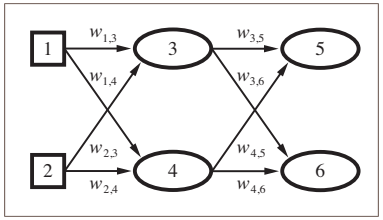
Tufts

Class #13:
Backpropagation &
Hyperparameter Tuning

Machine Learning (COMP 135)

1

Learning in Neural Networks



- A neural network can learn a classification function by adjusting its weights to compute different responses
- This process is another version of gradient descent: the algorithm moves through a complex space of partial solutions, always seeking to ***minimize*** overall error

2

Back-Propagation

Inputs: set of training examples, X
network, N, with ℓ layers

for every weight $w_{i,j} \in N$:
$w_{i,j} \longleftarrow$ small random value

while stopping condition not met:

for every training sample $(\mathbf{x}, \mathbf{y}) \in X$:
for every neuron n_j in input layer L_1:
output: $out_j \longleftarrow x_j$
for every layer L_k, $k = 2, \ldots, \ell$:
for every neuron n_j in L_k:
input: $in_j \longleftarrow \sum_i w_{i,j} \times out_i$
output: $out_j \longleftarrow g(in_j)$

for every neuron n_i in output layer L_ℓ:
error: $err_i \longleftarrow g'(in_i) \times (y_i - out_i)$
for every layer L_k, $k = (\ell - 1), \ldots, 2$:
for every neuron n_i in L_k:
error: $err_i \longleftarrow g'(in_i) \times \sum_j (w_{i,j} \times err_j)$

for every weight $w_{i,j} \in N$:
update: $w_{i,j} \longleftarrow w_{i,j} + (\alpha \times out_i \times err_j)$

Initial random weights

Stop when weights converge or error is minimized

Loop over all training examples, generating the network output…

… and then updating weights based on error made

3

Propagating Output Values Forward

for every neuron n_j in input layer L_1:
output: $out_j \longleftarrow x_j$
for every layer L_k, $k = 2, \ldots, \ell$:
for every neuron n_j in L_k:
input: $in_j \longleftarrow \sum_i w_{i,j} \times out_i$
output: $out_j \longleftarrow g(in_j)$

At first ("top") layer, each neuron simply outputs the relevant feature value

Go down layer-by-layer, calculating weighted input sums for each neuron, and computing output of the activation function g

4

Propagating Error Backward

for every neuron n_i in output layer L_ℓ:
  error: $err_i \longleftarrow g'(in_i) \times (y_i - out_i)$
for every layer L_k, $k = (\ell - 1), \ldots, 2$:
  for every neuron n_i in L_k:
    error: $err_i \longleftarrow g'(in_i) \times \sum_j (w_{i,j} \times err_j)$
for every weight $w_{i,j} \in N$:
  update: $w_{i,j} \longleftarrow w_{i,j} + (\alpha \times out_i \times err_j)$

At output ("bottom") layer, each error-value is set to the basic output error for the neuron, multiplied by the derivative of activation g

Go bottom-up and set error to derivative value multiplied by sum of error at the next layer down (weighting each such error appropriately)

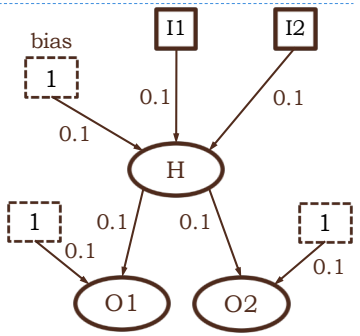
After all the error values are computed, update every weights in the network

5

A Back-Propagation Example

- Consider the following simple network, with:
  1. Two inputs
  2. A single hidden layer, consisting of one neuron
  3. Two output neurons
  4. Initial weights as shown
- Suppose we have the following data-point:

$(\mathbf{x}, \mathbf{y}) = ((0.5, 0.4), (1, 0))$



6

A Back-Propagation Example

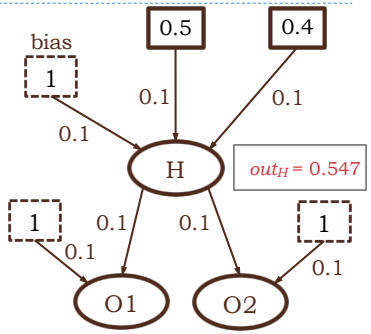
$(\mathbf{x}, \mathbf{y}) = ((0.5, 0.4), (1, 0))$

