DEEP LEARNING IN PYTHON

Outline

- Softmax: Going from binary classification to multi-class classification
- How to train a neural net: backpropagation
- XOR and Donut
  - I'll prove to you that neural networks can automatically learn discriminating features
- Numpy: Build neural networks by hand
- TensorFlow: Plug-n-play script
- TensorFlow Playground: Visualize what a neural network is learning
- More projects:
  - Facial expression recognition

High-level view of ANY supervised learning problem

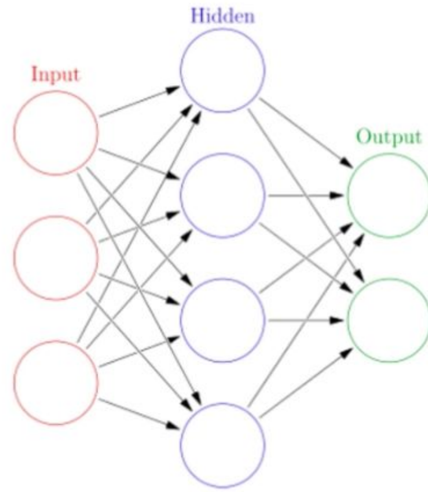
All models (logistic regression, k-nearest neighbor, Naive Bayes, SVM, decision tree, neural nets) have the same 2 functions:

train() - learn model params from the data

predict() - make accurate predictions using the params learned during training

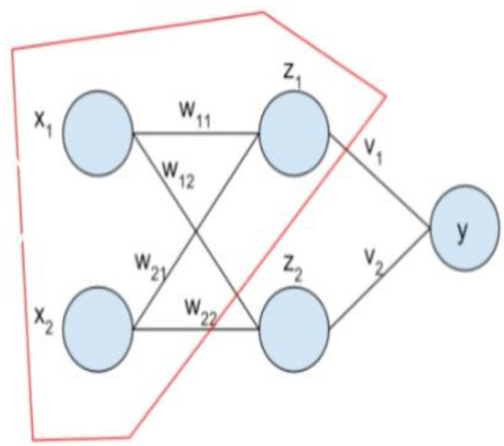
Architecture

- one or more hidden layers
- every node in one layer is connected to every node in the next layer
- signals get transmitted from the input, to the hidden layer, to the output
- the output is aiming for a target



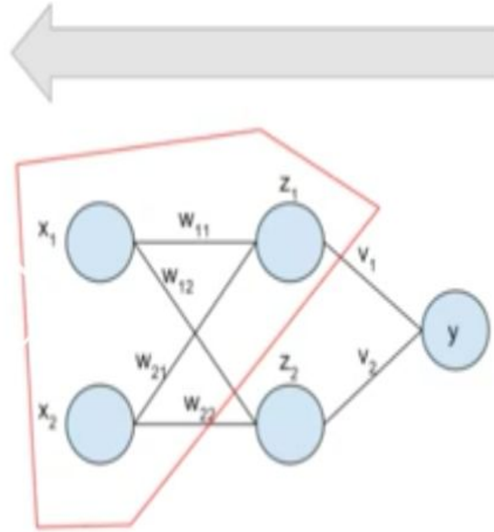
Neural networks are networks of neurons

- Logistic unit is a neuron (one is shown in red, can you spot the other 2?)
- A neural network is just layers of logistic regression units



Learning (Backpropagation)

- backpropagation
- the error gets "propagated" backwards
  - V depends on error at Y
  - W depends on error at Z
  - Same pattern if more layers
- the weights get updated based on this propagated error

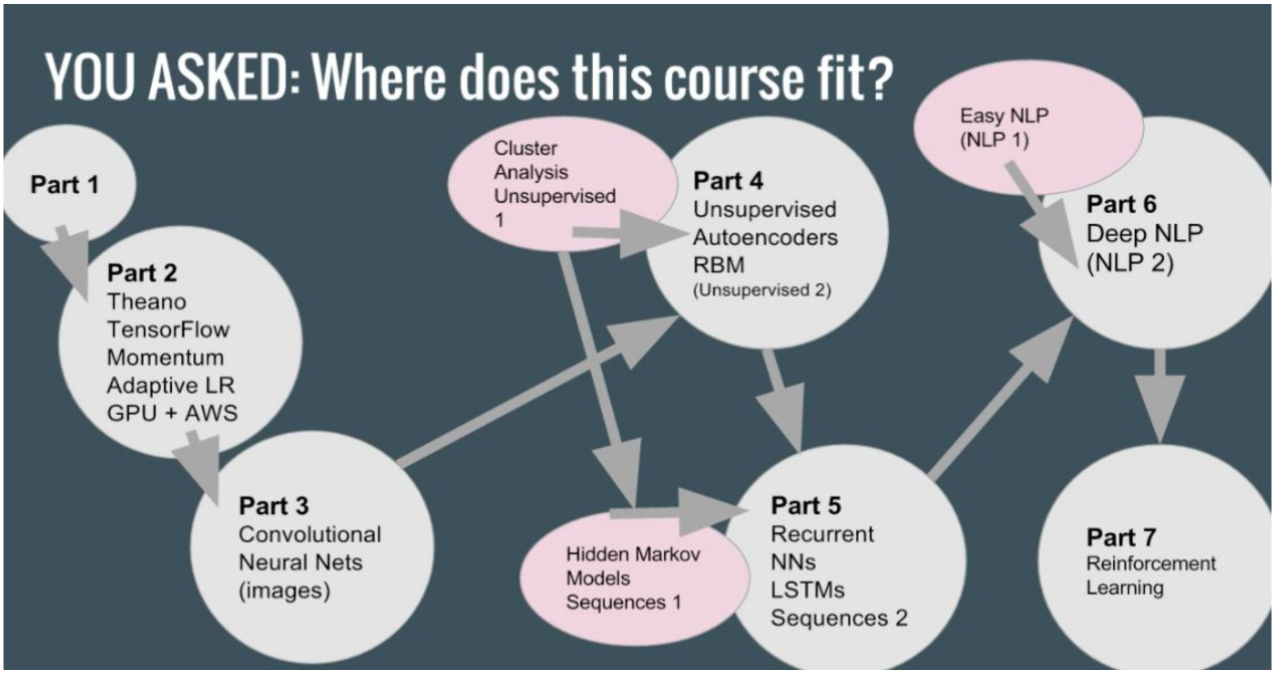


YOU ASKED: Where does this course fit?

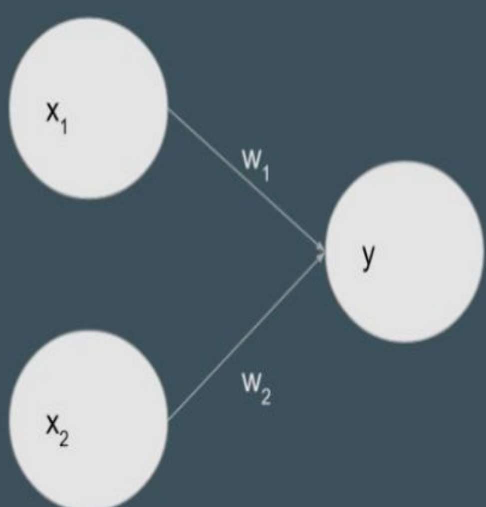
Linear Regression (part "-1")
$y = Wx + b$
→ Used everywhere
→ Closed-form solution
→ Data pre-processing

Logistic Regression (part 0)
$y = \sigma(Wx + b)$
→ No closed-form solution, use gradient descent
→ Regularization
→ Biological neurons

This class! (part 1)
Binary → Multi-class
→ Backpropagation
→ Automatically learning features



