Deep Learning Introduction

April 6, 2021

ddebarr@uw.edu

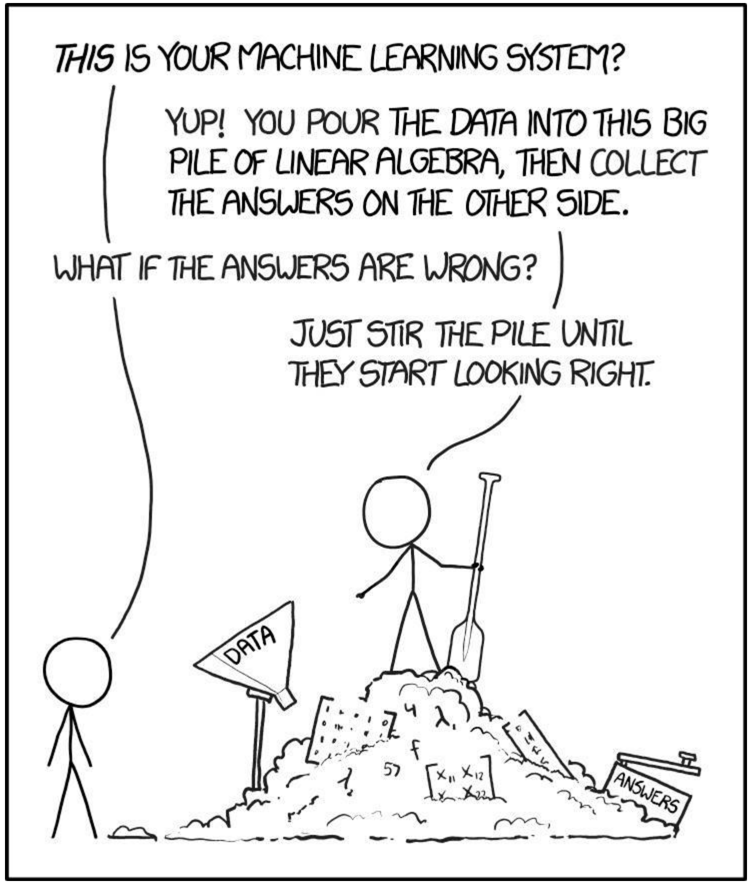
http://cross-entropy.net/ml530/Deep_Learning_0.pdf

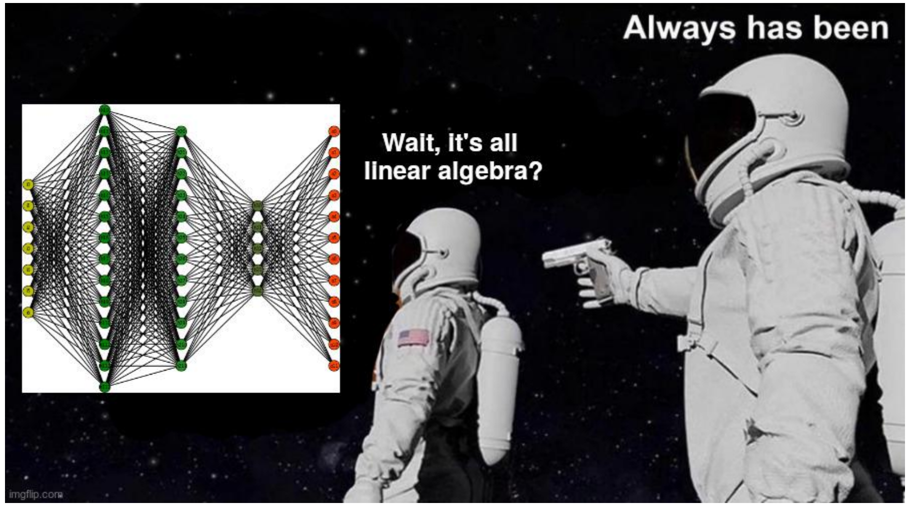
Agenda for Tonight

- About the Class [reviewing the Syllabus]
- About the Textbooks
- Multi-Layer Perceptrons
- Homework

First Things First

[Machine Learning Memes (reddit.com)](): you're welcome; and I'm sorry!





