Dec 4 Perceptrons

tentative final grades ~ eve of Monday 14th

Practicum Review Materials:

1. See 8tile.ipynb on the course schedule for a working example of an `adjacent_states` setup to a search/A^* problem.
2. See solutions to HW4 for a working example of an MDP setup, but be warned that value iteration might be very slow on a large state space!
3. See the Qlearning in-class notebook - and I'll post the one with `gym` syntax today - for some ideas on how to set up dictionaries for the Q-learning updates, which otherwise should borrow *heavily* from the class/functions you use for the MDP.

Exam Review Materials:

1. Review in class Monday
2. See the overview of concepts and the past exam with solutions on the associated course canvas module.

Supervised NLP

Suppose we are provided with a set of N documents that have been pre-labelled with classes, so we have a set of tuples $(document_i, class_i)$. Our goal might be to:

1. Describe the *features* (words, sentences, logical thoughts) that make a document *more likely* to belong to a specific class
2. Be able to assign a probability to a *new* or unobserved theoretical document belonging to the classes in the original corpus.

Definition: A *generative classifier* like the Naive Bayes spam filter builds a model of how a class could generate input data.

Definition: A *discriminative classifier* like the logistic regression model learns what features from the input are most useful to differentiate between classes.

NLP Logic

For discussion of implementation, efficiency, etc. you can find more examples in chapter 7-9 of Norvig. We discussed 3 options:

1. Forward-chaining logic: what conclusions flow from the logic inside your knowledge base? Is the desired query there?
2. Backward-chaining logic: what equivalences *would* prove the desired query? Are those things inside our knowledge base?
3. Resolution: assume the thing you want to disprove, generate a logical contradiction from things inside the knowledge base.

We haven't touched on the next major issue, which is how to teach an AI to correctly parse prose for the appropriate conjunctives or logicals. For this, we likely need access to a *dictionary* wherein the computer understands which terms are the verbs/adjectives for relations and which words are the nouns to be subjects/objects of those relations.

Extraction

In chapter 22, Norvig covers some techniques in *extraction*. We provide our reader with a background set of groupings. The FASTUS model classifies words in a sentence as:

1. NG: noun group
2. VG: verb group
3. CJ: conjunctions such as and/or **or** logicals that can be rewritten as conjunctions
4. PR: prepositions

Such a classification is *finite-state*, and probably won't pick up nuance very well. But it can start to understand texts!

Extraction

So we feed a test-reader a sentence or thought, and it will:

1. Tokenize the characters into tokens such as words, punctuation, numbers. This may be the same as shingling on words (1-grams).
2. Handle complex words: group tokens like Firstname Lastname, "set up," "exploratory analysis" into *single* NG or VG objects.
3. Handle basic groups: assign the NG/VG/CJ/PR statuses.
4. Handle complex phrases: create or classify NG/VG pairs, creating the relations like $loves(Beagles, Zach)$.
5. Merge Structures: use additional sentences or prepositions to glue together sequential and related thoughts.

Noise and Extraction

The model outlined before is a simple finite-state model approach, and may not work well on varied or noisy inputs. More complex models encode language as a Markov model or HMM. One task might be to create a language HMM that to tries to read a course description to figure out how much math is needed.

1. The HMM may look for sequences of words such as "requires (a background in)..." and then a set of trained keywords.
2. Words like "familiarity," "rigor," "proof" adjust the underlying probabilities of how much knowledge we might need.

Perceptrons

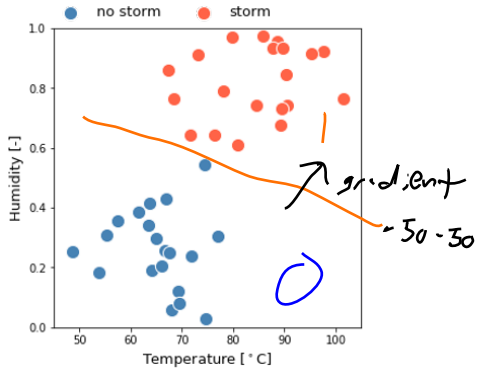
Now begins our brief discussion of what many modern computational data scientists consider the center of machine learning: the neural network.