Logistic Regression



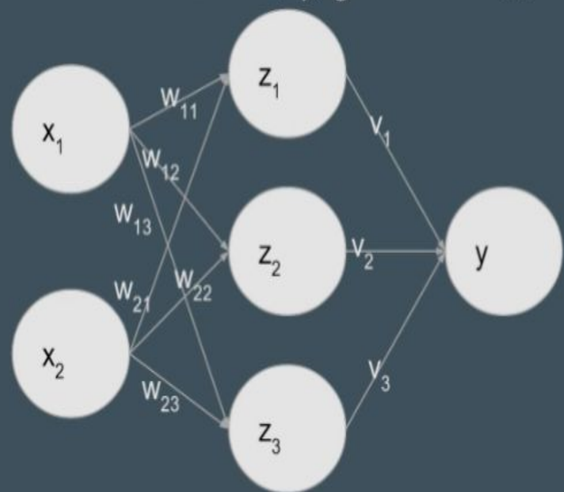
$$a = x_1 w_1 + x_2 w_2 + b$$

$$p(y|x) = 1 / (1 + e^{-a})$$

$$\begin{aligned} \text{prediction} &= \text{round}(p(y|x)) \\ &= 1 \text{ if } p(y|x) > 0.5, \text{ else } 0 \end{aligned}$$

Neural Networks (Feedforward)

w's are hard to see: W(i,j) goes from x(i) to z(j)



$z_j = \text{sigmoid}(\sum_i(W_{ij}x_i) + b_j)$

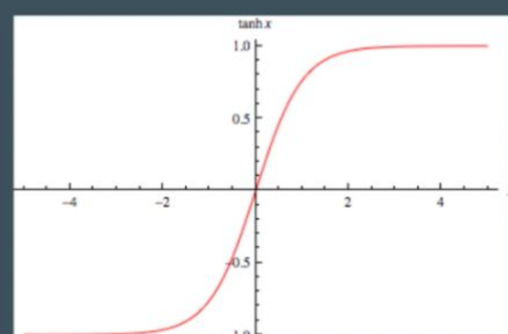
$p(y|x) = \text{sigmoid}(\sum_j(v_j z_j) + c)$

$i = 1,2$
$J = 1,2,3$

Nonlinearities

$\text{Tanh}(x) = (e^x - e^{-x}) / (e^x + e^{-x})$

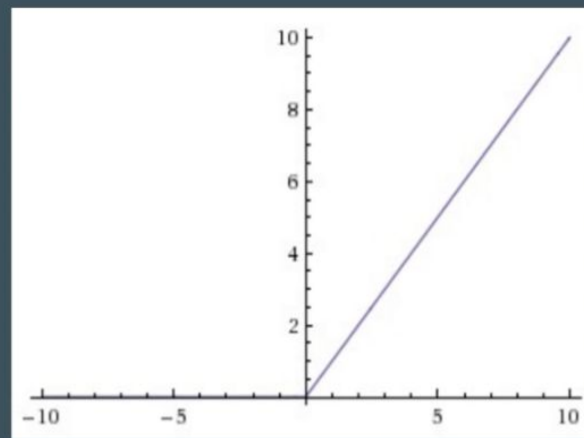
Exercise: show relationship between tanh and sigmoid



Nonlinearities

relu(x) = max(0, x)

Rectifier Linear Unit



Example

One hidden layer, 2 hidden units

x = [0, 1]
W(1,1) = W(1,2) = W(2,1) = W(2,2) = 1
v(1) = v(2) = 1
b = c = 0

Solution:

z(1) = sigmoid(0*1 + 1*1) = 0.731, z(2) = sigmoid(0*1 + 1*1) = 0.731
p(y|x) = sigmoid(0.731*1 + 0.731*1) = 0.812

Later: How do we choose W, V, b, c so that our predictions are accurate?

Vector Notation

Numpy operations are more efficient than Python for loops

$z_j = \text{sigmoid}(\sum_i(W_{ij}x_i) + b_j) \rightarrow z_j = \text{sigmoid}(W_j^T x + b_j) \rightarrow z = \text{sigmoid}(W^T x + b)$

$p(y|x) = \text{sigmoid}(\sum_j(v_j z_j) + c) \rightarrow p(y|x) = \text{sigmoid}(v^T z + c)$

x is a D-dimensional vector (D=2 in the previous image)

z is an M-dimensional vector (M=3 in the previous image)

Matrix Notation

We usually want to consider more than one sample at a time.

It's even more efficient to process the multiple samples simultaneously

- X is an NxD matrix (N = number of samples)
- Z is an NxM matrix
- p(Y|X) (sometimes just called "Y") is an Nx1 matrix (for binary classification, which is what we're currently doing) (for K classes, it'll be NxK)
- W is DxM, b is Mx1, v is Mx1, c is a scalar (1x1)

Z = sigmoid(XW + b)
Y = sigmoid(Zv + c)

Binary classification

Examples:

- (humidity, ground is wet, month, location) → (rain, no rain)
- (exercise frequency, age, BMI, nutrient intake) → (disease, no disease)

