Machine Learning: Introduction to Linear Regression, Logistic Regression, and Neural Networks

Chapter 6 Optimization, Validation, Accuracy & Regularization

Optimization, Validation, Accuracy & Regularization

Section	Title	Description
6.1	Stochastic and Mini-Batch Optimization	This section presents the mini-batch optimization algorithm
6.2	Momentum, RmsProp and Adam Optimization	This section presents optimizers that are alternatives to the Gradient Descent
6.3	Code Walkthrough Version 3.1	Code walkthrough of optimization updates
6.4	Validation	This section describes using a validation dataset to measure performance.
6.5	Performance Measures	This section describes metrices in addition to accuracy for measure performance.
6.6	Code Walkthrough Version 3.2	This section presents the updates to the code base for validation and performance measures.
6.7	Regularization	This section describes the purpose and approaches for regularization
6.8	Hyperparameter Search	This section describes the hyperparameter search process for tuning a machine learning system
6.9	Code Walkthrough Version 3.3	This section presents the updates to the code base for regularization and hyperparameter searches.
6.10	Underfitting and Overfitting	This section defines underfitting and overfitting and discusses diagnosis of these conditions and remedies to address these conditions.

Chapter 6.1 Stochastic and Mini-Batch Gradient Descent

Mini-Batch Gradient Descent

Goal of this Section:

- Present algorithms for Stochastic and Mini-Batch Gradient Descent

Batch versus Stochastic Gradient Descent

- Gradient Descent algorithm used in Training presented earlier is often referred to as “Batch” Gradient Descent
  - Called Batch because at each epoch gradient calculation/update to parameters is based on all data samples at once
- Alternative: Stochastic Gradient Descent – at each epoch compute gradient/update using one training data sample at a time, looping over all data samples

Training Algorithm: Stochastic Gradient Descent

Assume Neural Network with N layers

Input training data: feature matrix X and values Y

Make initial guess for parameters: $W^{[k]}$ and $b^{[k]}$ for k=1,…,N

Choose learning rate $\alpha > 0$

To simplify notation, remove subscript epoch=

1. Loop for epoch i = 1, 2, …
   - Randomly permute samples
   - Loop over samples j=0,…,m-1
     - Forward Propagate using feature vector X_j: compute $A_j^{[k]}$ for k=1,…,N
     - Back Propagate using X_j, Y_j: compute $\nabla_{W^{[k]}}L$, $\nabla_{b^{[k]}}L$ for k=1,…,N
     - Update parameters: for k=1,…,N
       - $W^{[k]} \leftarrow W^{[k]} - \alpha\nabla_{W^{[k]}}L$
       - $b^{[k]} \leftarrow b^{[k]} - \alpha\nabla_{W^{[k]}}L$
   - Forward Propagate using X: compute $A^{[k]}$ for k=1,…,N
   - Predict: compute Y_{pred} and compute accuracy comparing with Y
   - Compute loss using $A^{[N]}$

Loop for fixed number of epochs

Stochastic Gradient Descent - Example

- Consider Logistic Regression and data (2 features and 2 data points)

$X = \begin{bmatrix} 1 & 2 \\ -2 & -5 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & 1 \end{bmatrix}$

$X_{sample=0} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \quad Y_{sample=0} = [0] \quad X_{sample=1} = \begin{bmatrix} 2 \\ -5 \end{bmatrix} \quad Y_{sample=1} = [1]$

- Assume that initial parameter values are:

$W = \begin{bmatrix} 0.1 & 0.1 \end{bmatrix} \quad \mathrm{b} = [0.2]$

- Assume training is performed using Gradient Descent with $\alpha = 0.1$
- Epoch 1 – Sample Index = 0
- Forward Propagation:

$Z_{sample=0} = WX_{sample=0} + b = \begin{bmatrix} 0.1 & 0.1 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \end{bmatrix} + [0.2] = [0.1]$

$A_{sample=0} = f\left(Z_{sample=1}\right) = \left[\frac{1}{1+e^{-0.1}}\right] = [0.5250]$

Stochastic Gradient Descent - Example

- Back Propagation:

$$\nabla_A L_{sample=0} = -\frac{1}{1}\left(\frac{Y_{sample=0}}{A_{sample=0}} - \frac{1-Y_{sample=0}}{1-A_{sample=0}}\right) = -\frac{1}{1}\left[-\frac{1}{1-0.5250}\right] = [2.1052]$$

$$\frac{\partial A_{sample=0}}{\partial Z_{sample=0}} = A_{sample=0} - A^2_{sample=0} = [0.2494]$$

$$\nabla_Z L_{sample=0} = \nabla_A L_{sample=0} * \frac{\partial A_{sample=0}}{\partial Z_{sample=0}} = [2.1052] * [0.2494] = [0.5250]$$

$$\nabla_W L_{sample=0} = \nabla_Z L_{sample=0} X^T_{sample=0} = [0.5250][1 \quad -2] = [0.5250 \quad -1.0500]$$

$$\nabla_b L_{sample=0} = \sum_{j=0}^{m-1} \nabla_Z L_{sample=0,j} = [0.5250]$$ (sum of entries of $\nabla_Z L_{sample=0}$)

- Update the parameters:

$$W = W - \alpha \nabla_W L_{sample=0} = [0.1 \quad 0.1]-0.1*[0.5250 \quad -1.0500] = [0.0475 \quad 0.2050]$$

$$b = b - \alpha \nabla_b L_{sample=0} = [0.2]-0.1*[0.5250] = [0.1475]$$

Stochastic Gradient Descent - Example

- Epoch 1 – Sample Index=1
- Forward Propagation:

$$Z_{sample=1} = WX_{sample=1} + b = [0.0475 \quad 0.2050]\begin{bmatrix}2\\-5\end{bmatrix} + [0.1475] = [-0.7825]$$

$$A_{sample=1} = f\left(Z_{sample=1}\right) = \left[\frac{1}{1+e^{0.7525}}\right] = [0.3138]$$

- Back Propagation

$$\nabla_A L_{sample=1} = -\frac{1}{1}\left(\frac{Y_{sample=1}}{A_{sample=1}} - \frac{1-Y_{sample=1}}{1-A_{sample=1}}\right) = -\frac{1}{1}\left[\frac{1}{0.3138}\right] = [-3.1869]$$

$$\frac{\partial A_{sample=1}}{\partial Z_{sample=1}} = A_{sample=1} - A^2_{sample=1} = [0.2153]$$

$$\nabla_Z L_{sample=1} = \nabla_A L_{sample=1} * \frac{\partial A_{sample=1}}{\partial Z_{sample=1}} = [-3.1869] * [0.2153] = [-0.6862]$$

$$\nabla_W L_{sample=1} = \nabla_Z L_{sample=1} X^T_{sample=1} = [-0.6826][2 \quad -5] = [-1.3724 \quad 3.4311]$$

$\nabla_b L_{sample=1} = \sum_{j=0}^{m-1} \nabla_Z L_{sample=1,j} = [-0.6862]$ (sum of entries of $\nabla_b L_{sample=1}$)

- Update the parameters:

$W = W - \alpha \nabla_W L_{sample=1}$= $[0.0475 \quad 0.2050]$-0.1*$[-1.3724 \quad 3.4311]$= $[0.1847 \quad -0.1381]$

$b = b - \alpha \nabla_b L_{sample=1}$= $[0.1475]$-0.1*$[-0.6862]$= $[0.2161]$

Stochastic Gradient Descent - Example

- Example shows 1 epoch of Stochastic Gradient Descent Algorithm
- In that epoch, W and b are updated twice (since there are 2 samples – there would be m updates in case of m samples)
- Process is repeated for epoch = 2, 3, 4, …

Mini-Batch Gradient Descent

- Mini-Batch Gradient is compromise between Stochastic and Batch Gradient Descent
- Split training data into mini-batches (eg: 50 samples per mini-batch)
- At each epoch, loop over mini-batches and compute update to parameters one mini-batch at a time

Mini-Batch Creation Algorithm

Assume m training samples

Choose mini-batch size b

Let n = number of mini-batches

- n = m/b if b goes into m evenly
- n = ceil(m/b) if b doesn't go into m evenly

  (ceil() always rounds up to next integer eg: ceil(4.1)=5)

1. Randomly permute samples
2. Split training samples into n mini-batches
   - Take 1st b samples for mini-batch 0, 2nd b samples for mini-batch 1, … mini-batch n-1
   - Final mini-batch may have less than b samples

Mini-Batch Creation - Example

- Consider a case of 2 features and 5 data points

$$X = \begin{bmatrix} 1 & 2 & 4 & 1 & -1 \\ -2 & -5 & -8 & -2 & 2 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

- Assume a mini-batch size of 3
- 3 doesn’t go into 5 evenly so, there will be 2 mini-batches (3 & 2 data points)
- Split into 2 mini-batches:

$$X_{batch=0} = \begin{bmatrix} 1 & 2 & 4 \\ -2 & -5 & -8 \end{bmatrix} \qquad Y_{batch=0} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

$$X_{batch=1} = \begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix} \qquad Y_{batch=1} = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

Training Algorithm: Mini-Batch Gradient Descent

Assume Neural Network with N layers

Input training data: feature matrix X and values Y

Choose batch size b

Make initial guess for parameters: $W^{[k]}$ and $b^{[k]}$ for k=1,…,N

Choose learning rate $\alpha > 0$

To simplify notation, remove subscript epoch=

1. Loop for epoch i = 1, 2, …
    - Generate mini-batches $(X_{batch=j}, Y_{batch=j})$, j=0,…,n-1
    - Loop over mini-batches j=0,…,n-1
        - Forward Propagate using $X_{batch=j}$: compute $A_{batch=j}^{[k]}$ for k=1,…,N
        - Back Propagate using $X_{batch=j}, Y_{batch=j}$: compute $\nabla_{W^{[k]}} L_{batch=j}, \nabla_{b^{[k]}} L_{batch=j}$ k=1,…,N
        - Update parameters for k=1,…,N
            - $W^{[k]} \leftarrow W^{[k]} - \alpha \nabla_{W^{[k]}} L_{batch=j},$
            - $b^{[k]} \leftarrow b^{[k]} - \alpha \nabla_{b^{[k]}} L_{batch=j}$
    - Forward Propagate using X: compute $A^{[k]}$ for k=1,…,N
    - Predict: compute Y_{pred} and compute accuracy comparing with Y
    - Compute loss using $A^{[N]}$

Loop for fixed number of iterations

Training with Mini-Batch Gradient Descent - Example

- Consider Logistic Regression and data (2 features and 5 data points)
- Assume mini-batches have been created as in Mini-Batch Creation example:

$$X_{batch=0} = \begin{bmatrix} 1 & 2 & 4 \\ -2 & -5 & -8 \end{bmatrix} \qquad Y_{batch=0} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

$$X_{batch=1} = \begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix} \qquad Y_{batch=1} = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

- Assume that initial parameter values are:

$W = [0.1 \quad 0.1] \qquad \mathrm{b} = [0.2]$

- Assume training is performed using Gradient Descent with $\alpha = 0.1$
- Epoch 1 - Mini-batch 0
- Forward Propagation:

$$Z_{batch=0} = WX_{batch=0} + b = [0.1 \quad 0.1] \begin{bmatrix} 1 & 2 & 4 \\ -2 & -5 & -8 \end{bmatrix} + [0.2] = [0.1 \quad -0.1 \quad -0.2]$$

$$A_{batch=0} = f(Z_{batch=0}) = \begin{bmatrix} \frac{1}{1+e^{-0.1}} & \frac{1}{1+e^{0.1}} & \frac{1}{1+e^{0.2}} \end{bmatrix} = [0.5250 \quad 0.4750 \quad 0.4502]$$

Training with Mini-Batch Gradient Descent - Example

- Back Propagation:

$$\nabla_A L_{batch=0} = -\frac{1}{3}\left(\frac{Y_{batch=0}}{A_{batch=0}} - \frac{1-Y_{batch=0}}{1-A_{batch=0}}\right) = -\frac{1}{3}\begin{bmatrix} -\frac{1}{1-0.5250} & \frac{1}{0.4750} & -\frac{1}{1-0.4502} \end{bmatrix}$$
$$= \begin{bmatrix} 0.7017 & -0.7017 & 0.6062 \end{bmatrix}$$

$$\frac{\partial A_j}{\partial Z_j} = A_j - A_j^2 \quad \begin{bmatrix} \frac{\partial A_0}{\partial Z_0} & \frac{\partial A_1}{\partial Z_1} & \frac{\partial A_2}{\partial Z_2} \end{bmatrix} = \begin{bmatrix} 0.2494 & 0.2494 & 0.2475 \end{bmatrix}$$

$$\nabla_Z L_{batch=0} = \nabla_A L_{batch=0} * \begin{bmatrix} \frac{\partial A_0}{\partial Z_0} & \frac{\partial A_1}{\partial Z_1} & \frac{\partial A_2}{\partial Z_2} \end{bmatrix} = \begin{bmatrix} 0.7017 & -0.7017 & 0.6062 \end{bmatrix} *$$
$$\begin{bmatrix} 0.2494 & 0.2494 & 0.2475 \end{bmatrix} = \begin{bmatrix} 0.1750 & -0.1750 & 0.1501 \end{bmatrix}$$

$$\nabla_W L_{batch=0} = \nabla_Z L_{batch=0} X_{batch=0}^T = \begin{bmatrix} 0.1750 & -0.1750 & 0.1501 \end{bmatrix} \begin{bmatrix} 1 & -2 \\ 2 & -5 \\ 4 & -8 \end{bmatrix} = \begin{bmatrix} 0.4252 & -0.6755 \end{bmatrix}$$