At their core, neural networks arise from a problem *very* similar to Logistic Regression!

Goal: Predict Was there a storm?
(if storm then, $Y = 1$)
Use: Features X_1 :=Temp, X_2 :=humidity

In Logistic Regression, we used the model:

$$P(\text{Storm}) = \text{sigm}(\beta_0 + \beta_1 X_1 + \beta_2 X_2)$$



Gradient Ascent/Descent

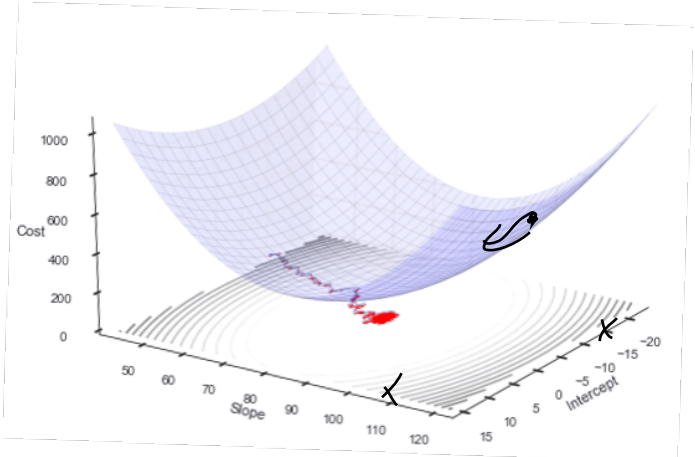
The idea of following the derivative to find the maximum or minimum value of a function is called gradient ascent. It requires:

1. The ability to calculate the slope of the function we're trying to min/max
2. An idea of how large of steps to take; a step size or *learning rate*

$$\underbrace{z^{(k+1)}}_{\text{new guess}} = \underbrace{z^{(k)}}_{\text{old guess}} - \underbrace{\nu}_{\text{step size}} \underbrace{f'(z^{(k)})}_{\text{step direction}}$$

Gradients in Regression

In something like SLR, we have two parameters to estimate, β_0 and β_1. This means our f function is a three-dimensional surface: it has a β_0 and a β_1 axis and heights given by the SSE.



Cost or error (squared)

SSE:

$$\sum_{\text{data}} (\text{estimate}_y - \text{true } y)^2$$

Gradients

Both Perceptrons and Logistic Regression perform their actual estimation of the w or β values via gradients. For simple linear regression, we are trying to minimize squared errors, and have to differentiate

$$\text{SSE}(\beta_0, \beta_1) = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_i))^2$$

A partial derivative takes this derivative while treating all other variables as constants. So we take a derivative w.r.t. β_0 and one w.r.t. β_1:

$$\frac{\partial SSE}{\partial \beta_0} = -2 \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_i))$$

$$\frac{\partial SSE}{\partial \beta_1} = -2 \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_i)) \, x_i$$

Gradients

For logistic regression, we are *still* trying to minimize squared errors, but classically we rewrite the problem to be one based on log-likelihood (LogLL) functions. If you aren't familiar with likelihood, it's the probability of observing the data *given* the model, or $P(classification|features, model)$

The partial derivatives of that are... basically the same thing as in simple linear regression, with 100% more sigmoids!

over all data

$$\frac{\partial LogLL}{\partial \beta_0} = \sum_{i=1}^{n} (y_i - \text{sigm}(\beta_0 + \beta_1 x_i))$$

$$\frac{\partial LogLL}{\partial \beta_1} = \sum_{i=1}^{n} [y_i - \text{sigm}(\beta_0 + \beta_1 x_i)] \, x_i$$

$\partial/\partial\beta_2$ $\partial/\partial\beta_3$ .. for each feature.