Yes / No

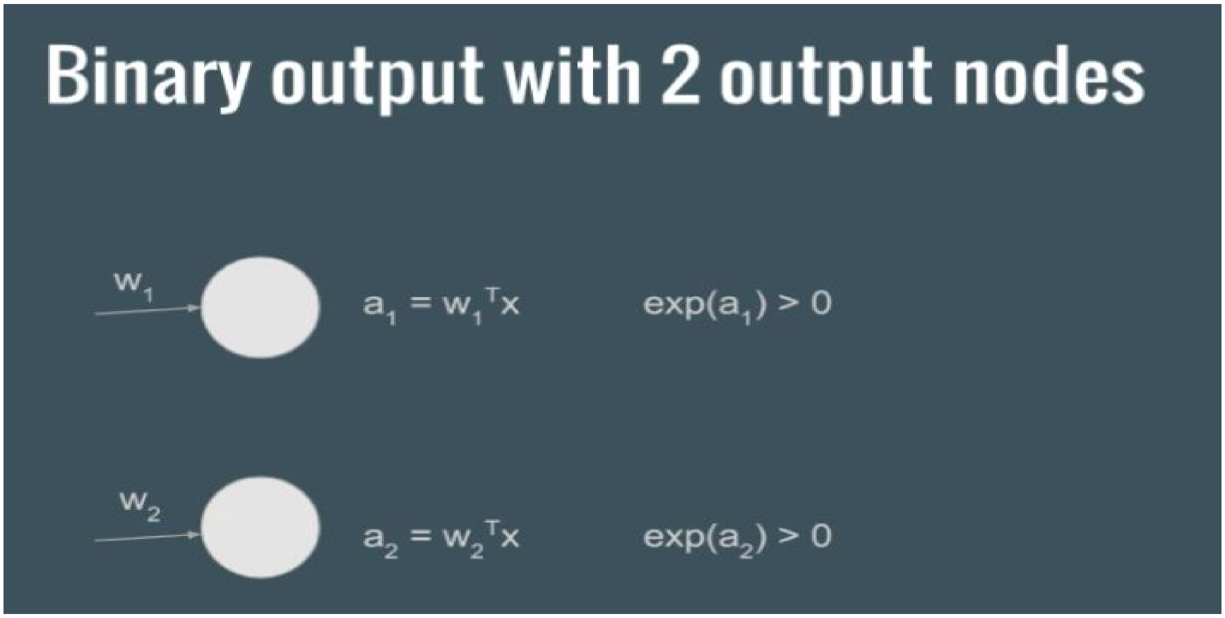
K Classes

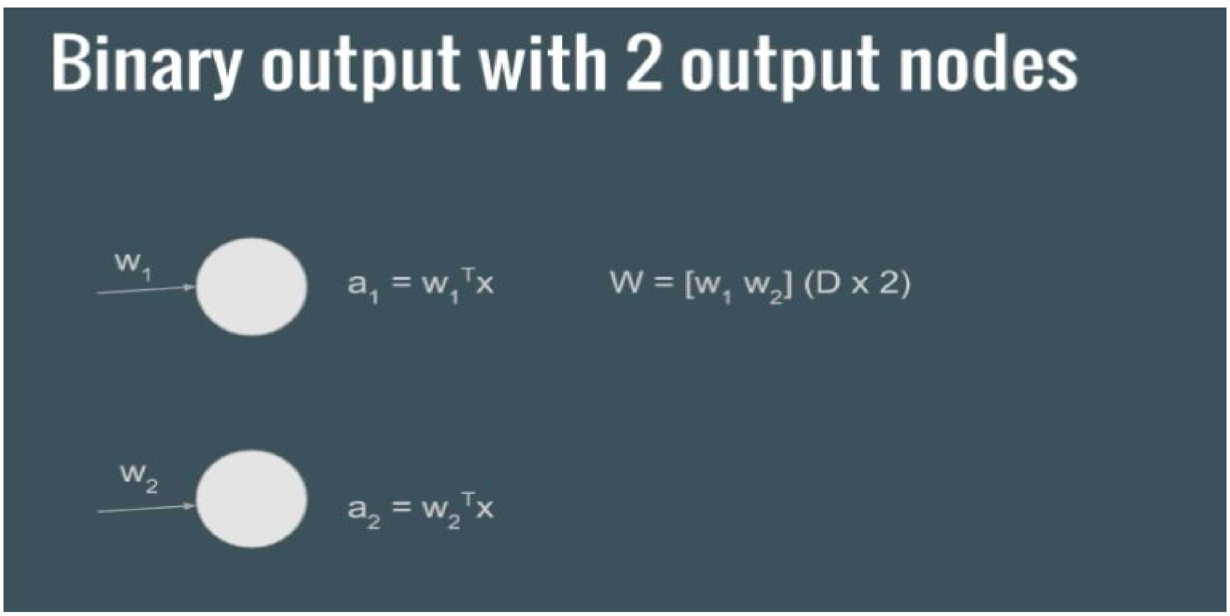
Facebook Images (Advertising applications):

- Faces
- Cars
- Wedding dresses
- Environment

MNIST:

- Digits between 0-9


Binary output with 2 output nodes
w_1
$a_1 = w_1^T x$
$\exp(a_1) > 0$
w_2
$a_2 = w_2^T x$
$\exp(a_2) > 0$


Binary output with 2 output nodes
w_1
$a_1 = w_1^T x$
$W = [w_1\ w_2]\ (D \times 2)$
w_2
$a_2 = w_2^T x$

Softmax for K Classes

$P(Y = k \mid X) = \exp(a_k) / Z$

$Z = \exp(a_1) + \exp(a_2) + \ldots + \exp(a_K)$

$W = [w_1\ w_2 \ldots w_K]$ (a D x K matrix)

Vectorized logistic regression with softmax:

$A_{NxK} = X_{NxD} W_{DxK} \rightarrow Y_{NxK} = \text{softmax}(A_{NxK})$

Sigmoid vs. Softmax

When are they equivalent?

$p(Y=1 \mid X) = \exp(w_1^T x) / [\exp(w_1^T x) + \exp(w_0^T x)]$

$p(Y=0 \mid X) = \exp(w_0^T x) / [\exp(w_1^T x) + \exp(w_0^T x)]$

...

Divide top and bottom of $p(Y=1|X)$ by $\exp(w_1^T x)$

Sigmoid vs. Softmax

But if you are trying to build a more general classifier, one that can handle anything... then softmax is a safer choice because it works for K=2 AND K>2

Training a Neural Network

- To refresh your memory, there are 2 main functions in a ML model:
  - train(X, Y)
  - predict(X)
- Previous section covered prediction
- Input: data X, Output: probability the data belongs to each output category
- None of those predictions made sense because the weights were random
- In this section, we answer the question:
- "What should we set the weights to?"

The Main Concepts

- We very intuitively define something called the "cost"
- Like any good businessman, we desire to minimize the "cost"
- How?
- Falls into the domain of calculus - calculus provides tools to find the min/max of a function
- We use a method called "gradient descent"
- Again, first presented in Logistic Regression, but I include a simple example in the Appendix

How do we define cost?

- For binary classification, exactly like how we calculate the likelihood of a coin toss
- Ex. We flip 2 heads, 3 tails
- Likelihood = p(H)p(H)p(T)p(T)p(T)
- We multiply because each toss is independent
- We can also say p = p(H), then:
- Likelihood = $p^{\#H}(1-p)^{\#T}$

Minimize or maximize?

- With likelihood, we want to find a "p" that maximizes the likelihood
- But we want to minimize "cost"
- So we take the negative log of the likelihood and call that the "cost"
- Negative log likelihood = $-\{ \#H \log p + \#T \log(1-p) \}$
- We call this the "cross-entropy cost function"

Cross-Entropy

y_n = output of logistic regression or neural network

t_n = actual target (0 or 1)

$$cost = J = -\sum_{n=1}^{N} t_n log(y_n) + (1 - t_n) log(1 - y_n)$$

- Recall: to find the best weights to minimize this cost, we use "gradient descent"
- We can also maximize the negative of this ("gradient ascent")

Cross Entropy for Multi-class Classification

- In this class, we want to handle any number of outputs
- Let's consider a die roll (6 faces, but let's call it K)
- y_k = probability of output being k
- t_k = 1 if we roll k, 0 otherwise
- N total die rolls, so $t_{n,k}$ = 1 if we rolled k on the nth die roll
- Therefore, only ONE of the $t_{n,k}$ can be 1 for any particular n
    - $t_{n,k}$ is thus an "indicator matrix" or "one-hot encoded" matrix of 1s and 0s

$$likelihood = \prod_{n=1}^{N} \prod_{k=1}^{K} y_{n,k}^{t_{n,k}}$$

Cross Entropy for Multi-class Classification

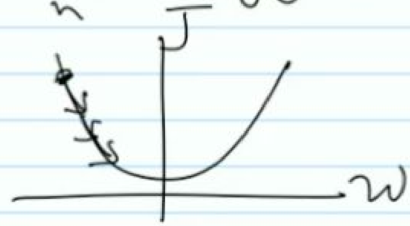
$$cost = J = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k} \log y_{n,k}$$

How to minimize the cost

- Next, we'll see how we perform gradient descent on the new cost function and how to write it in code
- People underestimate the importance of this: "TensorFlow will already do it for us"
- You will see how later in the course
- Researchers spent decades trying to find this algorithm
- It's the "secret sauce" of neural networks
- Same thing will be used no matter how complex the architecture: convolutional nets, recurrent nets, autoencoders, etc.
- All with this one algorithm

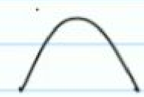
TRAINING A NEURAL NETWORK

$$J = -\sum_n t_n \log y_n + (1 - t_n)\log(1 - y_n)$$



$$w \leftarrow w - \eta \frac{\partial J}{\partial w}$$

small number



$$w \leftarrow w + \eta \frac{\partial L}{\partial w}$$

TOTAL DERIVATIVES

$f(x, y)$ $\quad x(t) \quad y(t)$

$$\frac{df}{dt} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

TOTAL DERIVATIVES

$f(x, y) \qquad x(t) \quad y(t)$

$$\frac{df}{dt} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

$x_k(t)$

$$\frac{df}{dt} = \sum_k \frac{\partial f}{\partial x_k} \cdot$$

TOTAL DERIVATIVES

$f(x, y) \qquad x(t) \quad y(t)$

$$\frac{df}{dt} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

$x_k(t)$

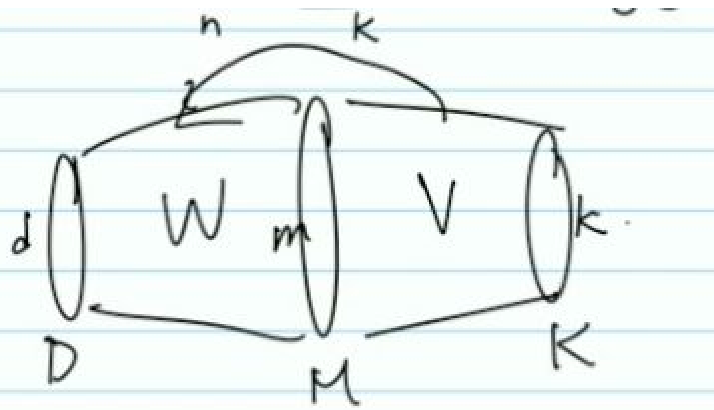
$$\frac{df}{dt} = \sum_k \frac{\partial f}{\partial x_k}\frac{dx_k}{dt}$$