Course Syllabus

https://www.cross-entropy.net/ml530/MLearn530.pdf

Our teaching assistant is ChenWei Lin: chenwl@uw.edu

Textbook #1: Deep Learning Illustrated [DLI]

I. Introducing Deep Learning

1. Biological and Machine Vision
2. Human and Machine Language
3. Machine Art
4. Game-Playing Machines

II. Essential Theory Illustrated

5. The (Code) Cart Ahead of the (Theory) Horse
6. Artificial Neurons Detecting Hot Dogs
7. Artificial Neural Networks
8. Training Deep Networks
9. Improving Deep Networks

III. Interactive Applications

10. Machine Vision
11. Natural Language Processing
12. Generative Adversarial Networks
13. Deep Reinforcement Learning

IV. You and AI

14. Moving Forward with Your Own Deep Learning Projects

Textbook #2: Deep Learning with Python [DLP]

I. Fundamentals

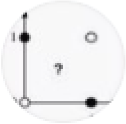
1. What is deep learning?
2. Before we begin: the mathematical building blocks of neural networks
3. Getting started with neural networks
4. Fundamentals of machine learning

II. Practice

5. Deep learning for computer vision
6. Deep learning for text and sequences
7. Advanced deep-learning best practices
8. Generative deep learning
9. Conclusions

Multi-Layer Perceptron (MLP) Agenda

- Gradient Descent and Back Propagation
- Sample Problems:
  - Linear Regression
  - Logistic Regression
  - Multi-Layer Perceptron (MLP)
- Notes
- Overfitting
- Nvidia
  - We're using Kepler generation Graphics Processing Units (GPUs): K80
  - Maxwell, Pascal, Volta, Turing, and Ampere are newer [and more expensive]
  - If purchasing, make sure you have appropriate power to support the GPU



ML Hipster
@ML_Hipster

"Oh sure, going in that direction will totally minimize the objective function" —Sarcastic Gradient Descent.

3:46 PM · Jul 20, 2012 · Twitter for iPhone

Pop Quiz

- What distinguishes a "deep" neural network from a "shallow" neural network?

- One possible answer: a "shallow" neural network has only one hidden layer, while a "deep" neural network has more than one hidden layer

- I wouldn't get too hung up on this distinction

Relating Parameter, Loss, and Gradient Values

Fun with the chain rule: for simplicity, this example uses a single slope parameter (w "hat") to estimate the number of minutes until the next eruption of "Old Faithful" given the number of minutes for the previous eruption https://www.webcitation.org/65V0OnUJ0?url=http://www.geyserstudy.org/geyser.aspx?pGeyserNo=OLDFAITHFUL

$$y = w * x$$

$$\hat{y} = activation(\hat{w} * x) = \hat{w} * x$$ [one neuron with a linear activation function]

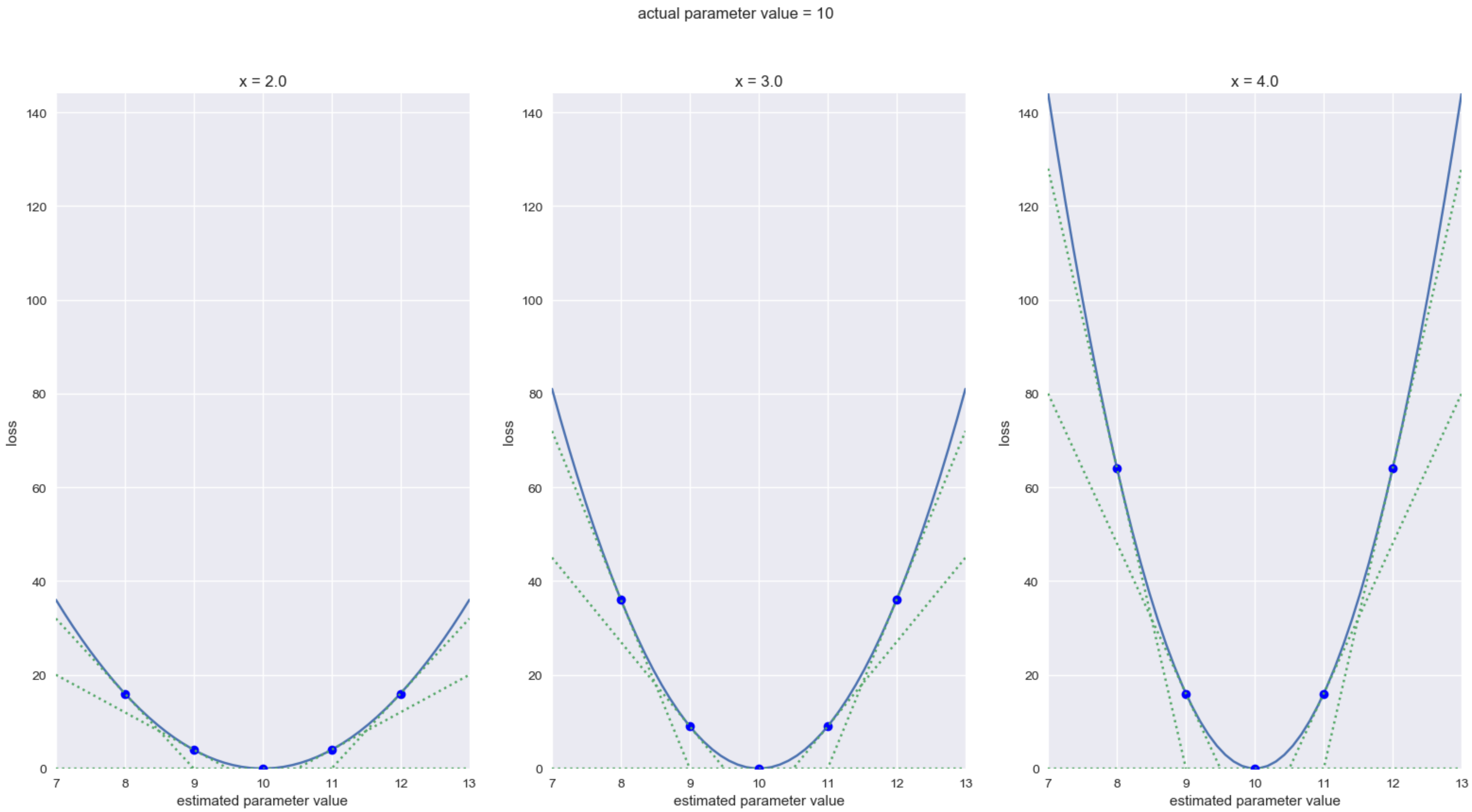
$$loss = (y - \hat{y})^2 = (w * x - \hat{w} * x)^2$$