$\nabla_b L_{batch=0} = \sum_{j=0}^{m-1} \nabla_Z L_{batch=0,j} = 0.1501$ (sum of entries of $\nabla_b L_{batch=0}$)

- Update the parameters:

$W = W - \alpha \nabla_W L_{batch=0}$= $[0.1 \quad 0.1]$-0.1*$[0.4252 \quad -0.6755]$= $[0.0575 \quad 0.1675]$

$b = b - \alpha \nabla_b L_{batch=0}$= $[0.2]$-0.1*$[0.1501]$= $[0.1850]$

Training with Mini-Batch Gradient Descent - Example

- Epoch 1 - Mini-batch 1
- Forward Propagation:

$$Z_{batch=1} = WX_{batch=1} + b = \begin{bmatrix} 0.0575 & 0.1675 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix} + \begin{bmatrix} 0.1850 \end{bmatrix} = \begin{bmatrix} -0.0926 & 0.4626 \end{bmatrix}$$

$$A_{batch=1} = f(Z_{batch=1}) = \begin{bmatrix} \frac{1}{1+e^{0.0926}} & \frac{1}{1+e^{-0.4626}} \end{bmatrix} = \begin{bmatrix} 0.4769 & 0.6136 \end{bmatrix}$$

- Back Propagation

$$\nabla_A L_{batch=1} = -\frac{1}{2}\left(\frac{Y_{batch=2}}{A_{batch=2}} - \frac{1 - Y_{batch=2}}{1 - A_{batch=2}}\right) = -\frac{1}{2}\begin{bmatrix} \frac{1}{0.4769} & \frac{1}{0.6136} \end{bmatrix} = \begin{bmatrix} -1.0485 & -0.8148 \end{bmatrix}$$

$$\frac{\partial A_j}{\partial Z_j} = A_j - A_j^2 \qquad \begin{bmatrix} \frac{\partial A_0}{\partial Z_0} & \frac{\partial A_1}{\partial Z_1} \end{bmatrix} = \begin{bmatrix} 0.2495 & 0.2371 \end{bmatrix}$$

$$\nabla_Z L_{batch=1} = \begin{bmatrix} -1.0485 & -0.8148 \end{bmatrix} * \begin{bmatrix} 0.2495 & 0.2371 \end{bmatrix} = \begin{bmatrix} -0.2616 & -0.1932 \end{bmatrix}$$

$$\nabla_W L_{batch=1} = \nabla_Z L_{batch=1} X_{batch=1}^T = \begin{bmatrix} -0.2616 & -0.1932 \end{bmatrix} \begin{bmatrix} 1 & -2 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} -0.0684 & 0.1368 \end{bmatrix}$$

$\nabla_b L_{batch=1} = \sum_{j=0}^{m-1} \nabla_Z L_{batch=1,j} = -0.4548$ (sum of entries of $\nabla_b L_{batch=1}$)

- Update the parameters:

$W = W - \alpha \nabla_W L_{batch=1}$= $[0.0575 \quad 0.1675]$-0.1*$[-0.0684 \quad 0.1368]$= $[0.0643 \quad 0.1539]$

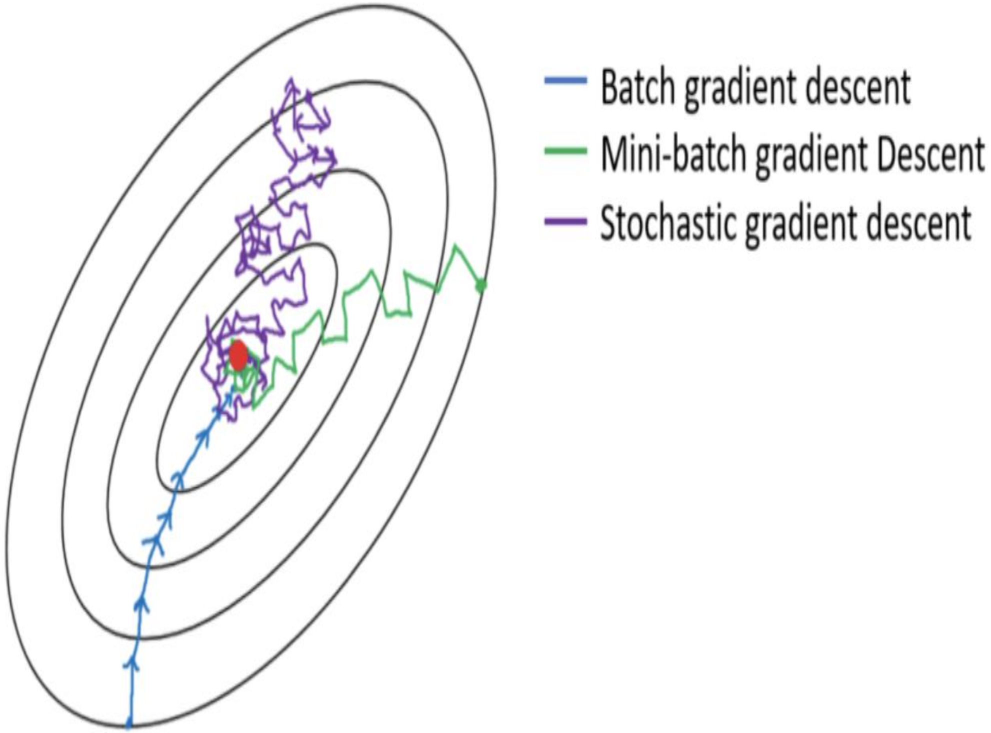
$b = b - \alpha \nabla_b L_{batch=1}$= $[0.1850]$-0.1*$[-0.4548]$= $[0.2305]$

Mini-Batch Gradient Descent

- If there are n mini-batches, then parameters are updated n times for each epoch
- Can compute Loss over all samples after each mini-batch update or after all mini-batch updates at end of each epoch
- Can compute training accuracy after each mini-batch update or after all mini-batch updates at end of each epoch

Summary

Batch Gradient Descent (BGD)	Stochastic Gradient Descent (SGD)	Mini-Batch Gradient Descent (MGD)
• Update $W^{[k]}$ and $b^{[k]}$ once per epoch • Computationally efficient, in general, but may be issues with large datasets • Usually stable learning path • May converge to a local minimum if loss function not convex	• Update $W^{[k]}$ and $b^{[k]}$ m (number of samples) times per epoch • Typically more computationally expensive than BGD • Noisiest learning path • Noisy learning process may help avoid local minimum	• Update $W^{[k]}$ and $b^{[k]}$ Ceil(m/batchSize) times per epoch • Computational expense between SGD and BGD • Noisy learning path • Noisy learning process may help avoid local minimums



Picture source (with permission): https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Jupyter Notebook Demo

- See files
  - IntroML/Examples/Chapter6/StochasticGradientDescent.ipynb
  - IntroML/Examples/Chapter6/MiniBatchGradientDescent.ipynb

Chapter 6.2 Momentum, RmsProp, and Adam Optimization

Momentum, RmsProp, and Adam Optimization

Goal of this Section:

- Present additional optimization algorithms:
  - Momentum
  - RmsProp
  - Adam

Optimization

- Gradient Descent is a rudimentary optimization algorithm
- Not guaranteed to converge to global minimum or may converge slowly or may get stuck near local minimum or not converge at all
- Has fixed learning rate α – may want to use larger learning rate at start of training and then move to smaller learning rate later in training
- Enhancements to Gradient Descent have been developed to address these issues and improve convergence
- In this section discuss additional optimization approaches
- Can apply mini-batch optimization to these methods as well

Optimization Algorithm

Let W = [W_0 W_1 W_2 W_{d-1}] denote parameter vector of variables. Let L(W)= L(W_0,W_1,W_2 ,…, W_{d-1}) be a function of these parameters

Make initial guess $W_{epoch=0}$ for parameters

1. Loop epoch i = 1, 2, 3, …

- Compute gradient vector $\nabla_W L_{epoch=i-1}$ at $W_{epoch=i-1}$
- Compute $\boxed{\text{Update}_{epoch=i}}$ ← Depends on approach
- Compute new epoch using formula: $W_{epoch=i}$<-$W_{epoch=i-1}$ + $\text{Update}_{epoch=i}$
- Compute $\mathrm{L}(\mathrm{W}_{epoch=i})$

Loop for fixed number of epochs (or if L(W) reduced sufficiently)

Momentum

- Momentum algorithm has a 2-step update formula:

$v_{epoch=i} = \beta v_{epoch=i-1} + \nabla_W L_{epoch=i-1}$ $\quad v_{epoch=0} = 0$

$Update_{epoch=i} = -\alpha v_{epoch=i}$

where $0 \leq \beta < 1$

- v is a weighted average of gradients at all previous epochs
- Motivation for momentum is to avoid getting stuck in a local minimum
  - See blogposts on Momentum approach for more details

Reference:

Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (8 October 1986). "Learning representations by back-propagating errors". Nature. 323 (6088): 533–536. Bibcode:1986Natur.323..533R. doi:10.1038/323533a0.

Momentum - Example

- Consider simple function (has minimum at [0 0])

$L(W) = L(W_0, W_1) = 2W_0^2 + W_1^2$ with gradient $\nabla_W L = [4W_0 \quad 2W_1]$

- Choose α=0.1, β=0.9, and set $v_{epoch=0}$=0 and make initial guess

$W_{epoch=0} = [2\ 2] \qquad L(W_{epoch=0}) = 2 * 2^2 + 2^2 = 12$

- Epoch 1:

$\nabla_W L_{epoch=0} = \nabla_W L\left(W_{epoch=0}\right) = \ [4W_0 \quad 2W_1] = [8\ \ 4]$

$v_{epoch=1} = \beta v_{epoch=0} + \nabla_W\, L_{epoch=0} \qquad v_{epoch=1} = [8\ \ 4]$

$W_{epoch=1} = W_{cpoch=0} - \alpha v_{epoch=1} = [2 \quad 2] - 0.1 * [8 \quad 4] = [1.2 \quad 1.6]$
$L(W_{epoch=1}) = 2 * 1.2^2 + 1.6^2 = 5.44$

- Epoch 2:

$\nabla_W L(W_{epoch=1}) = \ [4W_0 \quad 2W_1] = [4.8 \quad 3.2]$

$v_{epoch=2} = \beta v_{epoch=1} + \nabla_W\, L_{epoch=1} \qquad v_{epoch=2} = 0.9 * [8\ \ 4] + [4.8 \quad 3.2] = [12 \quad 6.8]$

$W_{epoch=2} = W_{epoch=1} - \alpha\, v_{epoch=2} \ = [1.2 \quad 1.6] - 0.1 * [12 \quad 6.8] = [0 \quad 0.92]$
$L(W_{epoch=2}) = +0.92^2 = 0.8464$

RMSProp

- RmsProp algorithm has a 2-step update formula:

$$v_{epoch=i} = \beta v_{epoch=i-1} + (1-\beta)\nabla_W L^2_{epoch=i-1} \qquad v_{epoch=0} = 0$$

$$Update_{epoch=i} = -\frac{\alpha}{\sqrt{v_{epoch=i}} + \epsilon}\nabla_W L_{epoch=i-1}$$

- v represents moving average of square of gradient for each entry
- Learning rate is divided by square root of this moving average – hence acronym RMS (Root Mean Square)
- “Effective” learning rate decreases as moving average increases
- ε included to avoid division by 0
- Typical parameter values: β=0.9 and $\varepsilon = 10^{-8}$

Reference: unpublished work(Hinton, Srivastava & Swersky)
http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

RmsProp - Example

- Consider simple function (has minimum at [0 0])

$L(W) = L(W_0, W_1) = 2W_0^2 + W_1^2$ with gradient $\nabla_W L = [4W_0 \quad 2W_1]$