OBJECTIVE

$$P = \prod_{n=1}^{N} \prod_{k=1}^{6} \theta_k^{n\, t_k^n} \leftarrow \text{die}$$

$$P(Y|X) = \prod_{n=1}^{N} \prod_{k=1}^{K} y_k^{n\, t_k^n}.$$

$$L = \sum_n \sum_k t_k^n \log y_k^n$$



$$\frac{\partial J}{\partial V_{mk}} \qquad \frac{\partial J}{\partial W_{dm}}$$

1st　　2nd

$$\frac{\partial J}{\partial V_{mk}} \qquad \frac{\partial J}{\partial W_{dm}}$$

1st 2nd

$$\frac{\partial J}{\partial V_{mk}} = \sum_n \sum_{k'} t^n_{k'} \frac{1}{y^n_{k'}} \frac{\partial y^n_{k'}}{\partial V_{mk}}$$

$\neq$

$$y_k = \frac{e^{a_k}}{\sum_j e^{a_j}} \qquad a_k = V_k^T z$$

$$a_k = \sum_m V_{mk} z_m$$

$$\frac{\partial y_{k'}}{\partial a_k} = y_{k'}(1 - y_k) \quad \text{if } k = k'$$

$$\frac{\partial}{\partial V_{mk}} = \sum_n \sum_{K'} t_{k'} \, y_{k'}^{n} \, \underbrace{\frac{\partial}{\partial V_{mk}}}_{?}$$

$$y_k = \frac{e^{a_k}}{\sum_j e^{a_j}} \qquad a_k = V_k^T z$$

$$a_k = \sum_m V_{mk} z_m$$

$$\frac{\partial y_{k'}}{\partial a_k} = \begin{cases} y_{k'}(1 - y_k) & \text{if } k = k' \\ -y_{k'} y_k & \text{if } k \neq k' \end{cases}$$

$$\frac{\partial y_{k'}}{\partial a_k} = \begin{cases} y_{k'}(1 - y_k) & \text{if } k = k' \\ -y_{k'} y_k & \text{if } k \neq k' \end{cases}$$

Kronecker delta

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & i \neq j \end{cases}$$

$$\frac{\partial y_{k'}}{\partial a_k} = y_{k'}(\delta_{kk'} - y_k)$$

$$\frac{\partial y_{k'}}{\partial a_k} = y_{k'}(\delta_{kk'} - y_k)$$

$$\frac{\partial a_k}{\partial V_{mk}} = z_m$$

Combine:

$$\frac{\partial J}{\partial V_{mk}} = \sum_n (t_k^n - y_k^n) z_m^n$$

Combine:

$$\frac{\partial J}{\partial V_{mk}} = \sum_n (t_k^n - y_k^n) z_m^n$$

$$\frac{\partial J}{\partial W_{dm}} = \sum_n \overbrace{\sum_k (t_k^n - y_k^n) V_{mk}}^{\partial J / \partial z_m} z_m^n (1 - z_m^n) x_d^n$$

k is not on the left

"total derivative"

∂V_{mk}

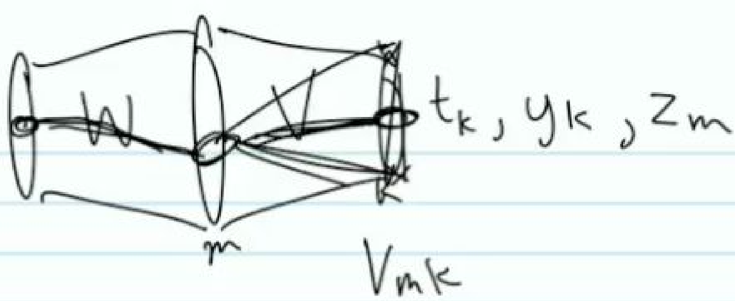
Combine:

$$\frac{\partial J}{\partial V_{mk}} = \sum_n (t_k^n - y_k^n) z_m^n$$

$$\frac{\partial J}{\partial W_{dm}} = \sum_n \overbrace{\sum_k (t_k^n - y_k^n) V_{mk}}^{\partial J / \partial z_m} \overbrace{z_m^n (1 - z_m^n) x_d^n}^{\partial z_m / \partial W_{dm}}$$

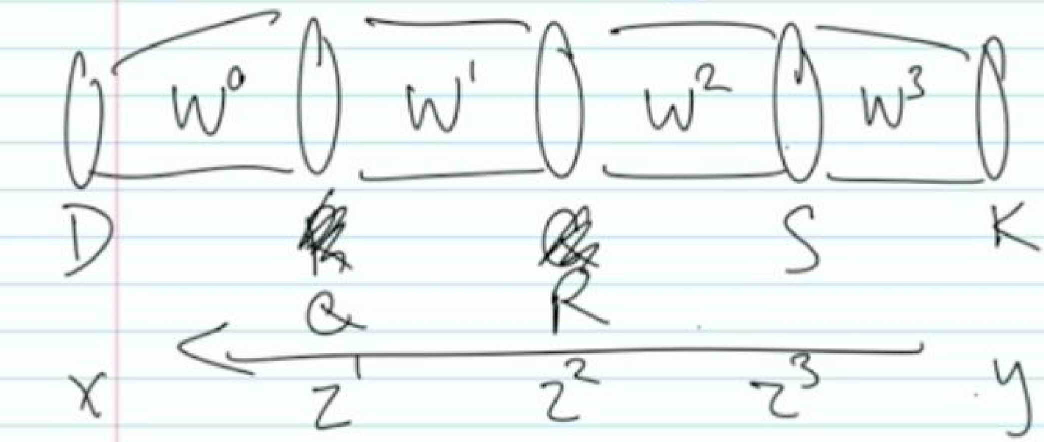
k is not on the left
"total derivative"

$$z_m = \sigma(W_m^T x)$$



$$W_{dm} = \sum_k (t_k, y_k, V, z_m, x_d)$$

DEEPER NETWORKS



$$\frac{\partial J}{\partial W^3_{sk}} = (t_k - y_k)\, z^3_s$$

$$\frac{\partial J}{\partial W^3_{sk}} = (t_k - y_k)\, z^3_s$$

$$\frac{\partial J}{\partial W^2_{rs}} = \sum_k \underbrace{(t_k - y_k)} W^3_{sk}\, z^3_s (1 - z^3_s)\, z^2_r$$

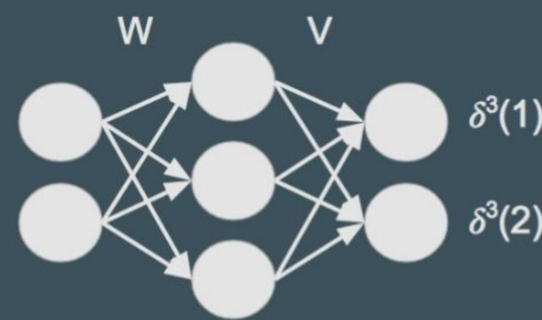
$$\frac{\partial J}{\partial W^1_{qr}} = \sum_k \sum_s (t_k - y_k) W^3_{sk}\, z^3_s (1 - z^3_s) W^2_{rs}\, z^2_r (1 - z^2_r)\, z^1_q$$

$$\frac{\partial J}{\partial W^1_{qr}} = \sum_k \sum_s (t_k - y_k) W^3_{sk} z^3_s (1 - z^3_s) W^2_{rs} z^2_r (1 - z^2_r) z^1_q$$

The Typical Approach

- $\delta^3(k) = y(k) - t(k)$
  = "error" at that node
- Why is this the error?
- Why not some other expression?

- $\Delta V(j,k) = z(j)\, \delta^3(k)$
- Then we say:
- $V(j,k) = V(j,k) - \eta \Delta V(j,k)$
- Why?