$$\nabla_{\hat{w}} loss = \frac{\partial\ loss}{\partial\ \hat{w}} = \underbrace{\frac{\partial\ loss}{\partial\ activation(\hat{w} * x)} * \frac{\partial\ activation(\hat{w} * x)}{\partial\ (\hat{w} * x)} * \frac{\partial\ (\hat{w} * x)}{\partial\ \hat{w}}}_{\text{the chain, linking loss to the parameter}} = \big(2 * (\hat{w} * x - w * x)\big) * 1 * x$$

This gradient tells us how to modify the weight to increase loss, so gradient descent uses the opposite direction; i.e. the negative gradient

The gradient gives us both direction (increase or decrease parameter) and magnitude: the steeper the slope, the larger the update

Relating Parameter, Loss, and Gradient Values



Code for the Previous Slide

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
mpl.style.use("seaborn")
figure = plt.figure(figsize = (9, 3))
figure.suptitle("actual parameter value = 10")
w = np.float32(10)
w_hat_sequence = np.linspace(7, 13, 50).astype("float32")
w_hat_list = np.asarray([ 8, 9, 10, 11, 12 ]).astype("float32")
x_list = [ 2, 3, 4 ]
```

```
for i in range(len(x_list)):
    x = np.float32(x_list[i])
    subfigure = plt.subplot(131 + i)
    subfigure.set_xlabel("estimated parameter value")
    subfigure.set_ylabel("loss")
    subfigure.set_title("x = " + str(x))
    loss_sequence = ((w * x - w_hat_sequence * x)**2)
    loss_list = ((w * x - w_hat_list * x)**2)
    gradient_list = (2.0 * (w_hat_list * x - w * x)) * 1 * x
    plt.xlim(7, 13)
    plt.ylim(0, 144)
    plt.plot(w_hat_sequence, loss_sequence, color = "C0")
    for j in range(len(w_hat_list)):
        plt.plot(w_hat_list[j], loss_list[j], "bo")
    for i in range(len(gradient_list)):
        plt.plot(w_hat_sequence, gradient_list[i] * w_hat_sequence - gradient_list[i] * w_hat_list[i] + loss_list[i], linestyle = ":", color = "C1")
figure.show()
```

You may want to reference this when doing homework ☺

Fear the Gradient, for It Represents Change

An alternative view of the gradient we all know and love ❤️

The gradient can be viewed as the limit of the ratio of the change in the loss to the change in the parameter, as the change in the parameter goes to zero.

$$gradient = \lim_{\Delta\, parameter \to 0} \left(\frac{\Delta\, loss}{\Delta\, parameter} \right)$$

In our example, gradient $= -36$ when $w = 10,\ \ x = 3,$ and $\hat{w} = 8$ …