Gradient Ascent

This provides us with the schema for *gradient-based* ascent or updating. Instead of trying to update the sums all at once, we can *incrementally* add small amounts to our estimates β_0 or β_1 according to the derivative(s) we took!

So the multiple sums $\sum_{i=1}^{n}$ becomes an *iteration* scheme. Pick a point, and then add the effects of that point into the estimates needed for each coefficient.

For each data point (in random order?!), do:

deriv of features

update features:

$$\beta_0 \leftarrow \beta_0 - \nu \cdot (-2) \cdot [y_i - \text{sigm}\,(\beta_0 + \beta_1 x_i)]$$
$$\beta_1 \leftarrow \beta_1 - \nu \cdot (-2) \cdot [y_i - \text{sigm}\,(\beta_0 + \beta_1 x_i)]\, x_i$$

Then repeat for lots of epochs until training is complete!

Perceptrons Notation

The goal of prediction/classification and features used for a Perceptron are identical to those in Logistic Regression. The underlying notation changes a little, and we may *drop* the concept of probability.

1. In a perceptron we drop the β_i notation that was used for linear regression coefficients and instead use *weights* w_i for the "importance" of predictor. Crucially, the model is still made up of multiplying these weights by each feature ina **linear** matter.
2. These weights are often written as a dot product.
3. We aren't required to use the sigmoid, since we're not using probabilities. Instead we can take *any* function, as long as we make a decision rule that tells us when to classify "1" versus "0."

Perceptron Weights

So our model has changed from the logistic sigmoid

$$P(\text{Storm}) = \text{sigm}(\beta_0 + \beta_1 X_1 + \beta_2 X_2)$$

into one with *weights* and some output function z

$$z = w_0 \cdot (1) + w_1 \cdot X_1 + w_2 \cdot X_2$$

What was β_0 represented the *intercept* of the logistic regression, and w_0 still represents that value, but in ML we often call the intercept term a *bias*, since it works apart from the features X_1 and X_2.

Additionally, we may not report the linear surface and instead report *only* the classification. Instead of a plane or a line, we'll get a *jump* or step.

Perceptron Activations Functions

The decision to take the linear surface or **dot product** between vectors for $\boldsymbol{w}$ and $\boldsymbol{x}$

$$z = w_0 \cdot (1) + w_1 \cdot X_1 + w_2 \cdot X_2 = \sum_{j=0}^{n} x_j w_j = \boldsymbol{w}^T \cdot \boldsymbol{x}$$

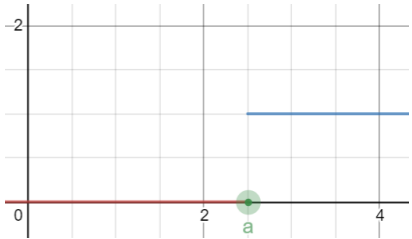
and instead of z use a step function is called the *activation* function for the perceptron.

Definition: the *Heaviside* step function is:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

Perceptron Activations

$$f(x) = \begin{cases} 0 & \text{if } x < \theta \\ 1 & \text{if } x \geq \theta \end{cases}$$



...which is to say that we take the $z = \boldsymbol{w}^T \cdot \boldsymbol{x}$ and instead turn it into boolean or binary output of 0 or 1, based on whether it passes a *threshold* or *activation* value.

Perceptron Training

We still need to estimate the weights w of our linear surface.

$$z = w_0 \cdot (1) + w_1 \cdot X_1 + w_2 \cdot X_2$$

We can again set up the idea of a classification *error* of our predictions z *given* some weights: $Error_i = (Data_i - z_i)$ ... but what we want to do is update the *weights*.

We do this by something like gradient ascent! To update *weight* j, take the error of each data point times the *feature* j. This is like the "weight error for j," so we update that with a **learning rate** and perform iterations.