- Choose α=0.1, $\beta = 0.9$, $\varepsilon = 0$ and initial guess

$W_{epoch=0} = [2\ 2] \qquad L(W_{epoch=0}) = 2 * 2^2 + 2^2 = 12$

Epoch 1:

$\nabla_W L_{epoch=0} = \nabla_W L\left(W_{epoch=0}\right) = [4W_0 \quad 2W_1] = [8\ \ 4] \qquad \nabla_W L_{epoch=0}^2 = [64 \quad 16]$

$v_{epoch=1} = \beta v_{epoch=0} + (1-\beta)\ \nabla_W L_{epoch=0}^2 = 0.1 * [64 \quad 16] = [6.4 \quad 1.6]$

$W_{epoch=1} = W_{epoch=0} - \frac{\alpha}{\sqrt{v_{epoch=1}} + \epsilon} \nabla_W L_{epoch=0} = [2 \quad 2] - 0.1 \left[\frac{8}{\sqrt{6.4}} \quad \frac{4}{\sqrt{1.6}}\right]$

$= [1.6838 \quad 1.6838]$

Component-wise square

$L(W_{epoch=1}) = 2 * 1.6838^2 + 1.6838^2 = 8.51$

Component-wise
square root and division

RmsProp - Example

Epoch 2:

$\nabla_W L_{epoch=1} = \nabla_W L\left(W_{epoch=1}\right) = \ [4W_0 \quad 2W_1] = [6.7351 \quad 3.3675] \quad \nabla_W L^2_{epoch=1} = [45.3614 \quad 11.3406]$

$v_{epoch=2} = \beta v_{epoch=1} + (1-\beta)\nabla_W L^2_{epoch=1} = 0.9 * [6.4. \quad 1.6] + 0.1[45.3614 \quad 11.3406] = [10.2961 \quad 2.5740]$

$W_{epoch=2} = W_{epoch=1} - \frac{\alpha}{\sqrt{v_{epoch=1}} + \epsilon}\nabla_W L_{epoch=1} = [1.6838 \quad 1.6838] - 0.1\left[\frac{6.7351}{\sqrt{10.2961}} \quad \frac{3.3675}{\sqrt{2.5740}}\right]$
$= [1.4739 \quad 1.4739]$

$L(W_{epoch=2}) = 2 * 1.4739^2 + 1.4739^2 = 6.5169$

Adam

- Adam algorithm has a 3-step update formula:

$$m_{epoch=i} = \beta_1 m_{epoch=i-1} + (1-\beta_1)\nabla_W L_{epoch=i-1} \qquad m_{epoch=0} = 0$$

$$v_{epoch=i} = \beta_2 v_{epoch=i-1} + (1-\beta_2)\nabla_W L^2_{epoch=i-1} \qquad v_{epoch=0} = 0$$

$$Update_{epoch=i} = -\frac{\alpha}{\sqrt{v_{epoch=i}} + \varepsilon} m_{epoch=i}$$

- v represents moving average of square of gradient
- Like RmsProp, learning rate is divided by this moving average
- “Effective” learning rate decreases as moving average increases
- m represents moving average of gradient
- Typical parameter values: β_1=0.9, β_2=0.999, and $\varepsilon = 10^{-8}$

Reference: (Kingma & Ba)
https://arxiv.org/abs/1412.6980v8

Adam - Example

- Consider simple function (has minimum at [0 0])

$L(W) = L(W_0, W_1) = 2W_0^2 + W_1^2$ with gradient $\nabla_W L = [4W_0 \quad 2W_1]$

- Choose α=0.1, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 0$ and set $v_{epoch=0}$=0, $m_{epoch=0}$=0 and make initial guess:

$W_{epoch=0} = [2\ 2] \qquad L(W_{epoch=0}) = 2 * 2^2 + 2^2 = 12$

Epoch 1:

$\nabla_W L_{epoch=0} = \nabla_W L\left(W_{epoch=0}\right) = [4W_0 \quad 2W_1] = [8\ \ 4] \qquad \nabla_W L_{epoch=0}^2 = [64 \quad 16]$

$m_{epoch=1} = \beta_1 m_{epoch=0} + (1-\beta_1)\nabla_W L_{epoch=0} = 0.1 * [8 \quad 4] = [0.8 \quad 0.4]$

$v_{epoch=1} = \beta_2 v_{epoch=0} + (1-\beta_2)\nabla_W L_{epoch=0}^2 = 0.001 * [64 \quad 16] = [0.064 \quad 0.016]$

Component-wise square

$$W_{epoch=1} = W_{epoch=0} - \frac{\alpha}{\sqrt{v_{epoch=1}} + \epsilon} m_{epoch=1} = [2 \quad 2] - 0.1\left[\frac{0.8}{\sqrt{0.064}} \quad \frac{0.4}{\sqrt{0.016}}\right] = [1.6838 \quad 1.6838]$$

$L(W_{epoch=1}) = 2 * 1.6838^2 + 1.6838^2 = 8.51$

Component-wise
square root and division

Adam - Example

Epoch 2:

$\nabla_W L_{epoch=1} = \nabla_W L\left(W_{epoch=1}\right) = \ [4W_0 \quad 2W_1] = [6.7351 \quad 3.3675] \quad \nabla_W L^2_{epoch=1}$= $[45.3614 \quad 11.3406]$

$m_{epoch=2} = \beta_1 m_{epoch=1} + (1-\beta_1)\nabla_W L_{epoch=1}$= 0.9* $[0.8 \quad 0.4] +$ 0.1* $[6.7351 \quad 3.3675] = [1.3935 \quad 0.6968]$

$v_{epoch=2} = \beta v_{epoch=1} + (1-\beta)\nabla_W L^2_{epoch=1} = 0.999 * [0.064 \quad 0.016] + 0.001 * [45.3614 \quad 11.3406] = [0.1093 \quad 0.0273]$

$$W_{epoch=2} = W_{epoch=1} - \frac{\alpha}{\sqrt{v_{epoch=1}} + \epsilon} m_{epoch=1} = [1.6838 \quad 1.6838] - 0.1\left[\frac{1.3935}{\sqrt{0.1093}} \quad \frac{0.6968}{\sqrt{0.0273}}\right] = [1.4418 \quad 1.4418]$$

$L(W_{epoch=2}) = 2 * 1.4418^2 + 1.4418^2 = 6.2363$

Optimization: Summary of Algorithms

Name	Description
Optimization	Optimization algorithms have general form: $W_{epoch=i} = W_{epoch=i-1} + Update_{epoch=i}$ Each algorithm will have a different formula for Update None of these algorithms is guaranteed to converge to local or absolute minimum
Gradient Descent	$Update_{epoch=i} = -\alpha \nabla_W L_{epoch=i-1}$
Momentum	$v_{epoch=i} = \beta v_{epoch=i-1} + \nabla_W L_{epoch=i-1}$ start with $v_{epoch=0}$ =0 $Update_{epoch=i} = -\alpha v_{epoch=i}$
RmsProp	$v_{epoch=i} = \beta v_{epoch=i-1} + (1-\beta) \nabla_W L^2_{epoch=i-1}$ start with $v_{epoch=0}$ =0 $Update_{epoch=i} = -\alpha \frac{\nabla_W L_{epoch=i-1}}{\sqrt{v_{epoch=i}} + \epsilon}$
Adam	$m_{epoch=i} = \beta_1 m_{epoch=i-1} + (1-\beta_1) \nabla_W L_{epoch=i-1}$ start with $m_{epoch=0}$ =0 $v_{epoch=i} = \beta_2 v_{epoch=i-1} + (1-\beta_2) \nabla_W L^2_{epoch=i-1}$ start with $v_{epoch=0}$ =0 $Update_{epoch=i} = -\alpha \frac{m_{epoch=i}}{\sqrt{v_{epoch=i}} + \varepsilon}$

Optimization – Jupyter Notebook Demo

- Open files
  - IntroML/Examples/Chapter6/Momentum.ipynb
  - IntroML/Examples/Chapter6/RmsProp.ipynb
  - IntroML/Examples/Chapter6/Adam.ipynb

Chapter 6.3 Code Walkthrough Version 3.1

Code Walkthrough Version 3.1

Goal of this Section:

- Walkthrough addition of mini-batch optimization
- Walkthrough addition of Momentum, RmsProp, and Adam optimizers

Coding Walkthrough: Version 3.1 To Do

File/Component	To Do
NeuralNetwork_Base	Add method to create mini-batches
NeuralNetwork_Base	Update fit method to perform mini-batch optimization
Optimizer	Add derived classes for Momentum, RmsProp, and Adam optimizers
driver_neuralnetwork_binary driver_neuralnetwork_multi	Update drivers to use mini-batch optimization and additional optimizers

NeuralNetwork_Base Class – Methods

Method	Input	Description
mini_batch	X (numpy array) Y (numpy array) batch_size (integer)	Method to create mini-batches of specified size Return: List of tuples (Xbatch,Ybatch) one for each mini-batch
fit	X (numpy array) Y (numpy array) epochs (integer) **kwargs batch_size (integer)	Update the existing fit method to generate mini-batches by calling mini-batch method and then train using the mini-batches Return: dictionary containing loss and accuracy histories

Momentum, RmsProp, Adam - Methods

Momentum: methods	Input	Description
__init__	learning_rate (float) beta (float)	Takes in relevant parameters and initializes v Return: nothing
update	gradient (numpy array)	Computes update to be used by optimization algorithm. Input is gradient Return: update

RmsProp: methods	Input	Description
__init__	learning_rate (float) beta (float) epsilon (float)	Takes in relevant parameters and initializes v Return: nothing
update	gradient (numpy array)	Computes update to be used by optimization algorithm. Input is gradient Return: update

Adam: methods	Input	Description
__init__	learning_rate (float) beta1 (float) beta2 (float) epsilon (float)	Takes in relevant parameters and initializes m and v Return: nothing
update	gradient (numpy array)	Computes update to be used by optimization algorithm. Input is gradient Return: update

Why Use Multiple Instances of Optimizer?

- Why use separate instance of optimizer object for each parameter matrix: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots, W^{[N]}, b^{[N]}$ in code?
- Momentum, RmsProp, and Adam methods require info from previous iteration to compute $Update_{epoch=i}$ not just $\nabla_W L_{epoch=i-1}$ and $\nabla_b L_{epoch=i-1}$
- This information depends on the parameter and can’t be mixed with information for other parameters – can’t mix histories for $W^{[1]}$ and $b^{[1]}$ or $W^{[1]}$ and $W^{[2]}$

Code Version 3.1 Walkthrough

- Code for this walkthrough located at:

IntroML/Code/Version3.1

- You can implement the changes suggested in this section by starting with a clean Version2.2 of the code
    - Have a look at Version 3.1 for hints
- We will walkthrough changes and show how these optimization updates can significantly improve training performance

Chapter 6.4 Validation

Validation

Goal of this Section:

- Review of validation and testing approaches

Effectiveness of Neural Network Models

- In code examples, we have used loss for training data and accuracy of prediction for the training data as a measure of effectiveness
- This provides some information but is not a reliable measure of model effectiveness
  - In trivial case, can create a model that is simply a look-up table based on training data (looks up output label based on input features in training data)
  - Accuracy is 100% for predicting results for input data in training set
  - This model is useless for prediction if new input is not in training data
- Effectiveness of model measured by how well predictions match actual results for data not used in training

Train, Validation, and Test Data Sets

Train

Validation

Test

Dataset	Description
Train	Data used to fit the model (Feature Matrix X and Label Vector Y in notation of this course)
Validation	Data separate from the training data that is not used in training. Loss/Accuracy calculations are performed for this dataset to see how well model performs at prediction for data not used in training. As we will see later in this chapter use this dataset to measure performance of as machine learning system set up is varied: including number of layers, number of units, optimization parameters, $\alpha, \beta, \beta_1, \beta_2, \varepsilon$, etc. (This is called hyperparameter tuning – see Section 6.8.)
Test	The sample of data not used in training or in tuning. This dataset is used to measure performance after machine learning set-up is finalized.
	Source: Towards Data Science https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7
Dataset split	No definitive answer in literature. In this course we will focus on train and validation and use roughly 80-90%/20-10% split

Training Algorithm With Validation

Assume Neural Network with N layers

Input training data: feature matrix Xtrain and values Ytrain

Input validation data: feature matrix Xvalid and values Yvalid

Choose batch size b and generate mini-batches $(Xtrain_{batch=j}, Ytrain_{batch=j})$, j=0,…,n-1

