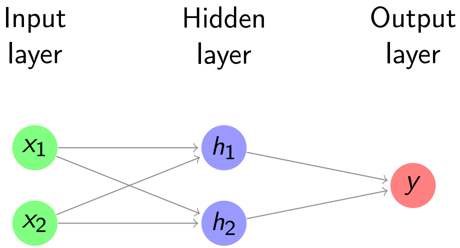
Lecture 8: Backpropagation

Ian J. Watson

University of Seoul

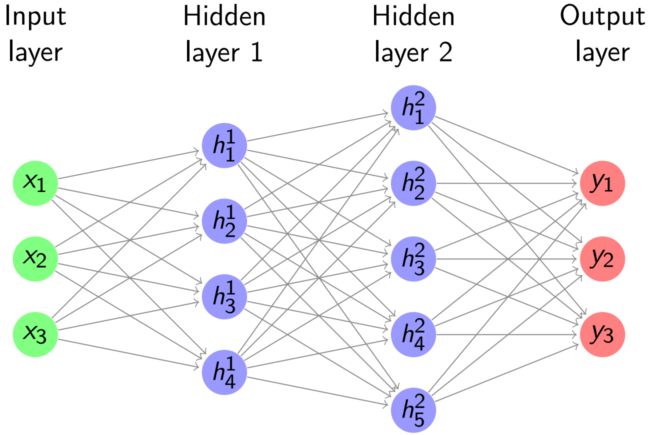
University of Seoul Graduate Course 2019

The Feed-Forward Neural Network



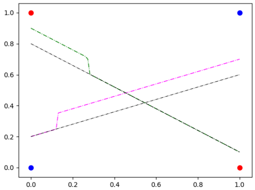
- A small feed-forward neural network
  - $y = f(x_1, x_2) = \sigma(-1 + 2x_1 + 2x_2) + \sigma(1 - 2x_1 - 2x_2)$
- Decompose the function into:
  - the *input layer* of $\hat{x}$,
  - the *hidden layer* which calculates $h_i = \beta_i \cdot x$ then passes if through the *activation function* σ, (called "sigmoid" in NN terms)
    - as in logistic, there is an extra β_0, called the *bias*, which controls how big the input into the node must be to activate
  - the *output layer* which sums the results of the hidden layer and gives y
    - $y = \sigma(0 + 1 \cdot h_1 + 1 \cdot h_2)$

Feed-Forward Neural Network

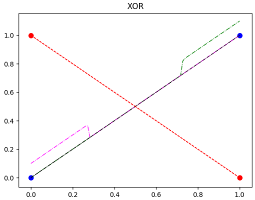


- We can even have several hidden layers
  - The previous layer acts the same as an *input layer* to the next layer
- We call each node in the network a *neuron*
  - At each neuron, the output of the node is $\sigma(\sum \text{weighted node inputs} + \text{bias})$

Training a Neural Network



$\rightarrow$

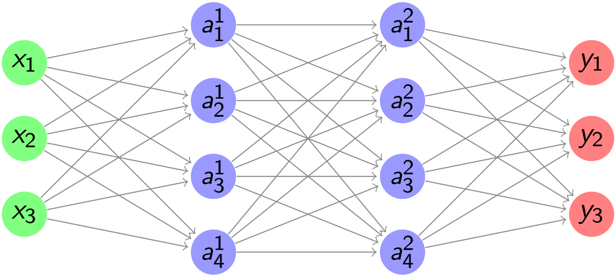


- What does it mean to train a neural network?
- Consider the XNOR network from last week
- There we set by hand, but could try to "train" the network
- Start with random weights and biases, reduce the loss function $C(x, y|w, b) = \sum_i |y_i^{\text{true}} - y(x_i)|^2$ where i ranges over our 4 samples (x_i, y_i) and $y(x_i)$ is the network output
  - And, of course, the way we've seen to do this is using *gradient descent*

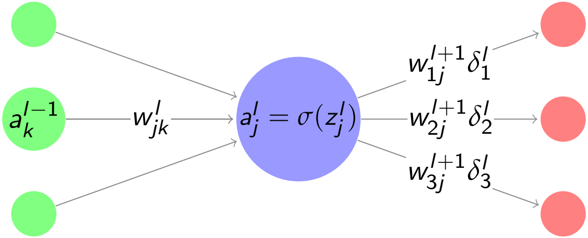
Gradient Descent on a Neural Network

- Consider running gradient descent on a neural network
- For some particular weight, w^l_{jk}, we want to find $\frac{\partial L}{\partial w^l_{jk}}$
- We could look at this and say, it big, complicated, lets use our gradient estimator: $\frac{\partial L}{\partial w^l_{jk}} = \frac{L(w^l_{jk}+\Delta)-L(w^l_{jk})}{\Delta}$ for some small Δ
- But in large networks, we can have millions of nodes: each evaluation of L requires one forward pass through the network, and we need two (at least) for each weight/bias
  - **This means millions of forward passes through the network for a single update**
- And remember, our stochastic algorithm used an update *per known datapoint*
- We need a better way ...