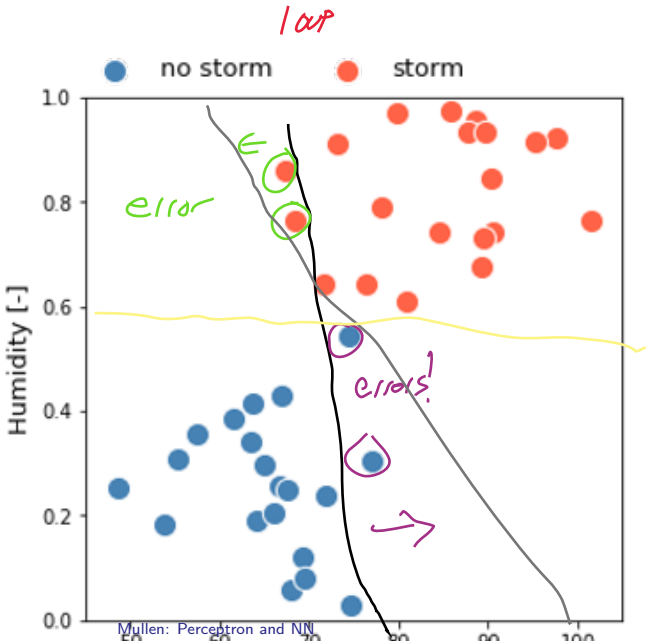
So we again loop over all the points i, and for each update the weights of each feature.

$$w_{j,new} = w_{j,old} + \nu \underbrace{(y_i - z_i)}_{error: 0\, or \pm 1} x_j$$

Picturing Training

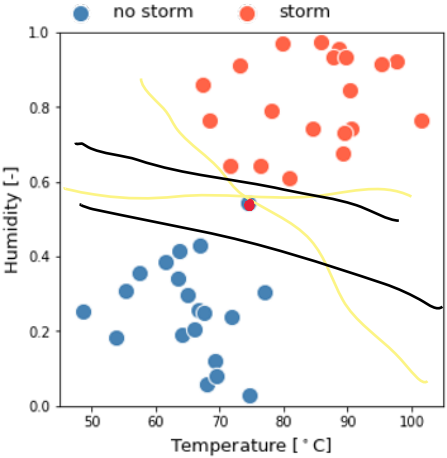
Suppose we start with some random or initialized weights that aren't quite right.

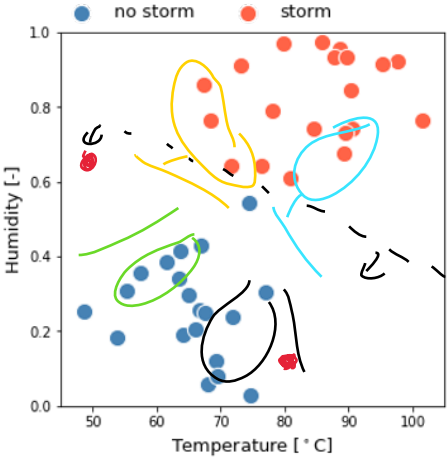
Over time, we'll have lots of little $+1$ and -1s *at the misclassified points* until our classification line nudges in the correct direction. (by feature-updating steps proportional to size of learning rate)



Picturing Training

What if there isn't only one answer?





Jumps are nasty!

This should raise a few red flags.

1. How are we taking *gradients* when our x values are living inside a step function, which always has slope 1? The updates might make sense, but the calculus doesn't!
2. What happens if the two data sets *aren't cleanly separated?* **Result:** the step-function perceptron may fail to converge and never stop updating weights if it continually observed *classification error*.

So what is actually done in practice? We bring back real calculus! A *continuous* function can be used for the z values, so the model becomes

$$g(z) = g\left(w_0 \cdot (1) + w_1 \cdot X_1 + w_2 \cdot X_2\right)$$

where the linear piece z is plugged into some function g.