```
>>> import numpy as np
>>> w = 10
>>> x = 3
>>> w_hat = 8
>>> delta_list = np.asarray([ - .1, -.01, -.001, .001, .01, .1 ])
>>> numerators = ((w * x - (w_hat + delta_list) * x)**2)  -  ((w * x - w_hat * x)**2)
>>> denominators = (w_hat + delta_list) - w_hat
>>> print(numerators / denominators)
[-36.9   -36.09  -36.009 -35.991 -35.91  -35.1  ]
```

∂: "wonky" d; for partial derivative (delta/change)

Examples

- Linear Regression
- Logistic Regression
- Multi-Layer Perceptron

Stochastic Gradient Descent (SGD) [Backpropagation of Error]

- Basis for learning in modern deep learning
- Update the weights using $-LearningRate * \frac{\partial loss}{\partial weight}$
- Last layer:

$$\frac{\partial loss}{\partial weight_{-1}} = \frac{\partial loss}{\partial activation_{-1}} \frac{\partial activation_{-1}}{\partial product_{-1}} \frac{\partial product_{-1}}{\partial weight_{-1}}$$

- Second to last layer:

$$\frac{\partial loss}{\partial weight_{-2}} = \frac{\partial loss}{\partial activation_{-1}} \frac{\partial activation_{-1}}{\partial product_{-1}} \frac{\partial product_{-1}}{\partial activation_{-2}} \frac{\partial activation_{-2}}{\partial product_{-2}} \frac{\partial product_{-2}}{\partial weight_{-2}}$$

loss only appears in first factor

lots of products and activations in the middle

appearing as the denominator of a factor means appearing as the numerator of the next

weight only appears in last factor

Alternatives to “Vanilla” Stochastic Gradient Descent

- Root Mean Square (gradient) Propagation (RMSProp)
- Adaptive Moments (AdaM: RMSProp plus momentum)
- These methods use Exponentially Weighted Moving Averaging (EWMA)

Root Mean Square (gradient) Propagation [RMSProp]

- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\ MeanSquare(w,\ t{-}1) + 0.1\left(\partial E / \partial w\,(t)\right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

$\partial E / \partial w$ Note: Professor Hinton uses this to represent $\frac{\partial loss}{\partial weight}$ [‘E’ for Error; ‘w’ for weight]

http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Adaptive Moments [AdaM]

This uses momentum, which helps to avoid local minima and improve convergence speed

The update rule for `variable` with gradient `g` uses an optimization described at the end of section2 of the paper:

$$t := t + 1$$

$$lr_t := extlearning\_rate * \sqrt{1 - beta_2^t}/(1 - beta_1^t)$$

$$m_t := beta_1 * m_{t-1} + (1 - beta_1) * g$$

$$v_t := beta_2 * v_{t-1} + (1 - beta_2) * g * g$$

$$variable := variable - lr_t * m_t/(\sqrt{v_t} + \epsilon)$$

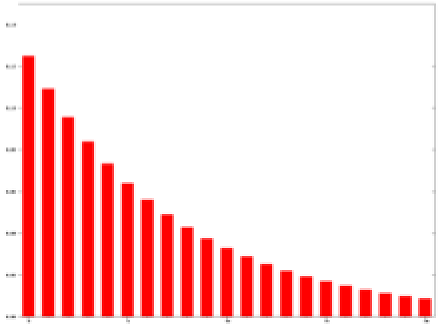
https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

Exponentially Weighted Moving Average (EWMA)

In case you haven't seen this before …

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1-\alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

for $\alpha = 0.1$ …
$0.100\ Y_t\ \ @\ \ time\ t$
$0.090\ Y_t\ \ @\ \ time\ t+1$
$0.081\ Y_t\ \ @\ \ time\ t+2$
…



https://en.m.wikipedia.org/wiki/Moving_average

Notes About Differentiation

- Derivatives are used to update weights (learn models)

- Deep learning can be applied to medicine; e.g. processing radiographs

… that’s right … calculus saves lives! ☺

Rules for Derivatives

Common Functions	Function	Derivative
Constant	c	0
Line	x	1
	ax	a
Square	x^2	$2x$
Square Root	$\sqrt{x}$	$(½)x^{-½}$
Exponential	e^x	e^x
	a^x	$\ln(a)\ a^x$
Logarithms	$\ln(x)$	$1/x$
	$\log_a(x)$	$1\ /\ (x\ \ln(a))$
Trigonometry (x is in radians)	$\sin(x)$	$\cos(x)$
	$\cos(x)$	$-\sin(x)$
	$\tan(x)$	$\sec^2(x)$
Inverse Trigonometry	$\sin^{-1}(x)$	$1/\sqrt{(1-x^2)}$
	$\cos^{-1}(x)$	$-1/\sqrt{(1-x^2)}$
	$\tan^{-1}(x)$	$1/(1+x^2)$

Rules	Function	Derivative
Multiplication by constant	cf	cf'
Power Rule	x^n	nx^{n-1}
Sum Rule	$f + g$	$f' + g'$
Difference Rule	$f - g$	$f' - g'$
Product Rule	fg	$f\,g' + f'\,g$
Quotient Rule	f/g	$(f'\,g - g'\,f)/g^2$
Reciprocal Rule	$1/f$	$-f'/f^2$
Chain Rule (as "Composition of Functions")	$f \circ g$	$(f' \circ g) \times g'$
Chain Rule (using ′)	$f(g(x))$	$f'(g(x))g'(x)$
Chain Rule (using $\frac{d}{dx}$)	$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$	

https://www.mathsisfun.com/calculus/derivatives-rules.html