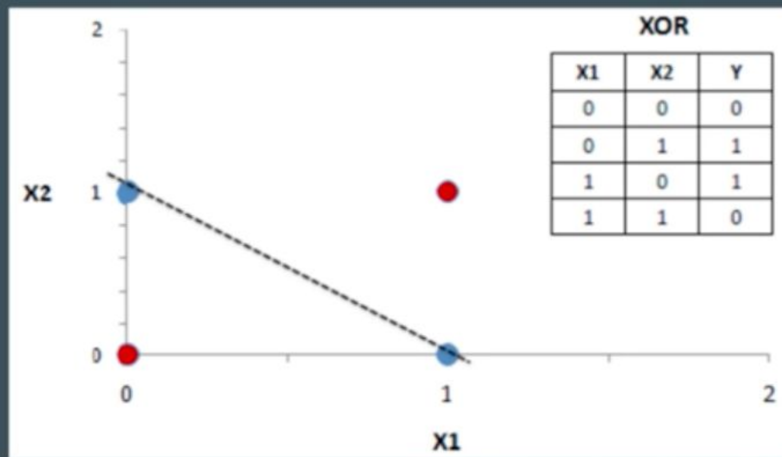


The Typical Approach

- That's the opposite of what you should do!
- Basic facts:
- We want to maximize likelihood, or equivalently minimize negative log-likelihood
- We know gradient descent is a general all-purpose way to optimize anything
- We used it already:
- Logistic regression
- Quadratic (high school math!)
- Starting with this is the right approach
- The "delta" is useful, because we can define backpropagation recursively
- But the key is to expose the pattern yourself (using GD), and use substitution for delta

XOR

- Probably the canonical problem used in transitioning from logistic regression to neural nets

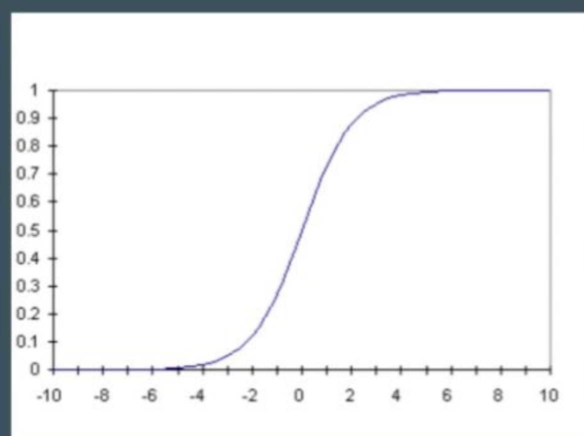


X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Sigmoid

$\sigma(x) = 1 / (1 + \exp(-x))$

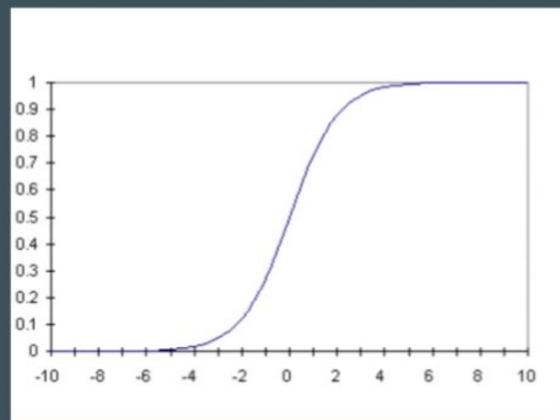
$\sigma'(x) = \sigma(x)(1 - \sigma(x))$



Sigmoid

$\sigma(x) = 1 / (1 + \exp(-x))$

$\sigma'(x) = \sigma(x)(1 - \sigma(x))$

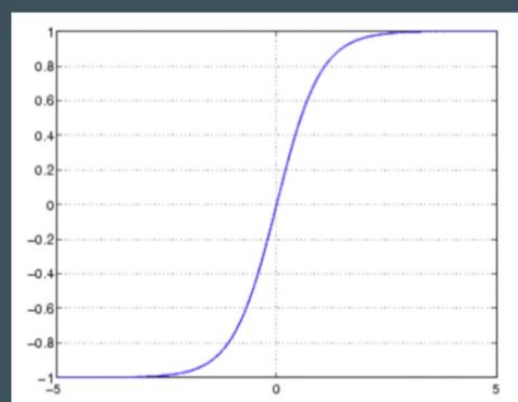


Hyperbolic Tangent

$y = \tanh(x) = (e^x - e^{-x}) / (e^x + e^{-x})$

$dy/dx = 1 - y^2$

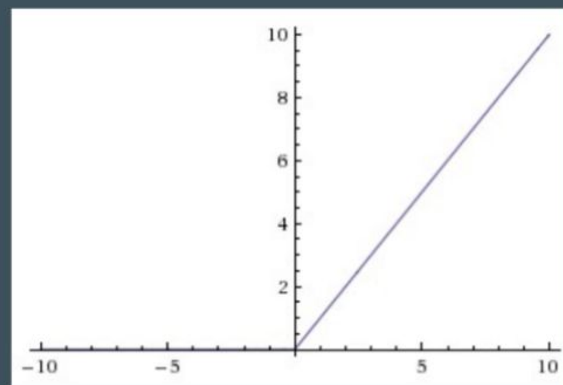
Exercise: find relationship between tanh(x) and sigmoid(x)



Rectifier Linear Unit (relu)

$y = \text{relu}(x) = \max(0, x)$

$dy/dx = 1$ if $x > 0$, 0 if $x < 0$



Hyperparameters

- Things that have possibly seemed arbitrary
  - How did I choose the learning rate?
  - How did I choose the regularization param?
  - How did I choose the number of hidden units / hidden layers?
  - How did I choose between sigmoid/tanh/relu?
- These are hyperparameters

Hyperparameters

- No precise way to choose
- Makes people uncomfortable
  - Isn't science supposed to give us exact answers?
  - Exact: Try everything possible, choose the best → Infeasible
- Choose manually, requires experience

Practice

- Practice!
  - People always ask, "how can I get better?" → By testing hyperparameter settings yourself!
  - Don't wait for me, I'm just a video!
  - Will be different for different problems
  - Choose your problem → What are you interested in? (I can only make recommendations, not choose for you)
- This isn't grade school, don't depend on someone to give you homework
- Do you work somewhere? Do they have data? Use it!
- Ask question on the discussion board. That's why Udemy made it!

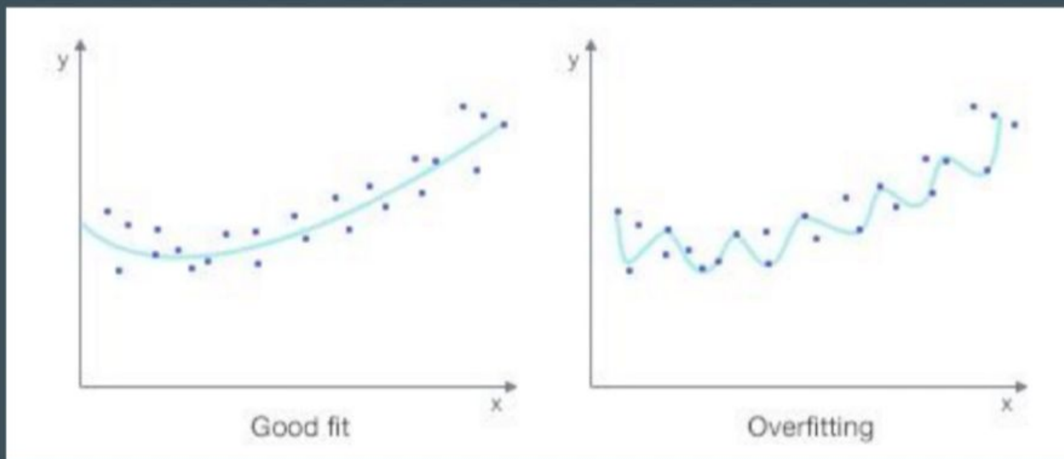
Cross-Validation

- General way to choose hyperparameters.
- E.g. We want to know if 4 vs 5 hidden units is better.
- Do cross-validation on both, choose the one with the best accuracy.
- "Best" could be defined as "statistically significantly better" if you're into stats

BUT

- We don't want to get "perfect" accuracy on our training data
- Data = signal + noise
- Want to fit to signal, not noise