Make initial guess for parameters: $W^{[k]}$ and $b^{[k]}$ for k=1,…,N

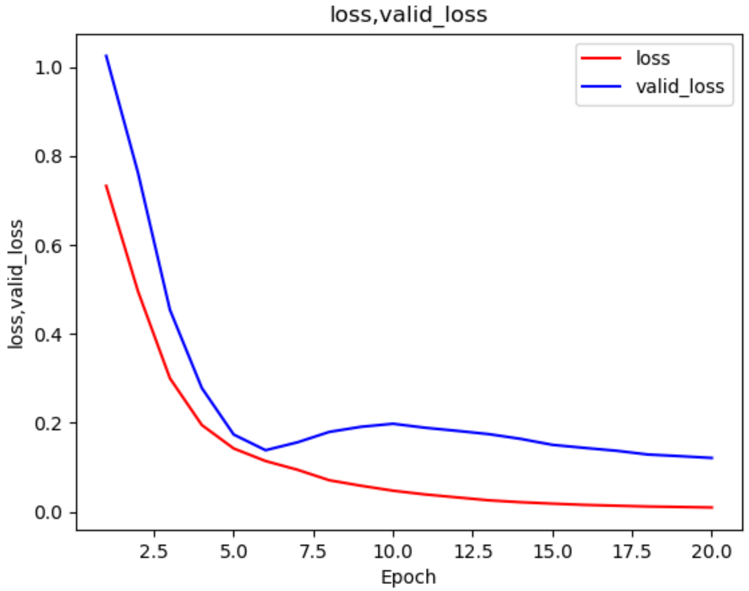
Choose optimizer and parameters: $\alpha, \beta, \beta_1, \beta_2, \varepsilon,$ etc

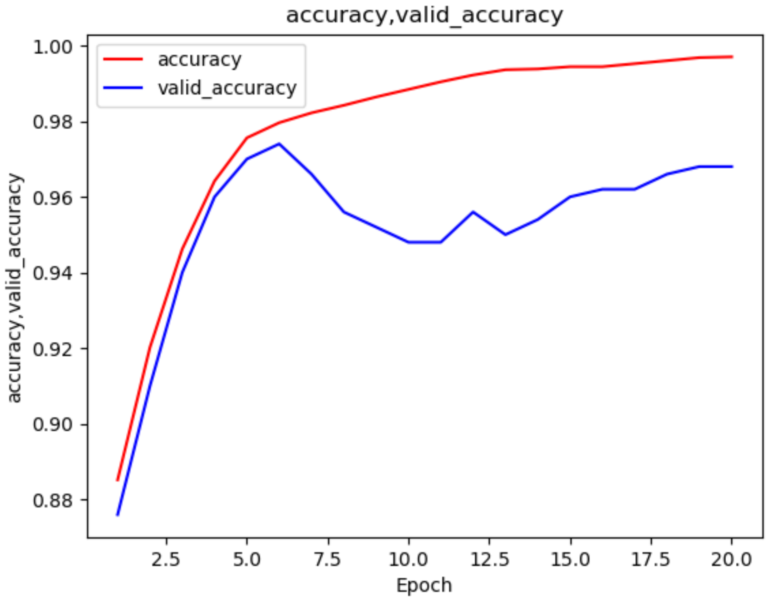
1. Loop for epoch i = 1, 2, …
   - Loop over mini-batches j=0,…,n-1
     - Forward Propagate using $Xtrain_{batch=j}$: compute $A^{[k]}_{batch=j}$ for k=1,…,N
     - Back Propagate using $Xtrain_{batch=j}, Ytrain_{batch=j}$: compute $\nabla_{W^{[k]}} L_{batch=j}$, $\nabla_{b^{[k]}} L_{batch=j}$ k=1,…,N
     - Update parameters: $W^{[k]}$ and $b^{[k]}$ for k=1,…,N
   - Forward Propagate using validation feature matrix Xvalid: compute $Avalid^{[k]}$ for k=1,…,N
   - Predict using validation data: compute $Yvalid_{pred}$ and compute validation accuracy comparing with $Yvalid$
   - Compute validation loss using $Avalid^{[N]}$
   - Forward Propagate using training feature matrix Xtrain: compute $Atrain^{[k]}$ for k=1,…,N
   - Predict using training data: compute $Ytrain_{pred}$ and compute training accuracy comparing with $Ytrain$
   - Compute training loss using $Atrain^{[N]}$

Loop for fixed number of iterations

Loss and Accuracy Plots

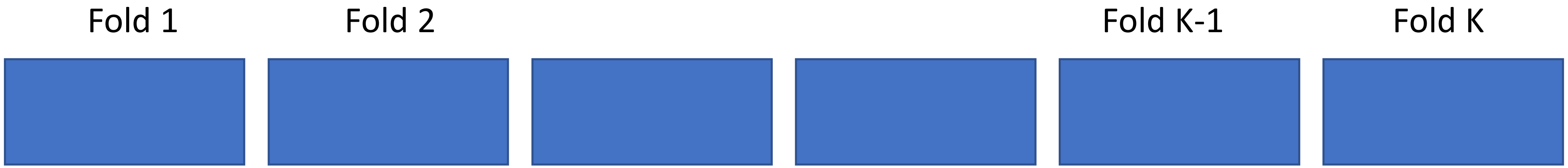
- Loss and Accuracy plots will include histories for both training and validation
- Will discuss how to assess and improve performance using these plots later in chapter





K-Fold Validation

- Using same dataset for training and validation in evaluation process may introduce bias
- K-Fold Validation
  - Randomize training and validation data
  - Split training and validation data into K “Folds” of roughly even size



  - Perform training with folds 1,2,…,K-1 and validation with fold K
  - Repeat: train with folds 2,3,…,K and validation with fold 1
  - Repeat with remaining combinations
  - Validation accuracy is average over all possible combinations
  - With this process each data point will be part of validation once and will be used in training K-1 times

Stratified K-Fold Validation

- Validation Data and Train Data should have same distribution
  - For binary classification, ensure proportion of 0 and 1 labels is same in train and validation data sets
  - For multiclass classification, ensure distribution of classes is same in train and validation data sets
- In K-Fold Validation, each fold should have similar distribution of outputs

K-Fold Validation Algorithm

Split data into K folds

Create K training/validation dataset combinations

1. Loop for combination i=1,…,K
   - Perform training using training dataset $Xtrain_{combination=i}$, $Ytrain_{combination=i}$ and validation dataset $Xvalid_{combination=i}$, $Yvalid_{combination=i}$
   - Save validation accuracy at final epoch for combination=i

Final validation accuracy is average over all validation accuracies for all combinations

Chapter 6.5 Performance Measures

Performance Measures

Goal of this Section:

- Review of performance measures for machine learning

Accuracy Calculation

- Accuracy by itself may not be best metric for measuring effectiveness of neural network
- Example:
  - Consider binary classification of x-ray images (normal versus abnormal)
  - Suppose that training data has few abnormal images (only 2%)
  - Without any neural network, we can create a trivial model: prediction of model is normal for all input images
  - This model will have 98% accuracy but is obviously a lousy model – it will never yield prediction of abnormal
- What are alternatives to accuracy metric for measuring effectiveness?
- References:

https://heartbeat.fritz.ai/evaluation-metrics-for-machine-learning-models-d42138496366

https://en.wikipedia.org/wiki/Confusion_matrix

Precision, Recall and F1 Score

- Accuracy is fraction of positives and negatives predicted correctly

$$Accuracy = \frac{\#TruePositve + \#TrueNegative}{\#Total} = \frac{\#Blue + \#Red}{\#Total}$$

- Precision is fraction of predicted positives that are correct

$$Precision = \frac{\#TruePositive}{\#PredictedPositive} = \frac{\#Blue}{\#Blue{+}\#Green}$$

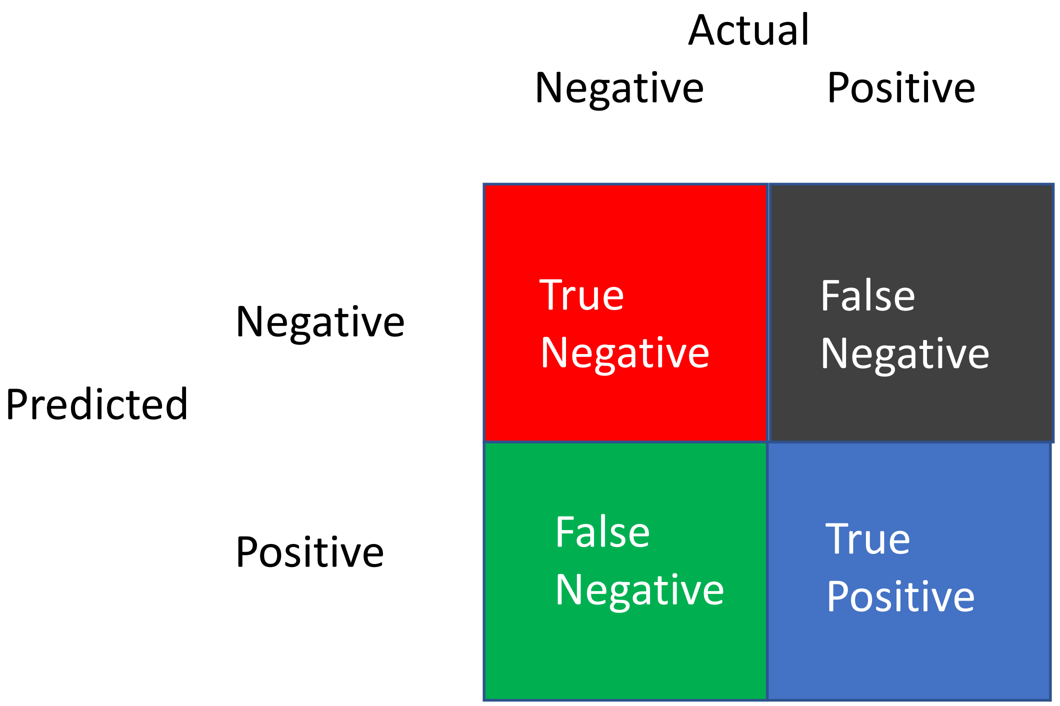
- Recall is fraction of actual positives predicted correctly

$$Recall = \frac{\#TruePositive}{\#ActualPostive} = \frac{\#Blue}{\#Blue{+}\#Black}$$

- F1 score combines into a single figure

$$F1 = 2\frac{Precision * Recall}{Precision + Recall}$$

- For each of these measures, 1 is optimal score



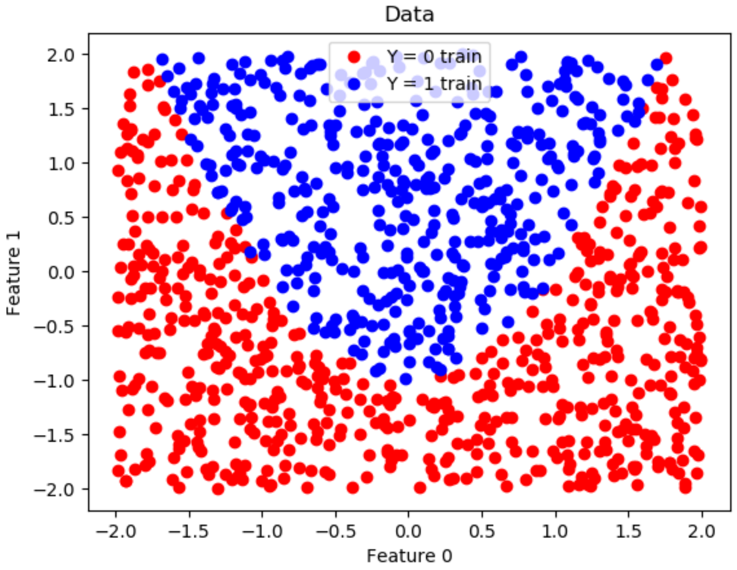
		Actual	
		Negative	Positive
Predicted	Negative	🟥 True Negative	⬛ False Negative
	Positive	🟩 False Negative	🟦 True Positive

Target/Achievable Accuracy

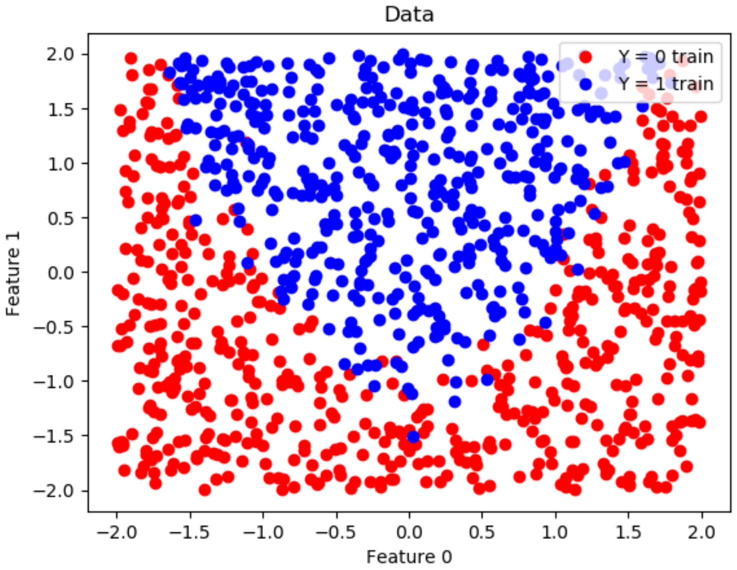
- What accuracy should we expect from machine learning system?
    - For classification, perfect score is 100% accuracy
    - For regression, perfect score is 0 mean absolute error
    - Nearly impossible to achieve these perfect scores
- Machine learning models have prediction errors
    - Errors result from incomplete data – feature matrix may not include all relevant features
    - Noise in data
    - Stochastic nature of machine learning algorithm
        - Random choice of parameters W and b
- Target or achievable accuracy is problem dependent
    - For classification of cats and dogs based on photos, target should be close to 100%
    - In general, in case of noisy data, may not be able to achieve 100% accuracy
- In machine learning training and testing, goal should be to achieve target accuracy for training, validation, and test sets