Partial Derivative of Mean Squared Error

$$\frac{\partial\left((y - activation)^2\right)}{\partial activation} = 2 * (y - activation) * \frac{\partial(y - activation)}{\partial activation}$$

$$= 2 * (y - activation) * \left(\frac{\partial y}{\partial activation} - \frac{\partial activation}{\partial activation}\right)$$

$$= 2 * (y - activation) * (0 - 1)$$

$$= 2 * (y - activation) * (-1)$$

$$= 2 * (activation - y)$$

“Partial” Derivative of Linear Function

$$\frac{dx}{dx} = 1$$

a change in x divided by a change in x … the rate of change is one ☺

A function is said to be linear iff …

$$f(x_1 + x_2) = f(x_1) + f(x_2)$$

$$f(c\,x_1) = cf(x_1)$$

$f(x) = x$ definitely qualifies as a linear activation function

Partial Derivative of Binary Cross Entropy Part 1 of 2

$$\frac{\partial - \ln\left(activation^{y} * (1 - activation)^{(1-y)}\right)}{\partial activation}$$

$$= -\frac{\partial \ln\left(activation^{y} * (1 - activation)^{(1-y)}\right)}{\partial activation}$$

$$= -\frac{\frac{\partial\left(activation^{y} * (1 - activation)^{(1-y)}\right)}{\partial activation}}{activation^{y} * (1 - activation)^{(1-y)}}$$

$$= -\frac{activation^{y} \frac{\partial\left((1 - activation)^{(1-y)}\right)}{\partial activation} + (1 - activation)^{(1-y)} \frac{\partial(activation^{y})}{\partial activation}}{activation^{y} * (1 - activation)^{(1-y)}}$$

$$= -\frac{activation^{y} * (1-y) * (1 - activation)^{(-y)} \frac{\partial(1 - activation)}{\partial activation} + (1 - activation)^{(1-y)} * y * activation^{(y-1)}}{activation^{y} * (1 - activation)^{(1-y)}}$$

$$= -\frac{activation^{y} * (1-y) * (1 - activation)^{(-y)} \left(\frac{\partial 1}{\partial activation} - \frac{\partial activation}{\partial activation}\right) + (1 - activation)^{(1-y)} * y * activation^{(y-1)}}{activation^{y} * (1 - activation)^{(1-y)}}$$

$$= -\frac{activation^{y} * (1-y) * (1 - activation)^{(-y)}(0-1) + (1 - activation)^{(1-y)} * y * activation^{(y-1)}}{activation^{y} * (1 - activation)^{(1-y)}}$$

$$= -\frac{activation^{y} * (1-y) * (1 - activation)^{(-y)}(-1) + (1 - activation)^{(1-y)} * y * activation^{(y-1)}}{activation^{y} * (1 - activation)^{(1-y)}}$$

Partial Derivative of Binary Cross Entropy Part 2 of 2

$$= -\frac{activation^{y} * (1-y) * (1-activation)^{(-y)}(-1) + (1-activation)^{(1-y)} * y * activation^{(y-1)}}{activation^{y} * (1-activation)^{(1-y)}}$$

$$= -\frac{(1-activation)^{(1-y)} * y * activation^{(y-1)} - activation^{y} * (1-y) * (1-activation)^{(-y)}}{activation^{y} * (1-activation)^{(1-y)}}$$

$$= -\frac{y * activation^{(y-1)} - activation^{y} * (1-y) * (1-activation)^{(-1)}}{activation^{y}}$$

$$= -\left(y * activation^{(-1)} - (1-y) * (1-activation)^{(-1)}\right)$$

$$= -\left(\frac{y}{activation} - \frac{(1-y)}{(1-activation)}\right)$$

$$= -\left(\frac{y * (1-activation) - (1-y) * activation}{activation * (1-activation)}\right)$$

$$= -\left(\frac{y - y * activation - activation + y * activation}{activation * (1-activation)}\right)$$

$$= \frac{activation - y}{activation * (1-activation)}$$

"Partial" Derivative of Sigmoid Function

$$\frac{d\left(\frac{1}{1+exp(-x)}\right)}{dx} = -\frac{\frac{d\left(1+exp(-x)\right)}{dx}}{\left(1+exp(-x)\right)^2} = -\frac{\frac{d(1)}{dx}+\frac{d\left(exp(-x)\right)}{dx}}{\left(1+exp(-x)\right)^2}$$

$$= -\frac{0+\frac{d\left(exp(-x)\right)}{dx}}{\left(1+exp(-x)\right)^2} = -\frac{exp(-x)\frac{d(-x)}{dx}}{\left(1+exp(-x)\right)^2} = -\frac{exp(-x)(-1)}{\left(1+exp(-x)\right)^2}$$

$$= \frac{exp(-x)}{\left(1+exp(-x)\right)^2} = \frac{1}{1+exp(-x)}\left(\frac{exp(-x)}{1+exp(-x)}\right)$$

$$= \frac{1}{1+exp(-x)}\left(1-\frac{1}{1+exp(-x)}\right)$$

Partial Derivative of Softmax Function [when i == j (used for "correct" class)]

$$\frac{\partial \frac{e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i}}}{\partial x_j} = \frac{\frac{\partial e^{x_i}}{\partial x_j} \sum_{i=0}^{k-1} e^{x_i} - \frac{\partial \sum_{i=0}^{k-1} e^{x_i}}{\partial x_j} e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i} \sum_{i=0}^{k-1} e^{x_i}} = \frac{e^{x_i} \sum_{i=0}^{k-1} e^{x_i} - e^{x_j} e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i} \sum_{i=0}^{k-1} e^{x_i}}$$