Overfitting

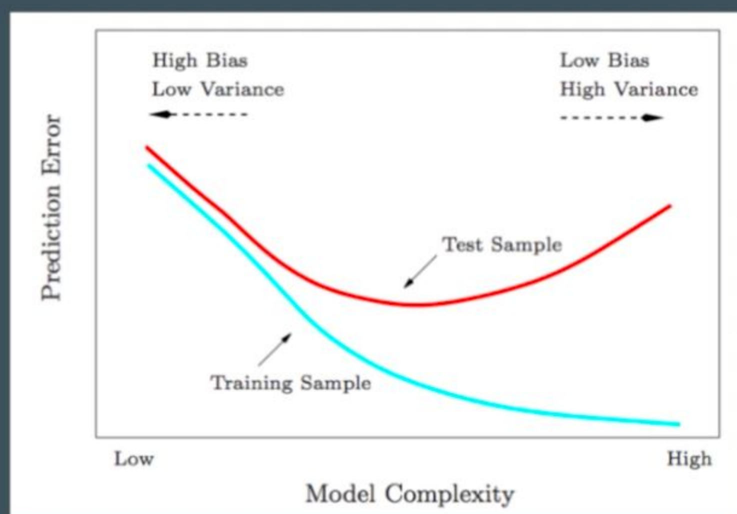


Split up your data

Train - train on this data

Validation - validate on this data -- defined shortly

Test - try not to touch this data until the very end



K-Fold Cross-Validation

Split data into K parts, suppose K=5.

5 iterations.

Iteration 1: Train on parts 2-5, test on part 1
Iteration 2: Train on parts 1,3,4,5, test on part 2
Iteration 3: Train on parts 1,2,4,5, test on part 3
Etc...

Then: take mean and variance of classification rate.

Can do statistical test to compare (i.e. T-test), there are Scipy functions for these

K-Fold Cross-Validation

Split data into K parts, suppose K=5.

5 iterations.

Iteration 1: Train on parts 2-5, test on part 1
Iteration 2: Train on parts 1,3,4,5, test on part 2
Iteration 3: Train on parts 1,2,4,5, test on part 3
Etc...

Then: take mean and variance of classification rate.

Can do statistical test to compare (i.e. T-test), there are Scipy functions for these

In Code

```
def crossValidation(model, X, Y, K=5):
    X, Y = shuffle(X, Y)
    sz = len(Y) / K
    scores = []
    for k in xrange(K):
        xtr = np.concatenate([ X[:k*sz, :], X[(k*sz + sz):, :] ])
        ytr = np.concatenate([ Y[:k*sz], Y[(k*sz + sz):] ])
        xte = X[k*sz:(k*sz + sz), :]
        yte = Y[k*sz:(k*sz + sz)]

        model.fit(xtr, ytr)
        score = model.score(xte, yte)
        scores.append(score)
    return np.mean(scores), np.std(scores)
```

Hyperparameters

- Manually choosing the learning rate and regularization penalty
- It’s a question I get often
- People are “uncomfortable” that there is a number you can’t calculate, that you just need to find
- Isn’t this science? Doesn’t science give direct and concrete answers?
- Get used to it!

Learning rate

- We know so far: it's a "small number"
- You've seen some examples, so you have an idea of order of magnitude
- The right scale:
- If I've already tried 0.1, then I don't need to try 0.09, 0.08, 0.07, etc.
- (But you would have discovered that through experience!)
- Better to go down on log scale / factors of 10
- 10e-1, 10e-2, 10e-3, etc...
- I've used 10e-7 before

Too high or too low

- Too high → cost goes to infinity / NaN
- Neural network will continue to train as normal, multiplying NaNs as if they were actual numbers
- Cost converges too slowly → learning rate is too low
- Try increasing by factor of 10
- Problems?
- Try to turn off all modifications except for regular gradient descent (no regularization or anything else you've learned)
- Learning rate that's too high will mess things up, but learning rate that's too low won't
- If things still don't work, could be a problem with your code

Normalizing cost and regularization penalty

- In deep learning part 2 we discuss training a neural network in batches
- Learning rate will be sensitive to number of training points / batch size
- Because cost is sum of individual errors
- Just divide by N (batch size) to normalize it

- Same issue with number of parameters and regularization
- To make regularization penalty independent of number of parameters, divide by number of parameters

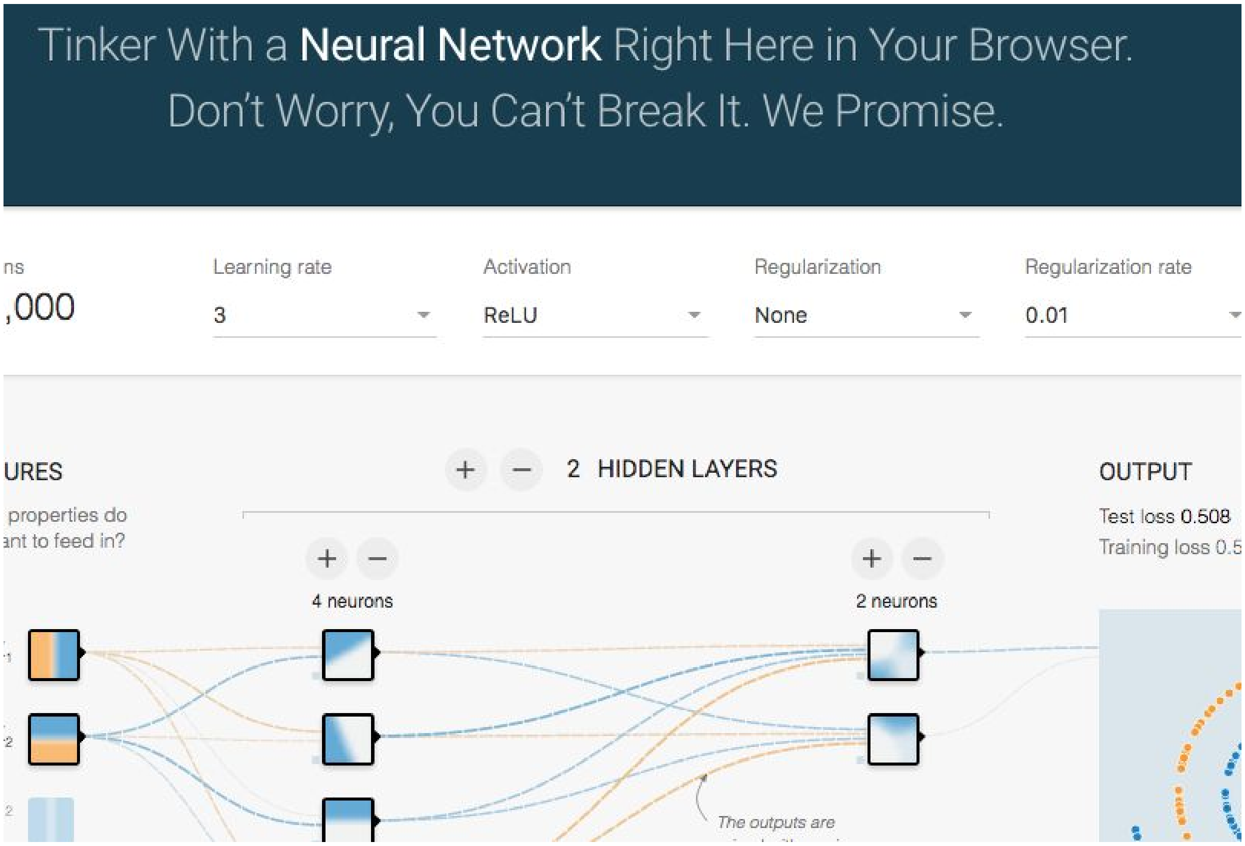
Normalizing cost and regularization penalty