Target Accuracy - Example

- Figure on left:
  - Relatively sharp boundary between 0 and 1 training data points
  - Should expect target accuracy to be close to 100%
- Figure on right:
  - “Fuzzy” boundary between 0 and 1 training data points – noise has been added
  - Expect target/achievable to be less than 100% - not clear what label should be for new data points near boundary



Copyright Satish



Confusion Matrix

- For classification (binary or multiclass), Confusion Matrix is a matrix/table listing counts of actual and predicted for each class
- Example: 3 classes
  - First Row indicates Predicted = 0 for 48 cases. Of these, 45 were actual class 0, 2 were actual class 1, and 1 was actual class 2

	Actual 0	Actual 1	Actual 2
Predicted 0	45	2	1
Predicted 1	3	51	2
Predicted 2	1	1	44

- Useful for determining patterns in mis-classification

Algorithm for Producing Confusion Matrix

Assume training has been performed

Assume prediction Y_pred is made for which actual labels Y are known

Input actual labels Y, predicted labels Y_pred and number of classes c

1. For predicted class p= 0,1,…,c-1
    - Determine indices (idx_p) where Y_pred = p
    - For actual class a = 0,1,…,c-1
        - Count = number of entries where Y[idx_p] = a
        - Confusion Matrix (Row = p, Col = a) = Count

	Actual 0	Actual 1	Actual 2
Predicted 0	45	2	1
Predicted 1	3	51	2
Predicted 2	1	1	44

Chapter 6.6 Code Walkthrough Version 3.2

Coding Walkthrough: Version 3.2

Goal of this Section:

- Add calculation of loss and accuracy for validation data set
- Add Confusion Matrix, Precision, Recall and F1 Score calculations

Coding Walkthrough: Version 3.2 To Do

File/Component	To Do
NeuralNetwork_Base	Add calculation of loss/accuracy for validation data set for each epoch in fit method and add input to turn off listing of epoch level results
metrics	Add function to calculate precision, recall and F1 score
metrics	Add function to print confusion matrix
example_classification	Add functionality to generate a validation dataset and add noise to data
plot_results	Add functionality to plot multiple curves on single loss history or accuracy history plot
plot_results	Add functionality to plot validation data as well as training data in scatter plot

NeuralNetwork_Base Class – Methods

Method/Function	Input	Description
fit	X (numpy array) Y (numpy array) epochs (integer) **kwargs batch_size (integer) verbose (boolean) validation_data (tuple containing feature matrix and values for validation dataset)	Update the existing train method to compute accuracy for validation dataset if provided, and turn off printing of loss and accuracy if verbose is to False Return: dictionary containing loss and accuracy histories for train and validation dataset (if provided)

Metrics

Function	Input	Description
f1score	Y (numpy array) Y_pred (numpy array)	Return: f1score, precision, and recall
confusion_matrix	Y (numpy array) Y_pred (numpy array) nclass (integer)	Prints confusion matrix Return: nothing

example_classification

Function	Input	Description
example	nfeature (integer) m (integer) case (string) nclass (integer) noise (boolean) validpercent (float)	Add functionality to add noise to data (if noise = True). Add functionality to create validation data – number of validation data points is m * validpercent Return: Xtrain,Ytrain,Xvalid,Yvalid

Code Version 3.2 Walkthrough

- Code for this walkthrough located at:

IntroML/Code/Version3.2

- You can implement the changes suggested in this section by starting with a clean Version3.1 of the code
  - Have a look at Version 3.2 for hints
- Rest of lecture is a walkthrough of changes and discussion of results

Chapter 6.7 Regularization

Regularization

Goal of this Section:

- Discuss purpose and approaches for regularization

Purpose of Regularization and Approaches

Purpose of Regularization:

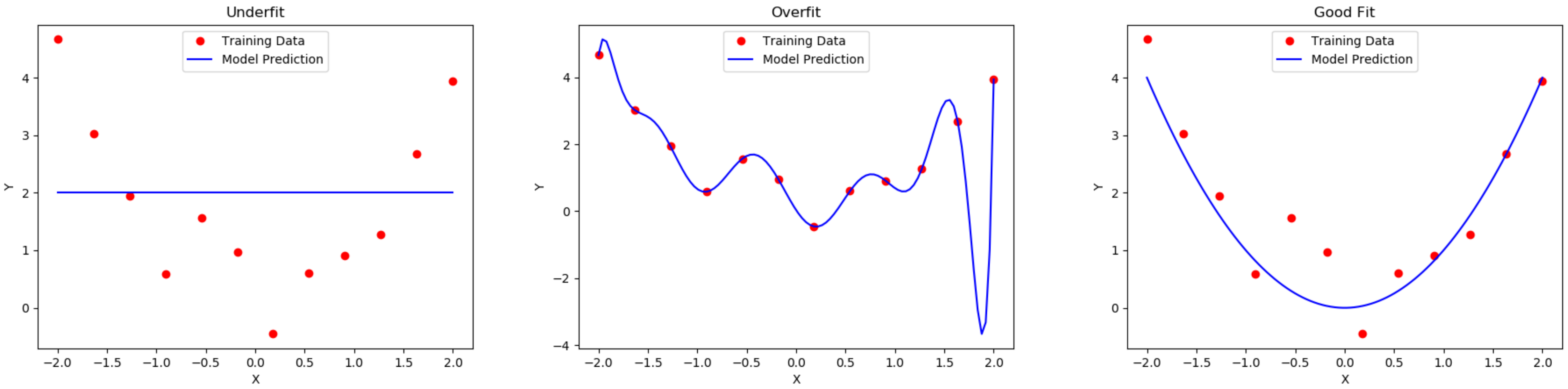
- Technique for solving ill-posed problems/avoid overfitting
- Tool to improve machine learning performance

Approaches for Regularization:

- See following article for a good discussion:
- https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/
    - Add “information” to problem
        - L1 regularization
        - L2 regularization
    - Stop training algorithm early
    - Dropout (remove some nodes in layers randomly)

Underfitting and Overfitting - Definitions

- Underfitting
  - Occurs when function structure cannot capture the trend of the data
  - Example: Linear function attempting to fit “quadratic” data
- Overfitting
  - Occurs when training yields function that captures training data extremely well but does not generalize to data not seen in training – fits training data well but does not capture “trend” of data
  - Example: 11th degree polynomial used to fit 12 data points – fits training data exactly, but large discrepancy at end point does not capture trend of data
- Good/Robust Fit:
  - Occurs when training yields function that captures training data well and also generalizes to data not seen in training
  - Example: Model prediction captures training data and generalizes to data not seen in training



Copyright Satish Reddy 2020

L1 and L2 Regularization

- L1,L2 Regularization involve adding “penalty” term to loss function

$\hat{L} = regular\ loss + Penalty = L + P$

- L1 Regularization:

$$Penalty = \sum_{k=1}^{N} \lambda^{[k]} \sum_{i=0}^{n^{[k]}-1} \sum_{j=0}^{n^{[k-1]}-1} \left| W_{ij}^{[k]} \right| \qquad \text{(Lasso)}$$

- L2 Regularization:

$$Penalty = \sum_{k=1}^{N} \lambda^{[k]} \sum_{i=0}^{n^{[k]}-1} \sum_{j=0}^{n^{[k-1]}-1} \left(W_{ij}^{[k]} \right)^2 \qquad \text{(Ridge)}$$

- Constants $\lambda^{[k]} \geq 0$ for k=1,…,N are user specified and may be different for each layer of neural network
- Penalty terms are always non-negative
- Typically don’t include parameters $b^{[k]}$ k=1,…,N in penalty term

Impact of Regularization

- Penalty term forces training algorithm to smaller parameter values $W_{ij}^{[k]}$
- Why is this important?
- Recall forward propagation:

  Loop for k=1,…,N (number of layers)
  - Linear part: $Z^{[k]} = W^{[k]}A^{[k-1]} + b^{[k]} \quad A^{[0]} = X$
  - Activation: $A^{[k]} = f^{[k]}(Z^{[k]})$
- Large entries in $W^{[k]}$ can lead to large changes in $Z^{[k]}$, $A^{[k]}$ for small changes in input X leading to overfitting
- Section 6.10 has much more detail regarding use of regularization to address overfitting. In this section, I want to present the underlying math

L2 Regularization - Example

- Consider binary classification with 2 features and 3 training data points (m=3)

$$X = \begin{bmatrix} 1 & 2 & 4 \\ -2 & -5 & -8 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

- Assume 2 layer neural network: layer 1 (tanh) has 2 units, layer 2 (sigmoid) has 1unit
- Parameter matrices:

$$W^{[1]} = \begin{bmatrix} W_{00}^{[1]} & W_{01}^{[1]} \\ W_{10}^{[1]} & W_{11}^{[1]} \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & -0.5 \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

$$W^{[2]} = \begin{bmatrix} W_{00}^{[2]} & W_{01}^{[2]} \end{bmatrix} = \begin{bmatrix} -1 & 1 \end{bmatrix} \qquad b^{[2]} = \begin{bmatrix} -0.1 \end{bmatrix}$$

- Forward Propagation:

$$A^{[1]} = \begin{bmatrix} 0 & -0.7616 & -0.9051 \\ 0.9640 & 0.9993 & 1.0 \end{bmatrix}, \qquad A^{[2]} = \begin{bmatrix} 0.7035 & 0.8404 & 0.8588 \end{bmatrix}$$

- Binary cross entropy loss function with penalty is:

$$\hat{L} = -\frac{1}{3}\sum_{j=0}^{2} Y_j \ln\left(A_j^{[2]}\right) + \left(1 - Y_j\right) \ln\left(1 - A_j^{[2]}\right) + \lambda^{[1]}(\left(W_{00}^{[1]}\right)^2 + \left(W_{01}^{[1]}\right)^2 + \left(W_{10}^{[1]}\right)^2 + \left(W_{11}^{[1]}\right)^2) + \lambda^{[2]}(\left(W_{00}^{[2]}\right)^2 + \left(W_{01}^{[2]}\right)^2)$$

Impact of L2 Regularization on Back Propagation

- L2 Regularization does not directly change Forward Propagation
- L2 Regularization does change Loss function and changes Back Propagation

$$\hat{L} = regular\ loss + Penalty = L + P$$

- Recall penalty term is:

$$Penalty = P = \sum_{k=1}^{N} \lambda^{[k]} \sum_{i=0}^{n^{[k]}-1} \sum_{j=0}^{n^{[k-1]}-1} \left(W_{ij}^{[k]}\right)^2$$

- Computing derivatives - each term appears quadratically, so

$\dfrac{\partial P}{\partial W_{ij}^{[k]}} = 2\lambda^{[k]} W_{ij}^{[k]}$ which implies $\nabla_{W^{[k]}} P = 2\lambda^{[k]} W^{[k]}$

- In the back propagation algorithm, the only change to gradient formulas is:

$$\nabla_{W^{[k]}} \hat{L} = \nabla_{Z^{[k]}} L A^{[k-1]^T} + \boxed{2\lambda^{[k]} W^{[k]}}$$

Chapter 6.8 Hyperparameter Search

Hyperparameter Search

Goal of this Section:

- Describe hyperparameter searches/tuning

What are Hyperparameters?