$$= \frac{e^{x_i}\left(\sum_{i=0}^{k-1} e^{x_i} - e^{x_j}\right)}{\sum_{i=0}^{k-1} e^{x_i} \sum_{i=0}^{k-1} e^{x_i}} = \frac{e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i}} \frac{\sum_{i=0}^{k-1} e^{x_i} - e^{x_j}}{\sum_{i=0}^{k-1} e^{x_i}}$$

$$= \frac{e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i}} \left(1 - \frac{e^{x_j}}{\sum_{i=0}^{k-1} e^{x_i}}\right)$$

$$= \frac{e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i}} \left(1 - \frac{e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i}}\right)$$

Partial Derivative of Softmax Function [when i != j (used for “incorrect” class)]

$$\frac{\partial \frac{e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i}}}{\partial x_j} = \frac{\frac{\partial e^{x_i}}{\partial x_j}\sum_{i=0}^{k-1} e^{x_i} - \frac{\partial \sum_{i=0}^{k-1} e^{x_i}}{\partial x_j} e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i} \sum_{i=0}^{k-1} e^{x_i}} = \frac{0\sum_{i=0}^{k-1} e^{x_i} - e^{x_j}e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i} \sum_{i=0}^{k-1} e^{x_i}}$$

$$= \frac{0 - e^{x_i}e^{x_j}}{\sum_{i=0}^{k-1} e^{x_i} \sum_{i=0}^{k-1} e^{x_i}} = -\frac{e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i}}\left(\frac{e^{x_j}}{\sum_{i=0}^{k-1} e^{x_i}}\right)$$

$$= \frac{e^{x_i}}{\sum_{i=0}^{k-1} e^{x_i}}\left(0 - \frac{e^{x_j}}{\sum_{i=0}^{k-1} e^{x_i}}\right)$$

Sigmoid versus Softmax for Binary Classification

I would go with using the sigmoid activation function for the last layer of a binary classification model; but it really doesn't matter much …

$$\frac{\exp(x)}{\exp(x) + \exp(-x)} = \frac{1}{1 + \exp(-2x)}$$

$$\frac{\exp\left(\frac{1}{2}x\right)}{\exp\left(\frac{1}{2}x\right) + \exp\left(-\frac{1}{2}x\right)} = \frac{1}{1 + \exp(-x)}$$

The sigmoid function is also known as the logistic function, the inverse of the logit function: the sigmoid maps a log odds (logit) value to a probability

Shifted Log Odds for Softmax

- By definition $\text{logit} = \log(odds) = \log\frac{p}{1-p}$
- For softmax …

$$\frac{p_i}{1-p_i} = \frac{\frac{\exp(z_i)}{\exp(z_i) + \sum_{j\neq i}\exp(z_j)}}{\frac{\sum_{j\neq i}\exp(z_j)}{\exp(z_i) + \sum_{j\neq i}\exp(z_j)}} = \frac{\exp(z_i)}{\sum_{j\neq i}\exp(z_j)}$$

$$(1-p_i) * \exp(z_i) = p_i * \sum_{j\neq i}\exp(z_j)$$

$$\exp(z_i) = \frac{p_i}{1-p_i} * \sum_{j\neq i}\exp(z_j)$$

$$z_i = \log\frac{p_i}{1-p_i} + \log\sum_{j\neq i}\exp(z_j)$$

Partial Derivatives of Product Function

Partial Derivative of Product with Respect to Weight:

$$\frac{\partial(x * w)}{\partial w} = x$$

Partial Derivative of Product with Respect to Input:

$$\frac{\partial(x * w)}{\partial x} = w$$

Reminder: $-\text{rate} * \frac{\partial loss}{\partial weight}$ is used to update a weight

[“d weight” (funky d = partial derivative) only appears at the end of the chain]

Functional Note

- Sometimes folks go ahead and multiply the Partial Derivative of Binary Cross Entropy with the Partial Derivative of the Sigmoid Function
  - (activation – y)
  - … instead of …
  - ((activation – y) / (activation * (1 – activation))) * (activation * (1 – activation))
- We'll avoid this, so we view back propagation across Product and Activation Functions consistently

Numerical Note

Floating point numbers do not necessarily have an “exact” representation

- Example …
  - import numpy as np
  - np.float32(0.1) * np.float32(0.1)
  - 0.010000001   # where did the stray one-billionth digit come from?
- Institute of Electrical and Electronic Engineers Standard for Floating-Point Arithmetic (IEEE 754) specifies the following for representing 32-bit floating-point numbers
  - ((-1)**(Sign)) * (1.0 + Fraction) * ((2)**(Exponent - Bias))
    - Sign: 1 bit
    - Exponent: 8 bits
    - Fraction: 23 bits
    - Bias = 127
  - 32-bit version of 0.1
    - 0 01111011 10011001100110011001101
    - Sign = 0
    - Exponent = 123
    - Fraction = np.sum(1.0 / (2 ** np.array([1, 4, 5, 8, 9, 12, 13, 16, 17, 20, 21, 23])))
    - print(binascii.b2a_hex(struct.pack(">f", np.float32(0.1))))   # float to (big-endian) binary to ascii
    - b'3dccccd'

Computational Note

In practice, you'll probably never implement training nor inference code for a neural network [optimization of operations that dominate computation time (such as dot products) can be tricky]