- For this data, we start by computing the output of H
- We have the weighted linear sum:

$$\sum_{in_H} = 0.1 + (0.1 \times 0.5) + (0.1 \times 0.4)$$
$$= 0.19$$

- And, assuming the logistic activation function, we get output:

$$out_H = \frac{1}{1 + e^{-0.19}} \approx 0.547$$



7

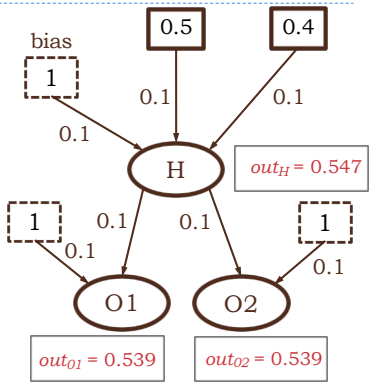
A Back-Propagation Example

$(\mathbf{x}, \mathbf{y}) = ((0.5, 0.4), (1, 0))$

- Next, we compute the output of each of the two output neurons
- Since each has identical weights, initial outputs will be the same:

$$\sum_{in_O} = 0.1 + (0.1 \times 0.547) = 0.1547$$

$$out_O = \frac{1}{1 + e^{-0.1547}} \approx 0.539$$



8

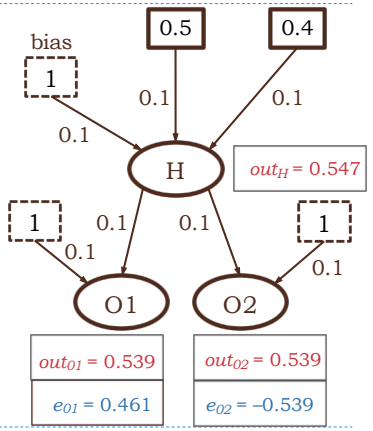
A Back-Propagation Example

$(\mathbf{x}, y) = ((0.5, 0.4), (1, 0))$

- Given the output vector y = (1,0), we can compute the error terms for the two output neurons:

$e_{O1} = 1 - 0.539 = 0.461$

$e_{O2} = 0 - 0.539 = -0.539$



9

A Back-Propagation Example

$(\mathbf{x}, \mathbf{y}) = ((0.5, 0.4), (1, 0))$

- The derivative of the activation function at each output neuron is:

$g'(in_O) = out_O \times (1 - out_O)$

$= 0.539 \times 0.461 \approx 0.2485$

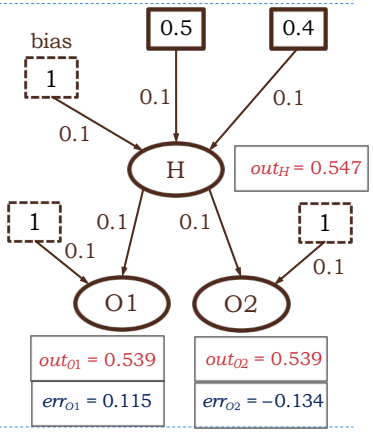
- And so we have our *err* values:

$err_{O1} = g'(in_O) \times e_{O1}$

$= 0.2485 \times 0.461 \approx 0.115$

$err_{O2} = g'(in_O) \times e_{O2}$

$= 0.2485 \times -0.539 \approx -0.134$



10

A Back-Propagation Example

$(\mathbf{x}, \mathbf{y}) = ((0.5, 0.4), (1, 0))$

- Similarly, we can compute the derivative of the activation function and the *err* value for the hidden-layer neuron, H:

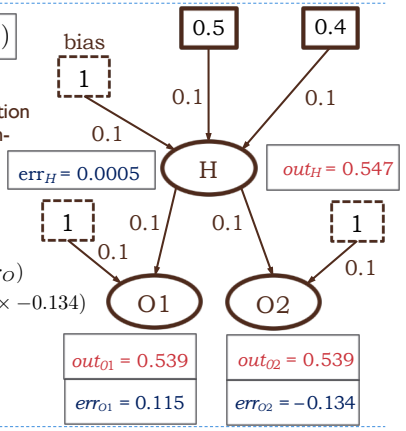
$g'(in_H) = out_H \times (1 - out_H)$

$= 0.547 \times 0.453 \approx 0.248$

$err_H = g'(in_H) \times (\sum_O w_{H,O} \times err_O)$

$= 0.248 \times (0.1 \times 0.115 + 0.1 \times -0.134)$

$= 0.248 \times -0.0019 \approx 0.0005$



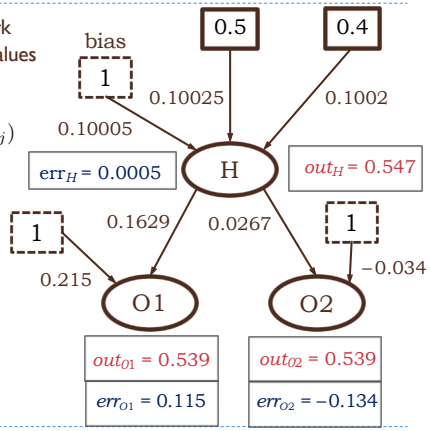
11

A Back-Propagation Example

- Finally, all weights in the network can be updated, using the *err* values (here, we will assume that the learning rate α = 1):