Parameters:

- Paramater matrices $W^{[k]}$ and $b^{[k]}$ for k=1,…,N
- Also refer to entries of $W^{[k]}$ and $b^{[k]}$ as parameters
- Learned during training process

Hyperparameters

- Settings/quantities that define neural network structure, regularization or are used to control learning/training process

Component	Hyperparameters
Neural Network	Number of hidden layers Units per hidden layer Activation function (not strictly a hyperparameter)
fit method	Number of epochs Batch size
Optimization	Gradient Descent: learning rate α Momentum: learning rate α, β RmsProp: learning rate α, β, ε Adam learning rate α, β_1, β_2 ε
L2 Regularization	Regularization multipliers $\lambda^{[k]}$ for layers k=1,…,N

Hyperparameter Search

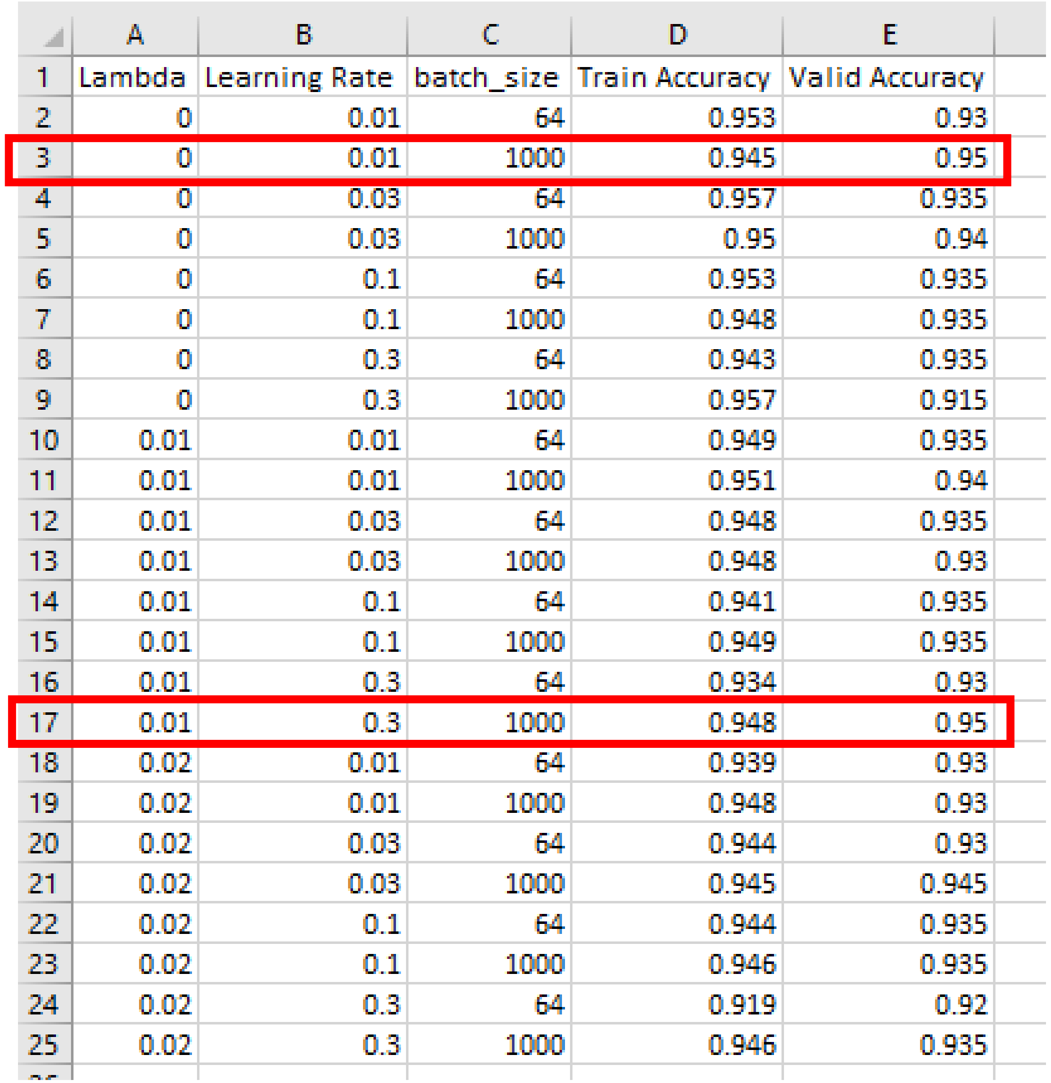
Hyperparameter search process

- Define training and validation datasets
- Create Neural Network Model
- Define optimization method
- Search for “Best” Hyperparameter combination
  - Perform training for each hyperparameter combination
  - Measure accuracy for validation dataset
- Choose Hyperparameter combination with highest validation accuracy

Hyperparameter Search - Example

- Example: Dataset
  - 1000 samples in training dataset
  - 200 samples in validation dataset
  - Binary classification: 2 features “quadratic” example with noise
  - Feature matrix is 2x1000 for training and 2x200 for validation
- Neural Network
  - 5 layer neural network (15, 11, 8, 4, 1 units in layers)
  - tanh activation in layers 1,2,3,4 and sigmoid in final layer
  - L2 regularization with variable $\lambda^{[k]}$
- Optimization:
  - Binary cross entropy loss function
  - Momentum with fixed β=0.9 and variable learning rate α
  - 300 epochs
  - Mini-batch with variable batch size
- Search parameters
  - L2 regularization parameter (same value for all layers) $\lambda^{[k]}$ = 0, 0.01, 0.02
  - Learning rate α = 0.01, 0.03, 0.1, 0.3
  - Mini-batch size = 64, 1000

Hyperparameter Search - Example



	A	B	C	D	E
1	Lambda	Learning Rate	batch_size	Train Accuracy	Valid Accuracy
2	0	0.01	64	0.953	0.93
3	0	0.01	1000	0.945	0.95
4	0	0.03	64	0.957	0.935
5	0	0.03	1000	0.95	0.94
6	0	0.1	64	0.953	0.935
7	0	0.1	1000	0.948	0.935
8	0	0.3	64	0.943	0.935
9	0	0.3	1000	0.957	0.915
10	0.01	0.01	64	0.949	0.935
11	0.01	0.01	1000	0.951	0.94
12	0.01	0.03	64	0.948	0.935
13	0.01	0.03	1000	0.948	0.93
14	0.01	0.1	64	0.941	0.935
15	0.01	0.1	1000	0.949	0.935
16	0.01	0.3	64	0.934	0.93
17	0.01	0.3	1000	0.948	0.95
18	0.02	0.01	64	0.939	0.93
19	0.02	0.01	1000	0.948	0.93
20	0.02	0.03	64	0.944	0.93
21	0.02	0.03	1000	0.945	0.945
22	0.02	0.1	64	0.944	0.935
23	0.02	0.1	1000	0.946	0.935
24	0.02	0.3	64	0.919	0.92
25	0.02	0.3	1000	0.946	0.935

0.95 validation accuracy

0.95 validation accuracy

Chapter 6.9 Code Walkthrough Version 3.3

Coding Walkthrough: Version 3.3

Goal of this Section:

- Walkthrough code changes for regularization
- Walkthrough of hyperparameter search

Coding Walkthrough: Version 3.3 To Do

File/Component	To Do
NeuralNetwork_Base	Update compute_loss method to include L2 regularization penalty term
LRegression	Update __init__ method to allow input of lambda Update back_propagation method to incorporate L2 regularization change
NeuralNetwork	Update add_layer method to allow input of lambda Update back_propagation method to incorporate L2 regularization change
driver_linearregression driver_logisticregression driver_neuralnetwork_binary driver_neuralnetwork_multiclass unittest_forwardbackpropagation	Update these drivers to include input of lambda value for L2 regularization
driver_neuralnetwork_binary_search driver_neuralnetwork_binary_search_	Create new driver for performing hyperparameter search/tuning
write_csv	Function for writing results to csv

LRegression, NeuralNetwork, NeuralNetwork_Base – Methods

Method	Input	Description
LRegression __init__	nfeature (integer) activation (string) lamb (float)	Update: take input lamb for regularization constant. Save in info[0][“lambda”] variable Return: nothing
LRegression back_propagate	X (numpy array) Y (numpy array)	Update: add $2\lambda W$ in calculation of $\nabla_W L$ Return: nothing
NeuralNetwork add_layer	nunit (integer) activation (string) lamb (float)	Update: take input lamb for regularization constant. Save in info[layer][“lambda”] variable Return: nothing
NeuralNetwork back_propagate	X (numpy array) Y (numpy array)	Update: add $2\lambda^{[k]}W^{[k]}$ for calculation of $\nabla_{W^{[k]}} L$ Return: nothing
NeuralNetwork_Base compute_loss	Y (numpy array)	Update to include L2 regularization penalty term Return: loss (including penalty term)

write_csv

File/Component	Input	Description
write_csv	filename (string) list_out (list of lists)	This program uses the csv.writer functionality to write to file. Variable list_out is a list of lists – list_out=[list1,list2, ….]. Each of list1,list2,… is written to separate row in filename. Return: nothing

Code Version 3.3 Walkthrough

- Code for this walkthrough located at:

IntroML/Code/Version3.3

- You can implement the changes suggested in this section by starting with a clean Version3.2 of the code
  - Have a look at Version 3.3 for hints
- Rest of lecture is a walkthrough of changes and discussion of results

Chapter 6.10 Underfitting and Overfitting

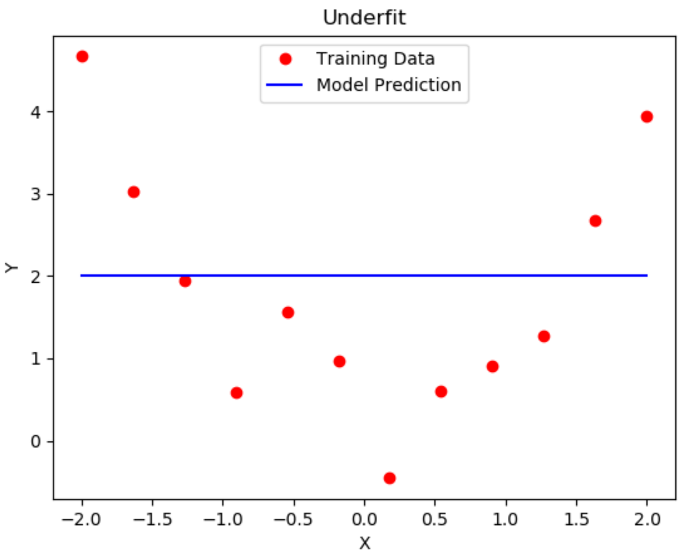
Underfitting and Overfitting

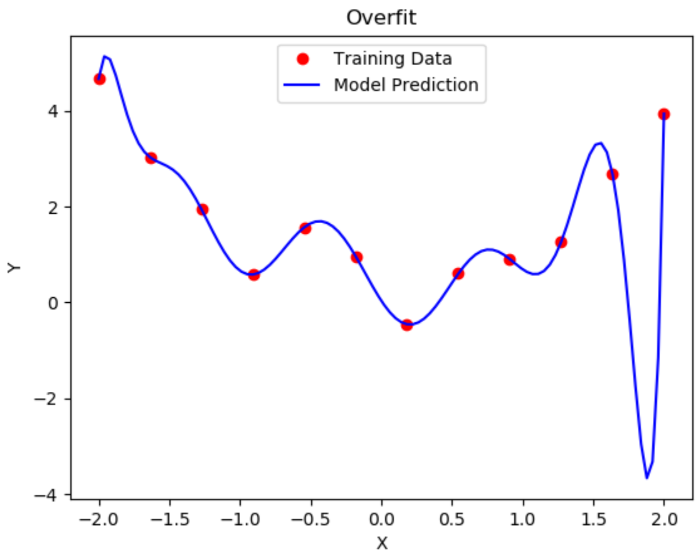
Goal of this Section:

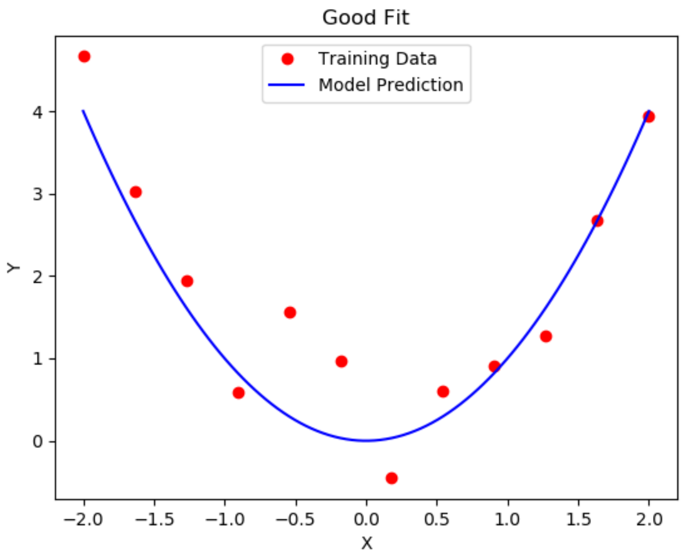
- Define and provide examples of Underfitting and Overfitting
- Show how to diagnose Underfitting and Overfitting
- Present remedies for addressing Underfitting and Overfitting

Underfitting and Overfitting - Definitions

- Underfitting
  - Occurs when function structure cannot capture the trend of the data
  - Example: Linear function attempting to fit “quadratic” data
- Overfitting
  - Occurs when training yields function that captures training data extremely well but does not generalize to data not seen in training – fits training data well but does not capture “trend” of data
  - Example: 11th degree polynomial used to fit 12 data points – fits training data exactly, but large discrepancy at end point does not capture trend of data
- Good/Robust Fit:
  - Occurs when training yields function that captures training data well and also generalizes to data not seen in training
  - Example: Model prediction captures training data and generalizes to data not seen in training







Bias Error and Variance

In Machine Learning literature Bias Error and Variance are used when discussing underfitting and overfitting

Bias Error

- Occurs from erroneous assumptions in learning algorithm. High bias (underfitting) can cause algorithm to miss relationship between input features and target outputs.