```
D:\ML410> .\UnrolledLoop.exe
last dot product: 45.12804     0.6841716 seconds elapsed (rolled)
last dot product: 45.12801     0.2353421 seconds elapsed (unrolled: -65.6% reduction)
last dot product: 45.12801     0.197472 seconds elapsed (unrolled+unchecked: -16.1% reduction)

D:\ML410> .\UnrolledLoop.exe
last dot product: -4.605535    0.697136 seconds elapsed (rolled)
last dot product: -4.605533    0.2353696 seconds elapsed (unrolled: -66.2% reduction)
last dot product: -4.605533    0.1964736 seconds elapsed (unrolled+unchecked: -16.5% reduction)

D:\ML410> .\UnrolledLoop.exe
last dot product: 34.96746     0.6741987 seconds elapsed (rolled)
last dot product: 34.96748     0.237367 seconds elapsed (unrolled: -64.8% reduction)
last dot product: 34.96748     0.1954762 seconds elapsed (unrolled+unchecked: -17.6% reduction)
```

http://cross-entropy.net/ml530/UnrolledLoop.cs.txt

Overfitting

- Overfitting is the term used to describe updates to a model that improve performance on the training set while hurting performance on a validation set

- Possible options to avoid overfitting include
  - Early stopping: you should (must?) definitely stop training when performance on a hold-out validation set starts to get worse
  - Regularization
  - Dropout

Regularization [Weight Decay]

- The goal of "regularization" is to produce smaller weights
  - Larger weights can yield higher variance; e.g. a small change in input values can produce a large change in output values
- A fraction of the Lp (p for power) norm for the parameters of a layer is added to the loss value

$$\|x\|_p = \left(|x_1|^p + |x_2|^p + \cdots + |x_n|^p\right)^{1/p}$$

  - L2: penalizing the sum of squared weights [tf.nn.l2_loss(): sum(w ** 2) / 2]
  - L1: penalizing the sum of absolute weights
- Typically applied to weights, not bias parameters
  - Hyperparameter: the regularization factor is a multiplier for the penalty
    - Typical multiplier is in the interval (0, 1)
    - Often small; e.g. 0.001
  - $-LearningRate * \left(\frac{\partial loss}{\partial weight} + \frac{\partial penalty}{\partial weight}\right)$ is used to update the weight

The "L" comes from the use of norms from Lebesgue spaces. It's often written with lowercase 'l' (which may look like a 1)

Dropout

- Dropout prevents memorization of inputs: can be viewed as randomly selecting a feature subset [consider random forests]
- "For each element of x, with probability rate, outputs 0, and otherwise scales up the input by 1 / (1 - rate)" [only performed during training]
- "The scaling is such that the expected sum is unchanged"
- Suppose we use a dropout rate of 20% applied to a layer with 100 outputs
  - We expect 20 of the 100 outputs to be set to 0
  - We expect the other 80 of the 100 outputs to be multiplied by the inverse propensity weight
    - If the probability of dropout is 0.2, the propensity for selection is 1 – 0.2 = 0.8
    - Inverse propensity weight = 1 / 0.8 = 1.25 [we're increasing the activation outputs to make up for the twenty that got dropped: 80 * 1.25 = 100]

https://www.tensorflow.org/api_docs/python/tf/nn/dropout

Initialization

- For initializing a layer, Xavier Glorot suggested setting

$$Variance(initial\_weights) = \frac{1}{\left(\frac{FeatureCount + NeuronCount}{2}\right)}$$

  - The Gaussian distribution is parameterized by mean and variance directly; mean = 0 and variance as above
  - The uniform distribution is parameterized by the boundaries of the values: for variance as above, we need

$$initial\_weight \in \left[-\sqrt{\frac{3}{\left(\frac{FeatureCount+NeuronCount}{2}\right)}}, \sqrt{\frac{3}{\left(\frac{FeatureCount+NeuronCount}{2}\right)}}\right]$$

Homework: How Wide? How Deep?

- Keras Tuner
  - From the same folks who brought you Keras
  - https://github.com/keras-team/keras-tuner
  - pip install keras-tuner
- We're going to …
  - Create a parameterized build_model() function
  - Declare a tuner
    - Hyperband (Hyperparameter Bandit)
    - BayesianOptimization (Gaussian Process)
    - RandomSearch
  - Execute the tuner's search() method to select the best model

https://cross-entropy.net/ml530/mnist-search.py.txt https://homes.cs.washington.edu/~jamieson/hyperband.html

Hyperparameter Bandits (HyperBand)

Iteratively train configurations: train a few epochs, keep the best few …