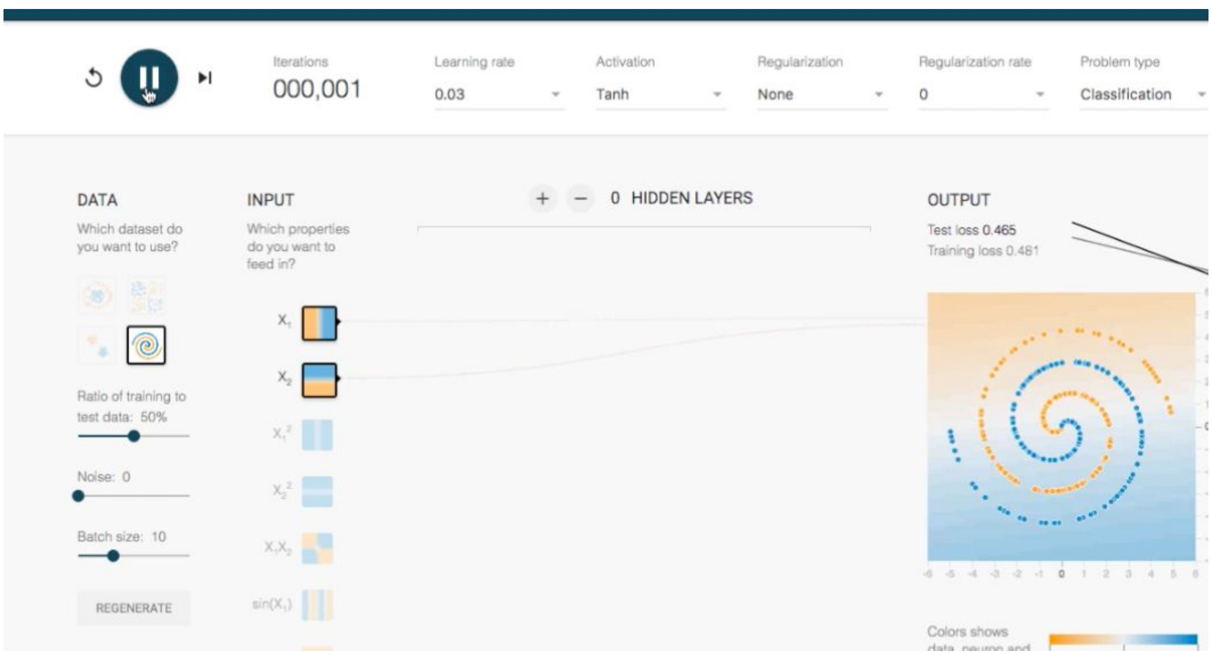
- In deep learning part 2 we discuss training a neural network in batches
- Learning rate will be sensitive to number of training points / batch size
- Because cost is sum of individual errors
- Just divide by N (batch size) to normalize it

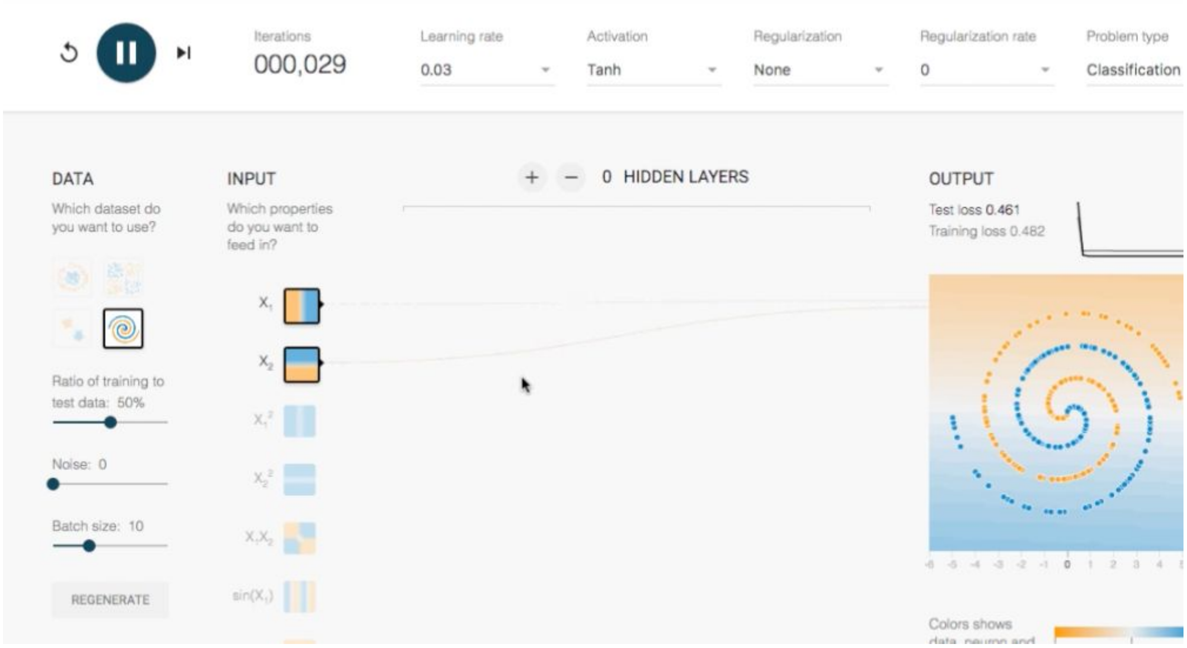
- Same issue with number of parameters and regularization
- To make regularization penalty independent of number of parameters, divide by number of parameters

Regularization too high

- In the past, I forgot to multiply my regularization penalty by the regularization parameter (effectively setting it to 1)
- For that particular problem, this was much too high
- So the error rate just hovered around random guessing
- Sometimes it's useful to just turn off regularization completely, and make learni rate very small
- Tells you if your model actually works or if there's a bug in your code


Tinker With a Neural Network Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.
Learning rate
3
Activation
ReLU
Regularization
None
Regularization rate
0.01
2 HIDDEN LAYERS
4 neurons
2 neurons
OUTPUT
Test loss 0.508
The outputs are


Iterations
000,001
Learning rate
0.03
Activation
Tanh
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
INPUT
Which properties do you want to feed in?
0 HIDDEN LAYERS
OUTPUT
Test loss 0.465
Training loss 0.481
Colors shows



What you've learned

- full neural network, incorporating concepts we learned before, about logistic regression
- how to classify K classes using softmax
- how to derive the gradient for backpropagation
- practical issues: learning rate, regularization penalty, # hidden units, # hidden layers

What else?

A lot!

- convolutional neural networks
- Word2vec / Deep NLP (king - man + woman = queen)
- LSTM
- more training methods:
  - Restricted Boltzmann Machines, Autoencoders
  - Dropout, DropConnect
- Reinforcement learning

You know more than you think you know

- I made this extra lecture a few months after starting the course
- To show you how applicable this stuff is
- Common end goal: understand latest and greatest
  - Convolutional Neural Networks
  - Recurrent Neural Networks (LSTMs)
- How does this class help?

Logistic Regression Review

- 2 functions for supervised machine learning: prediction and training
- Prediction: $y = \sigma(Wx)$
- Training: $W \leftarrow W - \eta\, \partial J/\partial W$

Logistic Regression Review

- 2 functions for supervised machine learning: prediction and trainin
- Prediction: $y = \sigma(Wx)$
- Training: $W \leftarrow W - \eta\ \partial J/\partial W$

Neural Networks (This Course)

- Prediction: $y = \sigma(W_1\sigma(W_2x))$
- Training: $W_i \leftarrow W_i - \eta\ \partial J/\partial W_i$
- Can be arbitrarily deep, as you'll see

Convolutional Neural Networks

- Prediction: $y = \sigma(W_1\sigma(W_2 * x))$
- Training: $W_i \leftarrow W_i - \eta\ \partial J/\partial W_i$
- * = convolution

Recurrent Neural Networks

- Prediction:

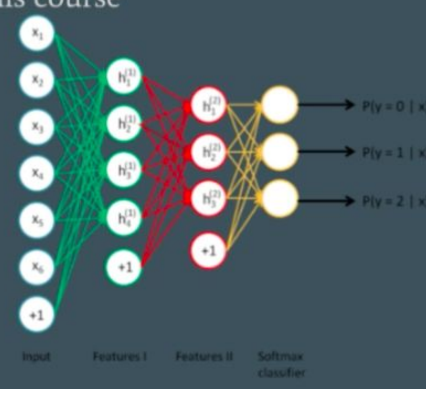
$$h(t) = \sigma(W_x x(t) + W_h h(t - 1))$$

$$y(t) = \sigma(W_o h(t))$$

- Training: $W_i \leftarrow W_i - \eta\, \partial J/\partial W_i$

Autoencoders

- Instead of train(inputs=X, outputs=Y), just call train(inputs=X, outputs=X)
  - Auto = self
- Stacking autoencoders + logistic regression works very well
- You can do this with **JUST** the knowledge from this course



Moral of the story

- Knowing how to take the derivative of the cost function is critical
- (The rest is just creatively building the architecture / layout of the NN)
  - You can even make up your own
  - This is what many researchers do, and they get published in scientific journals!
- Each of these just required one more simple step
- This class will make the rest go from out-of-your-reach → easy

Exercises and How to get Good

- "Can you put up more exercises?"
- Machine learning models are for making predictions
  - What do you want to predict?
  - What are you interested in?
  - Hopefully you can answer that!
- Standard ML datasets / benchmarks:
  - MNIST
  - CIFAR
  - SVHN
  - Iris
  - Government data
  - Enron emails

Theoretical ML is very, very hard

- No magical homework questions
- "Calculate the force on a man standing in an elevator"
- Theoretical questions in ML are extremely difficult
- Focus on understanding the material
- Do by yourself, don't skip any steps:
    - Derivative of sigmoid, hyperbolic tangent, softmax
    - Derivative of cost function wrt weights using softmax (requires much attention to detail)
        - Deliberately left it as an exercise in this course
    - Try dJ/dW on a very deep network, understand the recursive pattern
    - Vectorize the weight updates by hand so you can code them in Numpy

Predictive models are for data

- You choose what data to use it on, only you know what you are interested in
- Must practice
- Not just plug-and-chug
    - Need to pick right # hidden units, # hidden layers, learning rate, regularization, etc.
- Lots of time and waiting
- You will not get good just watching videos
    - That is like trying to master tennis from a book
- If you come across an issue, ask on the discussion board (that's what it's for!)
- Observe cost as gradient descent progresses
- Cost blowing up → something is wrong

Data pre-processing

- What to do with outliers, what to do with missing data?
  - Many techniques, do your own research - not deep learning specific
- What about images, sounds, and sentences?
  - Images → flatten it to a vector
  - Sound → extract features / FFT
- Common methods of pre-processing:
  - Make the range 0..1
  - Standardize = subtract mean, divide by standard deviation
- Words
  - Bag of words / one-hot encoding (discussed in Linear Regression course and N
  - TF-IDF
- Categorical variables: one-hot encoding

Data pre-processing

- What to do with outliers, what to do with missing data?
  - Many techniques, do your own research - not deep learning specific
- What about images, sounds, and sentences?
  - Images → flatten it to a vector
  - Sound → extract features / FFT
- Common methods of pre-processing:
  - Make the range 0..1
  - Standardize = subtract mean, divide by standard deviation
- Words
  - Bag of words / one-hot encoding (discussed in Linear Regression course and NLP course)
  - TF-IDF
- Categorical variables: one-hot encoding

Summary

- In each course, complexity will go up, hand-holding goes down
- At the beginning:
  - More verbose code, math not rigorous
- Later:
  - Skip more steps
  - Tighten up code
- How to get good at anything:
  - Deep learning is a tool
  - Those who are good with their tools are the ones that use them all the time

Help with Softmax Derivative

- Hardest part of backpropagation derivation is softmax portion (derivative of softmax, + putting it back into the full derivative)
- Only watch this if you've spent a few days on your own first
- No new skills if you already know differential calculus
- Main obstacle: you need to be very careful

Softmax

Definition of softmax:

$$y(k) = \frac{e^{a(k)}}{\sum_{j=1}^{K} e^{a(j)}}$$

KEY: answer depends on whether you take derivative wrt a(k) or a(k'), for k' != k

Softmax

Definition of softmax:

$$y(k) = \frac{e^{a(k)}}{\sum_{j=1}^{K} e^{a(j)}}$$

KEY: answer depends on whether you take derivative wrt a(k) or a(k'), for k' != k

Let's do a(k) first

Softmax derivative

Product rule

$$\frac{dy(k)}{da(k)} = \frac{d(e^{a(k)})}{da(k)} \frac{1}{\sum_{j=1}^{K} e^{a(j)}} + \frac{d\left[\sum_{j=1}^{K} e^{a(j)}\right]^{-1}}{da(k)} e^{a(k)}$$

Softmax derivative

Solve individual derivatives

$$\frac{dy(k)}{da(k)} = \frac{e^{a(k)}}{\sum_{j=1}^{K} e^{a(j)}} - \left[\sum_{j=1}^{K} e^{a(j)}\right]^{-2} e^{a(k)} e^{a(k)}$$

$$\frac{dy(k)}{da(k)} = y(k) - y(k)^2 = y(k)(1 - y(k))$$

Softmax derivative

Product rule

$$\frac{dy(k)}{da(k)} = \frac{d(e^{a(k)})}{da(k)} \frac{1}{\sum_{j=1}^{K} e^{a(j)}} + \frac{d\left[\sum_{j=1}^{K} e^{a(j)}\right]^{-1}}{da(k)} e^{a(k)}$$

Softmax derivative

Solve individual derivatives

$$\frac{dy(k)}{da(k)} = \frac{e^{a(k)}}{\sum_{j=1}^{K} e^{a(j)}} - \left[\sum_{j=1}^{K} e^{a(j)}\right]^{-2} e^{a(k)} e^{a(k)}$$

$$\frac{dy(k)}{da(k)} = y(k) - y(k)^2 = y(k)(1 - y(k))$$

Softmax Derivative

Wrt a(k'), for k' != k

$$\frac{dy(k)}{da(k')} = e^{a(k)} \frac{d\left[\sum_{j=1}^{K} e^{a(j)}\right]^{-1}}{da(k')}$$

$$\frac{dy(k)}{da(k')} = e^{a(k)}(-1)\left[\sum_{j=1}^{K} e^{a(j)}\right]^{-2} e^{a(k')}$$

Softmax Derivative

Separate the 2 terms and simplify

$$\frac{dy(k)}{da(k')} = -\frac{e^{a(k)}}{\sum_{j=1}^{K} e^{a(j)}} \frac{e^{a(k')}}{\sum_{j=1}^{K} e^{a(j)}} = -y(k)y(k')$$

Softmax Derivative

Combine the 2 answers using delta function. 2 possibilities, only 1 is useful:

$$\frac{dy(k)}{da(k')} = y(k)(\delta(k,k') - y(k'))$$

$$\frac{dy(k)}{da(k')} = y(k')(\delta(k,k') - y(k))$$

Softmax Derivative

$$\sum_{k'=1}^{K} t_n(k') \frac{1}{y_n(k')} y_n(k')(\delta(k,k') - y_n(k))$$

$$\sum_{k'=1}^{K} t_n(k')(\delta(k,k') - y_n(k))$$

We want k' on the outside so y(n, k') cancels. But how do we get rid of the k' and delta completely to get to the final answer?

$$t_n(k) - y_n(k)$$

Softmax Derivative

Split the summation

$$\sum_{k'=1}^{K} t_n(k')\delta(k,k') - \sum_{k'=1}^{K} t_n(k')y_n(k)$$

$$t_n(k) - \sum_{k'=1}^{K} t_n(k')y_n(k)$$

Softmax Derivative

Split the summation

$$\sum_{k'=1}^{K} t_n(k')\delta(k,k') - \sum_{k'=1}^{K} t_n(k')y_n(k)$$

$$t_n(k) - \sum_{k'=1}^{K} t_n(k')y_n(k)$$

Softmax Derivative

y(n, k) doesn't depend on k', so it can be brought outside the sum

$$t_n(k) - y_n(k)\sum_{k'=1}^{K} t_n(k')$$

Only 1 target over all classes can be 1, so sum over all classes is 1

$$t_n(k) - y_n(k)$$