Variance

- Sensitivity to small fluctuations in training data. High variance (overfitting) can cause algorithm to capture random noise in training data as opposed to general trend.

Source

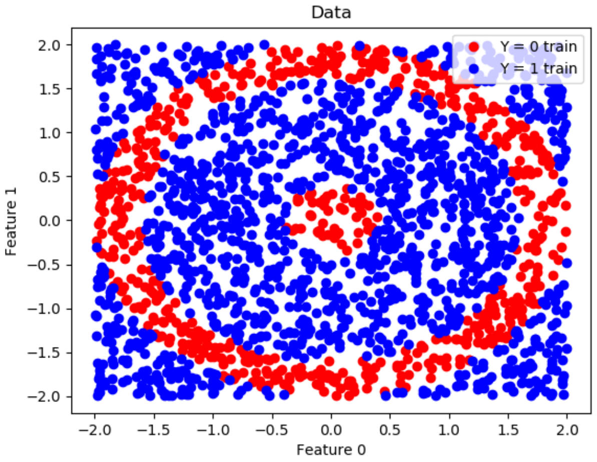
- https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff

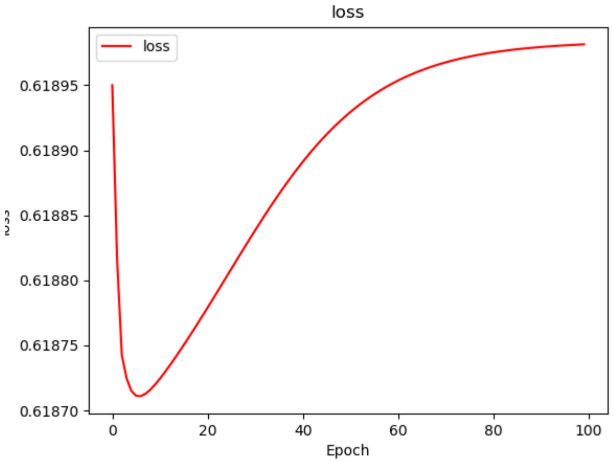
Bias/Variance Tradeoff

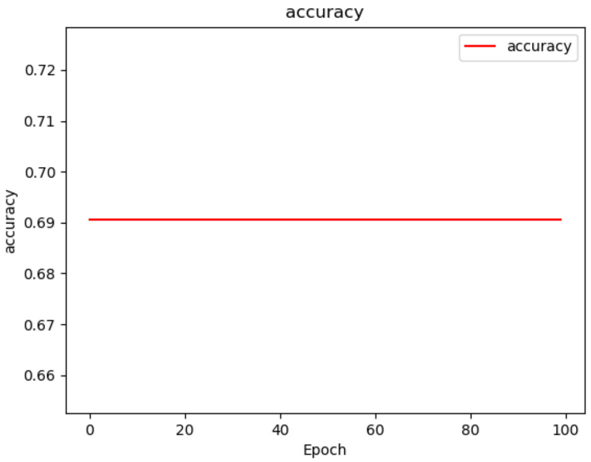
- Models with low bias often have high variance and vice versa
- Goal: find model with low bias and low variance

Underfitting - Example

- Example: Binary Classification for “Bullseye”
  - 1000 training data points chosen from uniform distribution [-2,2]x[-2,2]
  - 200 validation data points chosen from uniform distribution in [-2,2]x[-2,2]
- Machine Learning approach: Logistic Regression
- Logistic Regression has linear boundary – cannot capture Bullseye
  - Heatmap plot shows model predicts 1 for all points
- Symptoms:
  - Training Loss well above 0 – roughly 0.62 in example below
  - Training Accuracy well below target accuracy – stays at 0.69 in example below

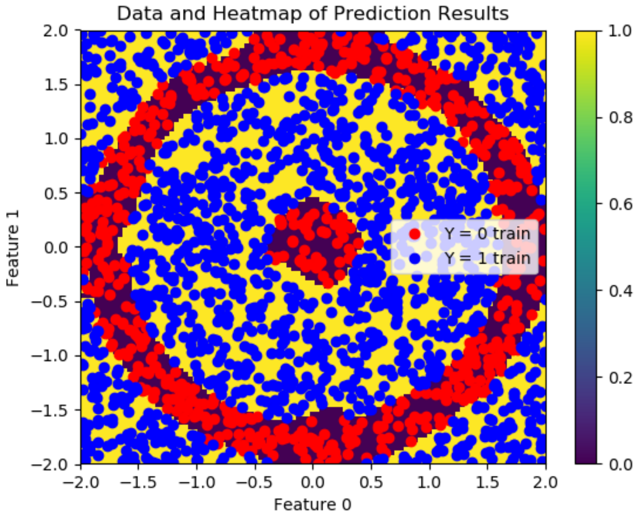


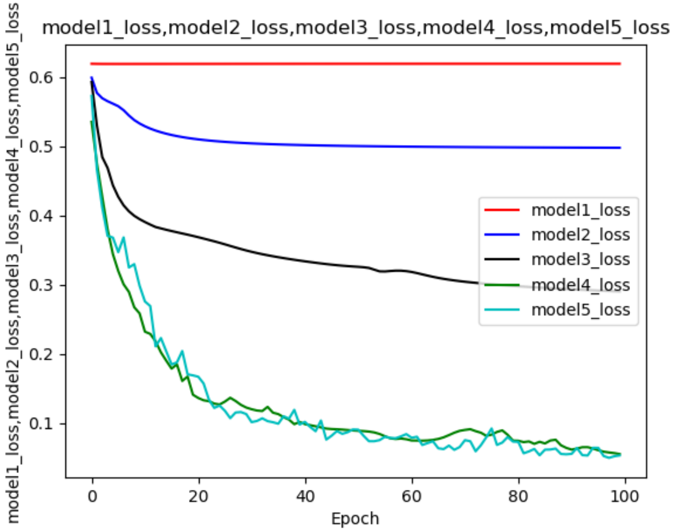


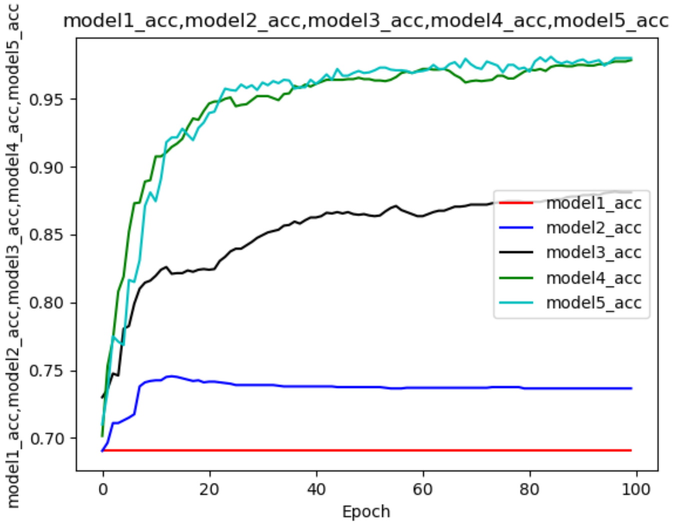


Underfitting - Remedy

- Remedy for underfitting: use “better” model
- 5 Models with increasing number of layers and units
  - Use tanh activation for hidden layers and sigmoid in final layer (train for 100 epochs)
  - Model1: 1 layer (1 unit)
  - Model2: 2 layers (4,1 units)
  - Model3: 3 layers (8,4,1 units)
  - Model4: 4 layers (12,8,4,1 units)
  - Model5: 5 layers (16,12,8,4,1 units)
- Heatmap shows results for Model5
- Model4 and 5 achieve loss <0.1
- Model4 and 5 achieve accuracy of >98%

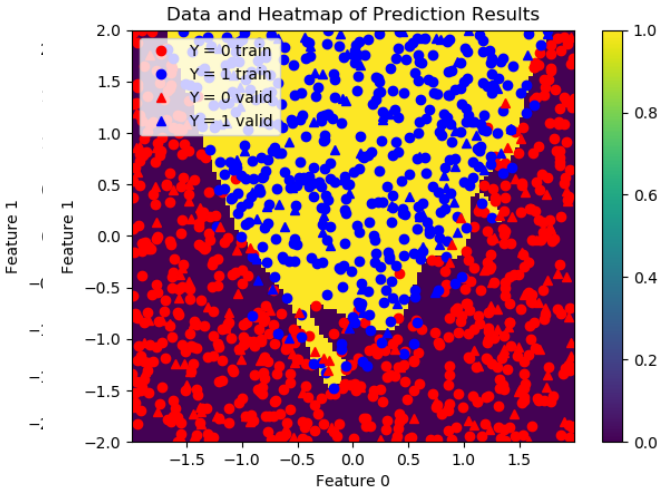


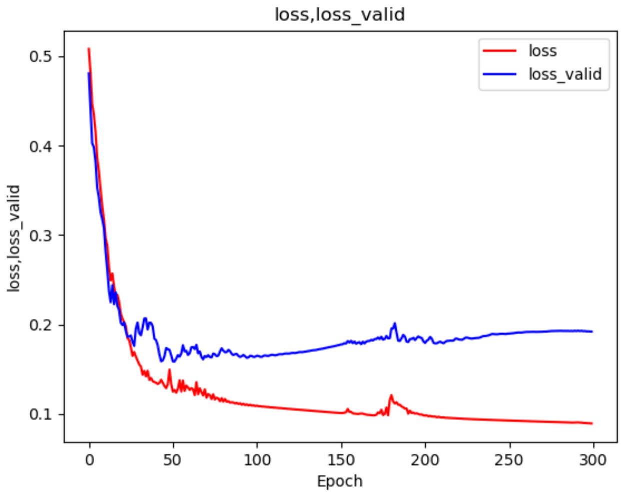


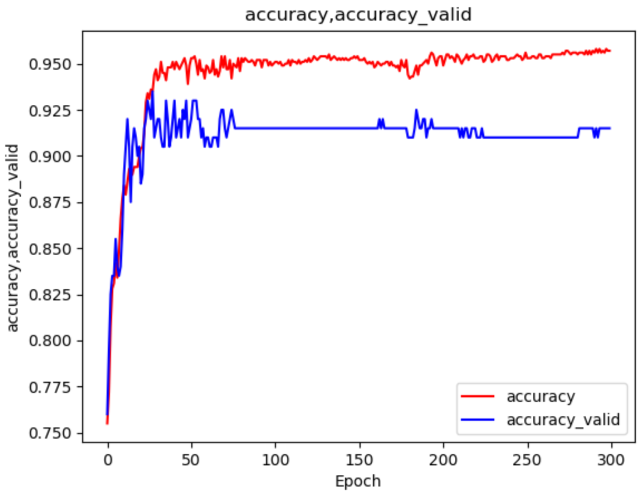


Overfitting - Example

- Example: Binary Classification – Quadratic with Noise
  - 1000 training data points chosen from uniform distribution in [-2,2]x[-2,2]
  - 200 validation data points chosen from uniform distribution in [-2,2]x[-2,2]
- Machine Learning approach:
  - Neural Network with 5 layers (15,11,8,4,1 units)
  - tanh activation for all layers except sigmoid in last
  - Momentum Optimization with α=0.3 and β=0.9
  - Train with 300 epochs and batchsize = 1000 (number training data points)
- Symptoms:
  - Loss for validation dataset greater than that for training dataset (plot below shows that loss increases with epoch for validation dataset)
  - Accuracy for validation dataset less than that for training dataset (Training Accuracy = 0.957, Validation Accuracy = 0.915)