```python
# you need to write the following hooks for your custom problem
from problem import get_random_hyperparameter_configuration,run_then_return_val_loss

max_iter = 81  # maximum iterations/epochs per configuration
eta = 3 # defines downsampling rate (default=3)
logeta = lambda x: log(x)/log(eta)
s_max = int(logeta(max_iter))  # number of unique executions of Successive Halving (minus one)
B = (s_max+1)*max_iter  # total number of iterations (without reuse) per execution of Succesive Halving (n,r)

#### Begin Finite Horizon Hyperband outlerloop. Repeat indefinetely.
for s in reversed(range(s_max+1)):
    n = int(ceil(int(B/max_iter/(s+1))*eta**s)) # initial number of configurations
    r = max_iter*eta**(-s) # initial number of iterations to run configurations for

    #### Begin Finite Horizon Successive Halving with (n,r)
    T = [ get_random_hyperparameter_configuration() for i in range(n) ]
    for i in range(s+1):
        # Run each of the n_i configs for r_i iterations and keep best n_i/eta
        n_i = n*eta**(-i)
        r_i = r*eta**(i)
        val_losses = [ run_then_return_val_loss(num_iters=r_i,hyperparameters=t) for t in T ]
        T = [ T[i] for i in argsort(val_losses)[0:int( n_i/eta )] ]
    #### End Finite Horizon Successive Halving with (n,r)
```

HyperBand Example: Brackets and Rounds

```
>>> import math
>>> max_epochs = 32
>>> factor = 3
>>> for bracket in [ 3, 2, 1, 0 ]:
...     for round in range(bracket + 1):
...         candidates = math.ceil((math.ceil(1 + math.log(max_epochs, factor)) / (bracket + 1)) * (factor ** (bracket - round)))
...         epochs = math.ceil(max_epochs / factor ** (bracket - round))
...         print(bracket, round, candidates, epochs, sep = "\t")

bracket    round    candidates    epochs
      3        0            34         2
      3        1            12         4
      3        2             4        11
      3        3             2        32
      2        0            15         4
      2        1             5        11
      2        2             2        32
      1        0             8        11
      1        1             3        32
      0        0             5        32    # bracket 0 is random search
```

Bayesian Optimization

- Given results for a set of trials, we construct a GaussianProcessRegressor() to predict the objective value
  - Matern kernel is used for measuring similarity
  - Predicted values can be viewed as a weighted estimate of observed neighbors
- Hyperparameter configurations to evaluate are selected based on an Upper Confidence Bound (UCB)
  - UCB: predicted performance plus a multiple of the standard deviation
  - The multiple (beta) parameter controls the amount of exploration
  - Bounded Limited-memory Broyden Fletcher Goldfarb Shanno (L-BFGS-B) optimization is used to find candidates [recent differences in gradients to approximate Hessian]
  - Technically, we're using a lower confidence bound when our objective needs to be minimized ☺

Accessing Your Virtual Machine (VM)

Check your name@uw.edu email account: https://gmail.uw.edu