Notation



- Of course, the network has a very particular structure: series of evaluations passed from one layer to another, sums inside functions
- Some notation:
  - We have a network of L layers [input layer 0, output layer L]
  - j'th node on the l'th layer have output $a^l_j = \sigma(z^l_j) = \sigma(\sum_k w^l_{jk} a^{l-1}_k + b^l_j)$
  - So, the output of the network is a^L_j
  - and the the inputs $x_j = a^0_j$



- It turns out (from the chain rule), that the gradients can be calculated very simply with one forward pass, and one backward pass propagating the derivatives (hence *backpropagation*)
- Imagine we sit at node a_j^l and we want to find the derivative of w_{jk}^l
  - $\frac{\partial L}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$, $\frac{\partial L}{\partial b_j^l} = \delta_j^l$
  - $\delta_j^l = \sigma'(z_j^l)\sum_{k'} w_{k'j}^{l+1}\delta_{k'}^{l+1}$
- That is, the derivative is a product of the activation in a_k^{l-1} and the weighted sum of derivatives coming from the outputs $\delta^{l+1}{}_{k'}$
  - Notice that the δ_j^l we calculate on this layer will then be used when setting weights on layer $l-1$

Backpropagation at the Output Layer

- $\delta^l_j = \sigma'(z^l_j)\sum_{k'} w^{l+1}_{k'j}\delta^{l+1}_{k'}$ can be thought of as the error of the node (look closely on previous page, all w^l_{jk} use the same δ^l_j)
- So, where does it originally come from?
- Well, at the final layer there is no δ^{L+1} to be able to use, so this is our starting point by considering the cost function
- $C = \frac{1}{2}\sum_j (y_j - a^L_j)^2 = \frac{1}{2}\sum_j (y_j - \sigma(z^L_j))^2 = \frac{1}{2}\sum_j (y_j - \sigma(\sum_k w^L_{jk} a^{L-1}_k + b^L_j))^2$
  - Think of the chain rule operating on the expanding piece at each step
- $\frac{\partial C}{\partial w^L_{jk}} = (a^L_j - y_j)\sigma'(z^L_j)a^{L-1}_k = a^{L-1}_k\delta^L_j$, $\frac{\partial C}{\partial b^L_j} = (a^L_j - y_j)\sigma'(z^L_j) = \delta^L_j$
- So, $\delta^L_j = (y_j - a^L_j)\sigma'(z^L_j)$ is our starting point for the backpropagation
  - Use it to set the weights on layer L, then go back a layer, use it as input to find δ^{L-1}_j and then set the weights on layer $L-1$ and so on
- Notice in the derivation, there was no particular property of σ used other than the fact that we can differentiate it
  - Implies that any activation function will work for backpropagation

Backpropagation Equations and Operation

- $\delta^L_j = (a^L_j - y_j)\sigma'(z^L_j)$
- $\delta^l_j = \sigma'(z^l_j)\sum_{k'} w^{l+1}_{k'j}\delta^{l+1}_{k'}$
- $\frac{\partial L}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$
- $\frac{\partial L}{\partial b^l_j} = \delta^l_j$
- TODO: WRITE OUT DEFNS of a vs z

- In the same way that the a^l_j are wrapping up the weighted sums and activations of the layers feeding forward, the δ^l_j wrap up the partial derivatives of the chain rule which must be expanding from the cost function
  - Hopefully, you can see how the proof for the transfer to previous layer would work by running further expansions of a^{L-1}_k on the previous page
- We calculate the a^l_j forward, then calculate the $\frac{\partial C}{\partial w^l_{jk}}$, δ^l_j backward
- And then use this to find $\frac{\partial C}{\partial b^l_j}$ and run our SGD

Exercises

- `initialize_weights(n_nodes, initialize_fn=random)`
  - `n_nodes` should be a list of the number of nodes at each layer, including input and output (see the `test_initialize_weights` in `test_neural` for further commentary)
  - Use your `rand.random` function to initialize randomly between 0 and 1
- Should have `feedforward` from last week, today, lets assume we always use `sigmoid` activation (so we can use $\sigma'(x) = \sigma(x)(1 - \sigma(x))$)
- `calculate_deltas(network, activations, y)`
  - Calculates the δ_j^l from the previous page
- `batch_update_nn(network, activation, deltas, eta)`
  - Returns the weights after one round of gradient descent updates
  - $w_{jk}^l \to w_{jk}^l - \eta \frac{\partial C}{\partial w_{jk}^l}$, $b_j^l \to b_j^l - \eta \frac{\partial C}{\partial b_j^l}$
  - Probably easiest to use deepcopy `from copy import deepcopy`, make a copy of the network, then update using indices, rather than trying to make the network as you go

Exercises

- `sgd_nn(x, y, theta0, eta=0.1)`
  - Similar structure as our previous stochastic gradient descent, but uses the functions above to do the updates of the weights on each sample
  - Instead of input functions, assume a sum of squares cost function and use the batch update sequence you've just written `feedforward_`, `calculate_deltas`, `batch_update_nn`
  - It can be useful to save the values of the cost function to monitor how much the network is changing, particularly to try out different eta
  - You might find it easier to drop the n_iterations and run n_epochs (times over dataset) with your own training schedule (eta choice)
- Try training a network on our xor problem from last week.
- Hint: use gaussian initialized weights, play with the alpha and n_iterations hyperparameters. You might need to try it a few times with different starting points to get good convergence
- Try training a network for the Fisher classification problem from two weeks ago
  - Play around with the network architecture (number of layers/nodes)
- Use the `multi_accuracy` and print out your best network and accuracy into `results.txt`