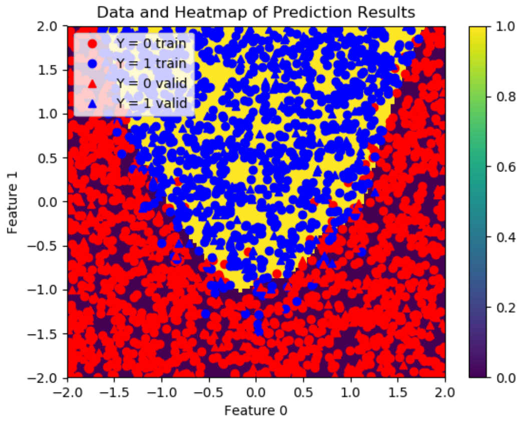
Overfitting - Remedies

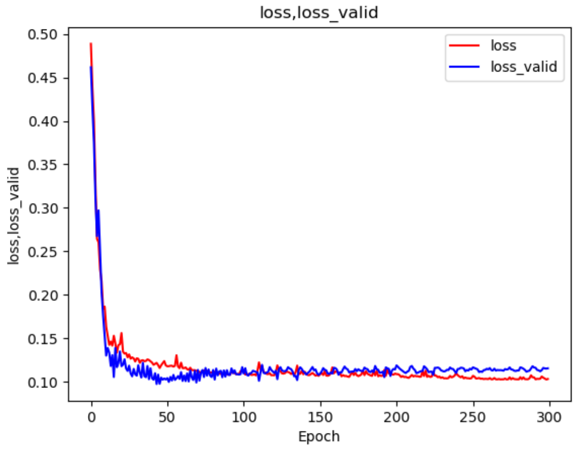
Overfitting Remedies

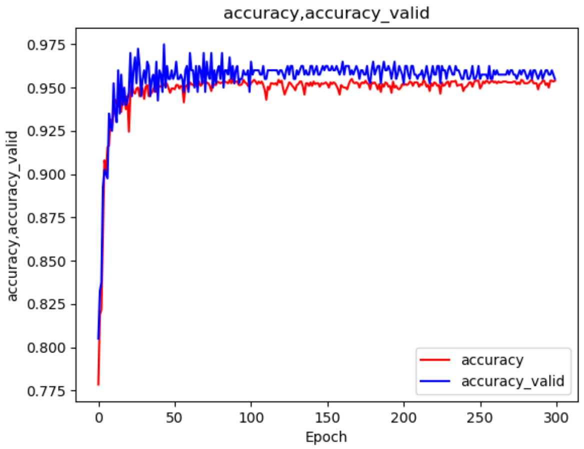
- Use more data points
  - This may not always be possible for in real world setting
- Use simpler function structure
  - In previous example, see “fingers” in model prediction to attempt to fit training points near boundary
  - Fewer layers in neural network should lead to cleaner boundary between 0 and 1 regions
- Regularization
  - Stop training early
  - L2 regularization

Overfitting Remedy – Use More Data

- Example: Binary Classification – Quadratic with Noise – Double Number of Data Points
  - 2000 training data points chosen from uniform distribution in [-2,2]x[-2,2]
  - 400 validation data points chosen from uniform distribution in [-2,2]x[-2,2]
- Machine Learning approach:
  - Neural Network with 5 layers (15,11,8,4,1 units)
  - tanh activation for all layers except sigmoid in last
  - Momentum Optimization with α=0.3 and β=0.9
  - Train with 300 epochs and batch_size = 1000
- Results
  - Heatmap shows smoother boundary between 0 and 1 regions
  - Loss for validation dataset slightly greater than that for training dataset
  - At final epoch: Training Accuracy 0.954, Validation Accuracy 0.955

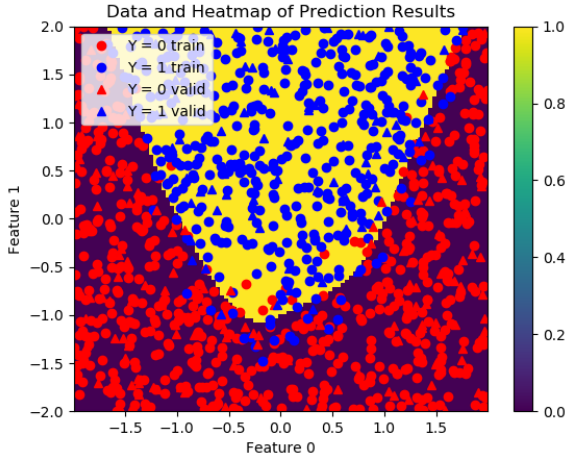


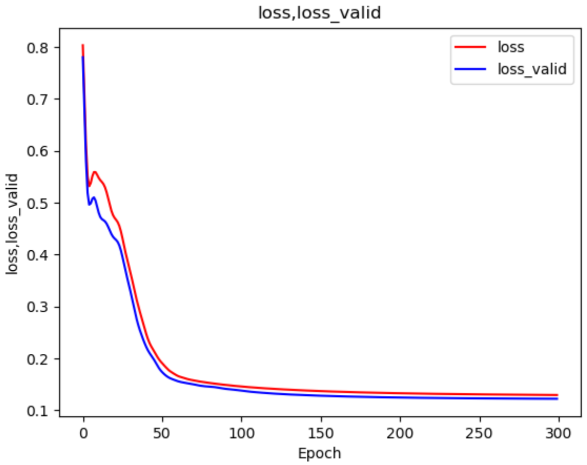


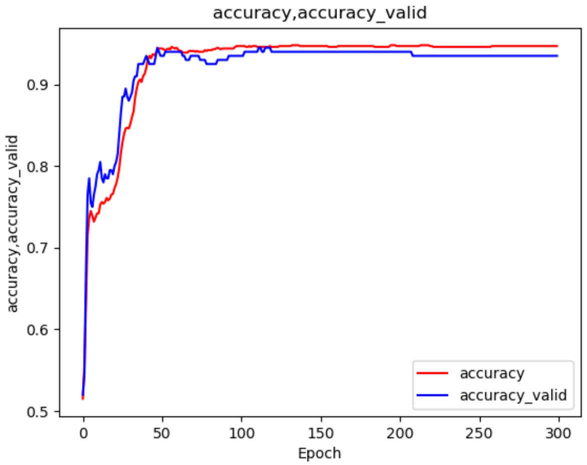


Overfitting Remedy – Simpler Neural Network

- Example: Binary Classification – Quadratic with Noise
  - 1000 training data points chosen from uniform distribution in [-2,2]x[-2,2]
  - 200 validation data points chosen from uniform distribution in [-2,2]x[-2,2]
- Machine Learning approach:
  - Neural Network with 2 layers (4,1 units)
  - tanh activation for all layers except sigmoid in last
  - Momentum Optimization with α=0.3 and β=0.9
  - Train with 300 epochs and batch_size = 1000
- Results
  - Heatmap shows smoother boundary between 0 and 1 regions
  - Loss for validation dataset slightly greater than that for training dataset
  - At final epoch: Training Accuracy 0.947, Validation Accuracy 0.935

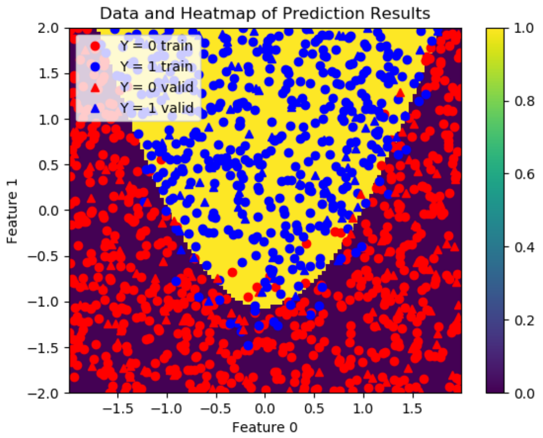


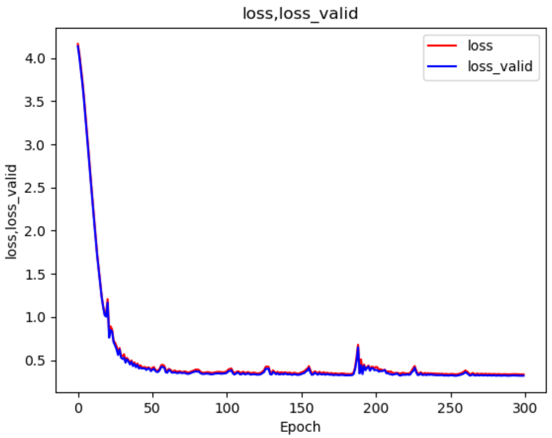


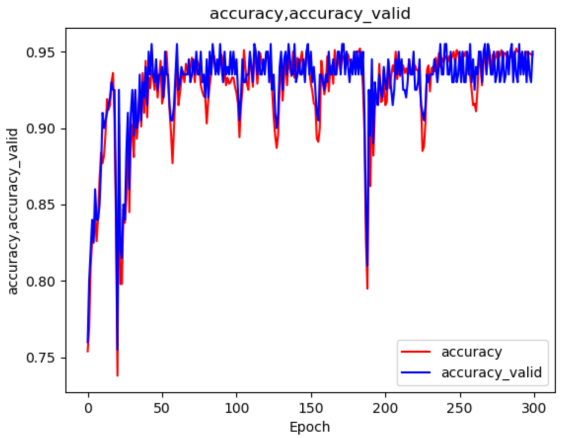


Overfitting Remedy – L2 Regularization

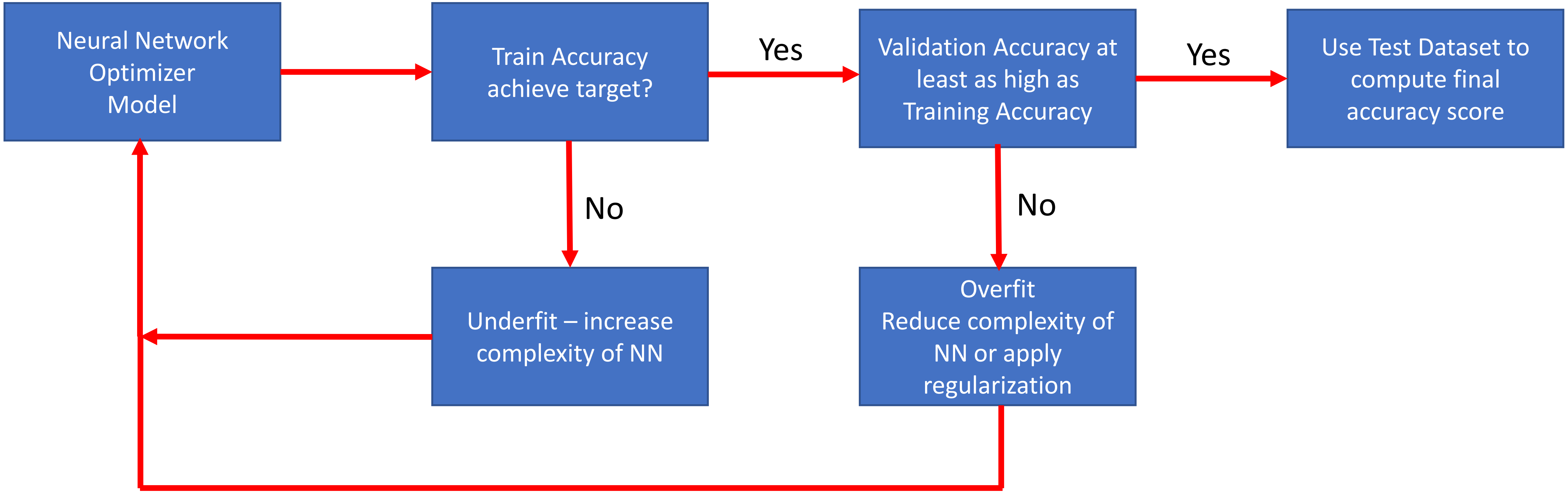
- Example: Binary Classification – Quadratic with Noise
  - 1000 training data points chosen from uniform distribution in [-2,2]x[-2,2]
  - 200 validation data points chosen from uniform distribution in [-2,2]x[-2,2]
- Machine Learning approach:
  - Neural Network with 5 layers (15,11,8,4,1 units)
  - tanh activation for all layers except sigmoid in last
  - Momentum Optimization with α=0.3 and β=0.9
  - Train with 300 epochs and batch_size = 1000 (number training data points)
  - Use L2 regularization with $\lambda^{[k]}$=0.02 for all layers k=1,2,3,4,5
- Results
  - Heatmap shows smoother boundary between 0 and 1 regions
  - Loss for validation dataset slightly greater than that for training dataset
  - At final epoch: Training Accuracy 0.948, Validation Accuracy 0.95 (validation accuracy oscillates between 0.93 and 0.95)







Machine Learning Approach – Flowchart


Neural Network Optimizer Model
Train Accuracy achieve target?
Yes
Validation Accuracy at least as high as Training Accuracy
Yes
Use Test Dataset to compute final accuracy score
No
Underfit – increase complexity of NN
No
Overfit Reduce complexity of NN or apply regularization