$w_{i,j} \longleftarrow w_{i,j} + (\alpha \times out_i \times err_j)$

Exercise for the reader:
Now that the weights are updated, re-compute the outputs of the network, along with the error values for each output neuron. What do we see?



12

Hyperparameters for Neural Networks

- Multi-layer (deep) neural networks involve a number of different possible design choices, each of which can affect classifier accuracy:
  - Number of hidden layers
  - Size of each hidden layer
  - Activation function employed
  - Regularization term (controls over-fitting)
- This is not unique to neural networks
  - Logistic regression: regularization (`C` parameter in `sklearn`), class weights, etc.
  - SVM: kernel type, kernel parameters (like polynomial degree), error penalty (`C` again), etc.
- Question is often ***how*** we can tune these model control parameters effectively to find best combinations

13

Review: Heldout Cross-Validation

- We can use k-fold cross-validation techniques to estimate the real effectiveness of various parameter settings:

1. Divide labeled data into k folds, each of size $1/k$
2. Repeat k times:
   a. Hold aside one of the folds; train on the remaining $(k-1)$; test on the heldout data
   b. Record classification error for both training and heldout data
3. Average over the k trials

- This can give us a more robust estimate of real effectiveness
- It can also allow us to better detect over-fitting: when average heldout error is significantly worse than average training error, model has grown too complex or otherwise problematic

14

Modifying Model Parameters

- Using heldout validation techniques, we can begin to explore various parts of the hyperparameter-space
  - In each case, we try to maximize average performance on the heldout validation data
- For example: ***number*** of layers in a neural network can be explored iteratively, starting with one layer, and increasing one at a time (up to some reasonable) limit until over-fitting is detected
- Similarly, we can explore a range of layer ***sizes***, starting with hidden layers of size equal to the number of input features, and increasing in some logarithmic manner until over-fitting occurs, or some practical limits reach

15

Using Grid Search for Tuning

- One basic technique is to list out the different values of each parameter that we want to test, and systematically try different combinations of those values
  - For P distinct tuning parameters, defines a P-dimensional space (or "grid"), that we can explore, one combination at a time
- In many cases, since building, training, and testing the models for each combination all take some time, we may find that there are far too many such combinations to try
  - One possibility: many such models can be explored in ***parallel***, allowing large numbers of combinations to be compared at the same time, given sufficient resources

16

Costs of Grid Search

- When we have large numbers of combinations of possible parameters, we may decide to limit the range of some of the parts of our "grid" for feasibility
- For example, we might try:
  1. # Hidden layers: 1, 2, …, 10
  2. Layer size: N, 2N, 5N, 10N, 20N (N: # input features)
  3. Activation: Sigmoid, ReLU, tanh
  4. Regularization (`alpha`): 10^{-5}, 10^{-3}, 10^{-1}, 10^{1}, 10^{3}
- Produces (10 x 5 x 3 x 5) = 750 different models
  - If we are doing 10-fold validation, need to run 7,500 total tests
  - Still only a small fragment of the possible parameter-space

17

Random Search

- Instead of limiting our grid even further, or trying to spend even more time on more combinations, we might try to ***randomize*** the process
- Instead of limiting values, we choose randomly from any of a (larger) range of values:
  1. # Hidden layers: [1, 20]
  2. Layer size: [8, 1024]
  3. Activation: [Sigmoid, ReLU, tanh]
  4. Regularization (`alpha`): $[10^{-7}, 10^{7}]$
- For each of these, we assign a probability distribution over its values (uniform or otherwise)
  - We may presume these distributions are independent of one another
- For T tests, we sample each of the ranges for ***one*** possible value, giving us T different combinations of those values

18

Performance of Random Search

- This technique can sometimes out-perform grid search
  - When using a grid, it is sometimes possible that we just ***miss*** some intermediate, and important, value completely
  - The random approach can often hit upon the better combinations with the same (or far less) testing involved

J. Bergstra & Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research* 13 (2012).

- Compare random search to grid search over 100 possible network hyperparameter configurations
- As few as 8 randomly selected configurations showed statistically significant improvement in model performance

19