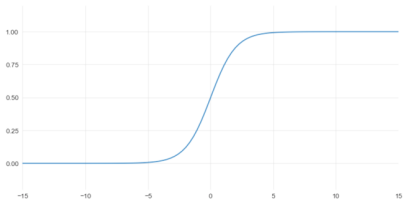
Jumps are nasty!

Why activation functions?

1. This can ensure a unique solution, since the optimization metric we'll use will feed error from g back into the model. The **final** classification might still use a *quantizer* like the Heaviside function, but the model will be tuned continuously.
2. *Gradient* methods relied on the derivative of the error function $y_i - z_i = y_i - (w^T x)$. If *instead* we differentiate $y_i - g(w^T x)$, it will change our updates, because we'll have to use the differentiation chain rule on g.
3. As a result, control over the *shape* of the activation function lets us change the structure of the gradient updates.

Activation Functions: Sigmoid

$$\text{sigm}(z) = \frac{1}{1 + e^{-z}}$$

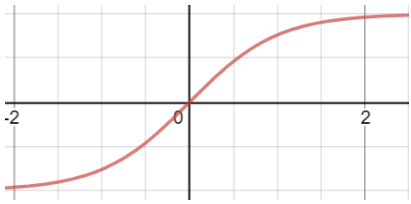


The sigmoid function happens to have a nice property that $f'(z) = f(z)(1 - f(z))$, which can make for very useful calculus!

Of course, if we use the sigmoid we're minimizing error over sigm $\left(w^T x\right)$... which is **the same classifier** as logistic regression.

Activation Functions: tanh

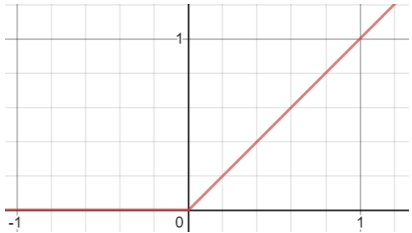
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



For the hyperbolic tangent function $f'(x) = (1 - f(x)^2)$.

Activation Functions: ReLU

$$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x > 0 \end{cases}$$



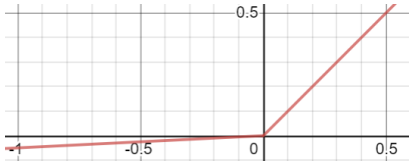
For the Relaxed Linear Unit, $f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$.

This makes for the nicest gradients yet, but leads to a new issue: having *no* slope for $x < 0$ can cause large regions of the data to provide zero updates. That's a little scary!

Activation Functions: Leaky ReLU

$$f'(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x > 0 \end{cases}$$

where α is small.



For the Leaky Relaxed Linear Unit, $f'(x) = \begin{cases} \alpha & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$.

This is one of the more popular recent functions, but plenty of work is still continuing on things that make the gradient updates **faster**, and **more consistent**.

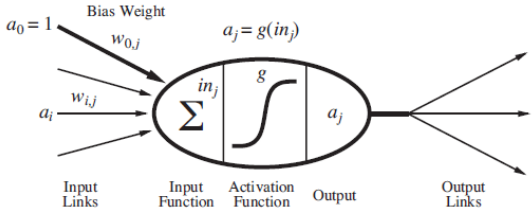


Figure 18.19 A simple mathematical model for a neuron. The unit's output activation is $a_j = g(\sum_{i=0}^{n} w_{i,j} a_i)$, where a_i is the output activation of unit i and $w_{i,j}$ is the weight on the link from unit i to this unit.

Norvig's depiction of a perceptron node. Neural Networks take inspiration from biology, where each neuron fires an impulse if its *activated*. Connection a series or graph of neurons can form a network.

Neural Nets versus Perceptrons

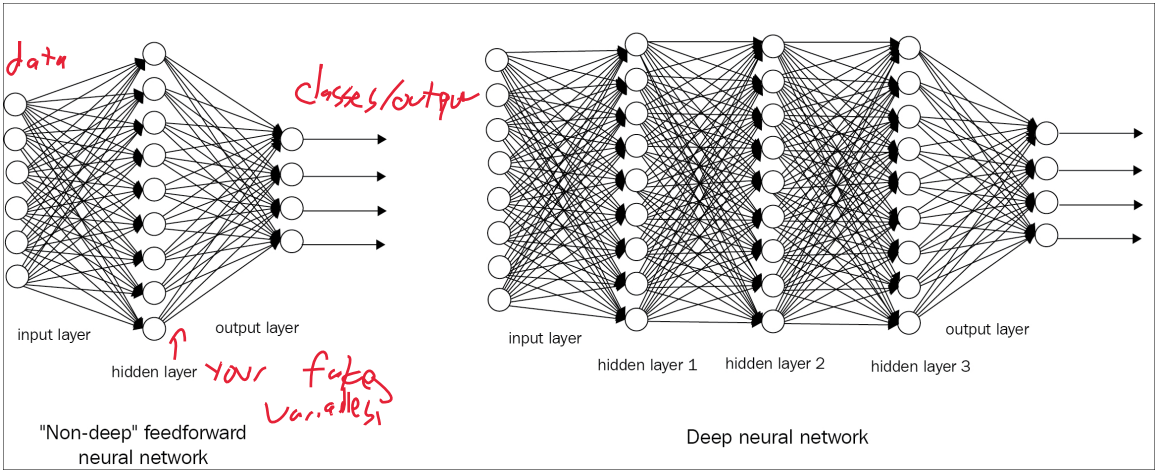
The first type of neural network is a *multilayer* perceptron. It's formed similarly to a Hidden Markov Model. We allow for a set of intermediate *unobserved* values or nodes, each of which will receive weights from the data features x.

1. Instead of 1 output v, we estimate a whole list of them... which become a vector $v = w^t x$.
2. We can then take the v values, give them weights, and estimate $y = w_v^t v^t$
3. Repeat ad nauseam if we want, where each step may rely on the same underlying activation function and gradient ascents (first the x weights, then the v weights, then repeat).

Why do this? It sounds computationally expensive, since even computing the predictions and the errors require us pass every x to every v to see what the predicted classification is! That's more like order $n \cdot m$ than just order n!

Additionally, we need to be able to take the errors and measure the effects of the x-weights given the error, a problem known as **back-propogation**.

Neural Nets versus Perceptrons



www.packtpub.com/product/deep-learning-with-hadoop

Neural Nets have power

The payoff of the Neural net is that allowing even just one extra "layer" gives a powerful result: it's possible to estimate *any* true classification with a two-layer neural net.

1. **The Weierstrauss Approximation Theorem**: Let $f : [a, b] \to \mathbb{R}$ be continuous. Then for any $\varepsilon > 0$ there exists a polynomial p such that $|f(x) - p(x)| < \varepsilon$ for all $x \in [a, b]$.

   What does this mean? It means for for any continuous 1D function over a finite interval, you can - with high enough exponents - draw a polynomial that nearly perfectly copies that function. It turns out that we have a similar result that says that two layers on a neural network lets you capture similar **nonlinear** behavior!

2. **The Universal Approximation Theorem**: similar setup... then for any $\varepsilon > 0$ there is a function of the form

$$g(x) = \sum_{i=1}^{d} v_i \phi(w_i^T x_i + b_i)$$

   such that $|f(x) - p(x)| < \varepsilon$ for all $x \in [a, b]$

Moving Forward

In practice, people often use "deeper" neural nets with less layers rather than only one wide-but-shallow hidden layer. The algorithms to pass information from x through the hidden layers to the final quantizer and classifier are covered in brief in Norvig, ch. 18.
Take 4622 for more depth (I hope?)

- Coming up:
  1. Pen-and-paper review on Monday: quick worked examples for BN, HMM, MDPs, or whatever you have questions on!