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

Chapter 4.1 Mini-Batch Gradient Descent

4.1 Mini-Batch Gradient Descent

Goal of this Section:

- Present algorithm for Mini-Batch Gradient Descent

Batch versus Stochastic Gradient Descent

- Training Algorithm (with Gradient Descent) presented earlier is often referred to as “Batch” Gradient Descent
- Called Batch because gradient calculation/update to parameters is based on all samples
- Alternative: Stochastic Gradient Descent - compute gradient/update using one training data sample at a time

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, …
   - 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 $\nabla_{W^{[k]}} L$, $\nabla_{b^{[k]}} L$
     - Update parameters: $W^{[k]} \leftarrow W^{[k]} - \alpha \nabla_{W^{[k]}} L$, $b^{[k]} \leftarrow b^{[k]} - \alpha \nabla_{W^{[k]}} L$
   - Forward Propagate using X: compute $A^{[k]}$ for k=1,…,N
   - Compute loss using $A^{[N]}$

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

Stochastic Gradient Descent - Example

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

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

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

- 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 α=0.1
- Epoch 1 – Sample Index = 0
- Forward Propagation:

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

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

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 $\nabla_b L_{sample=0}$)

- 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:

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

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

- 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 1 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, …

Batch vs Stochastic Gradient Descent

Batch:

- Single update to parameters per epoch using all training data samples
- Advantages:
  - Computationally efficient
  - Can lead to more stable convergence
- Disadvantages
  - Stability may lead to convergence to local minimum
  - May need to keep all data in memory – could lead to memory issues and performance degradation for large datasets
  - If new data appears, need to compute update based on all data, can't just simply use the new data

Stochastic:

- Update parameters m times per epoch (m is number of training data samples)
- Advantages:
  - More frequent updates may lead to faster convergence
  - Noisy update process may lead algorithm to avoid local minimums
- Disadvantages
  - Typically more computationally expensive than batch optimization
  - Noisy learning process may lead to convergence issues

Mini-Batch Gradient Descent

- Mini-Batch Optimization is compromise between Stochastic and Batch Optimization
- Split training data into batches (example 50 samples per batch)
- At each epoch, compute update to parameters one batch at a time, cycling through all batches

Mini-Batch Creation Algorithm

Assume m training samples

Assume mini-batch size of b

Let n = number of batches

- n = m/b if b goes into m evenly
- n = rounddown(m/b) + 1 if b does not go into m evenly

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

May want to randomly permute samples before performing splitting

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 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

1. Input training data: feature matrix X and values Y
2. Choose batch size b/generate mini-batches $(X_{batch=j}, Y_{batch=j})$, j=1,…,n
3. Make initial guess for parameters: $W^{[k]}$ and $b^{[k]}$ for k=1,…,N
4. Choose learning rate $\alpha > 0$
5. Loop for epoch i = 1, 2, …
   - Loop over mini-batches j=0,…,n-1
     - Forward Propagate using $X_{batch=j}$: compute $A^{[k]}_{batch=j}$ 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: $W^{[k]} \leftarrow W^{[k]} - \alpha \nabla_{W^{[k]}} L_{batch=j}$, $b^{[k]} \leftarrow b^{[k]} - \alpha \nabla_{b^{[k]}} L_{batch=j}$
   - Forward Propagate using X: compute $A^{[k]}$ for k=1,…,N
   - Compute loss using $A^{[N]}$

Loop for fixed number of iterations (or if Loss reduced sufficiently)

To simplify notation, remove subscript epoch=

Training with Mini-Batch Gradient Descent - Example

See file IntroML/Examples/Chapter4/Chapter4.1_MiniBatchGradientDescent.py

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

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

$$X_{batch=1} = \begin{bmatrix} 1 & -1 \\ -2 & 2 \end{bmatrix} \quad Y_{batch=1} = [1 \quad 1]$$

- Assume that initial parameter values are:

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

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

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

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

Training with Mini-Batch Gradient Descent - Example

- Back Propagation:

$$\nabla_A L_{batch=0} = -\frac{1}{3}\left(\frac{Y_{batch=0}}{A_{batch=0}} - \frac{1-Y_{batch=0}}{1-A_{batch=0}}\right) = -\frac{1}{3}\left[-\frac{1}{1-0.5250} \quad \frac{1}{0.4750} \quad -\frac{1}{1-0.4502}\right]$$

$$= [0.7017 \quad -0.7017 \quad 0.6062]$$

$$\frac{\partial A_j}{\partial Z_j} = A_j - A_j^2 \quad \left[\frac{\partial A_0}{\partial Z_0} \quad \frac{\partial A_1}{\partial Z_1} \quad \frac{\partial A_2}{\partial Z_2}\right] = [0.2494 \quad 0.2494 \quad 0.2475]$$

$$\nabla_Z L_{batch=0} = \nabla_A L_{batch=0} * \left[\frac{\partial A_0}{\partial Z_0} \quad \frac{\partial A_1}{\partial Z_1} \quad \frac{\partial A_2}{\partial Z_2}\right] = [0.7017 \quad -0.7017 \quad 0.6062] *$$

$$[0.2494 \quad 0.2494 \quad 0.2475] = [0.1750 \quad -0.1750 \quad 0.1501]$$

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

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

- Update the parameters:

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

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

$$A_{batch=1} = f(Z_{batch=1}) = \left[\frac{1}{1+e^{0.0926}} \quad \frac{1}{1+e^{-0.4626}}\right] = [0.4769 \quad 0.6136]$$

- Back Propagation

$$\nabla_A L_{batch=1} = -\frac{1}{2}\left(\frac{Y_{batch=2}}{A_{batch=2}} - \frac{1-Y_{batch=2}}{1-A_{batch=2}}\right) = -\frac{1}{2}\left[\frac{1}{0.4769} \quad \frac{1}{0.6136}\right] = [-1.0485 \quad -0.8148]$$

$$\frac{\partial A_j}{\partial Z_j} = A_j - A_j^2 \qquad \left[\frac{\partial A_0}{\partial Z_0} \quad \frac{\partial A_1}{\partial Z_1}\right] = [0.2495 \quad 0.2371]$$

$$\nabla_Z L_{batch=1} = [-1.0485 \quad -0.8148] * [0.2495 \quad 0.2371] = [-0.2616 \quad -0.1932]$$

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

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

- Update the parameters:

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

Derivative Testing – Jupyter Notebook Demo

- Open files
  - IntroML/Examples/Chapter4/Chapter4.1_StochasticGradientDescent.ipynb
  - IntroML/Examples/Chapter4/Chapter4.1_MiniBatchGradientDescent.ipynb

Chapter 4.2 Momentum, RmsProp, Adam Optimizers

4.2 Momentum, RmsProp, and Adam Optimization

Goal of this Section:

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

Optimization

- Gradient Descent is rudimentary optimization algorithm
- Not guaranteed to converge to global minimum – 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 issues and improve convergence
- Mini-batch Gradient Descent approach can also be applied to Momentum, RmsProp, and Adam optimization methods

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

1. Make initial guess $W_{epoch=0}$ for parameters
2. Loop epoch i = 1, 2, 3, …

- Compute gradient vector $\nabla_W L_{epoch=i-1}$ at $W_{epoch=i-1}$
- Compute $Update_{epoch=i}$ ← Depends on approach
- Compute new epoch using formula: $W_{epoch=i}$ $W_{epoch=i-1}$ + $Update_{epoch=i}$
- Compute $L(W_{epoch=i})$

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

Momentum

- Momentum algorithm has a 2-step update formula:

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

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

where $0 \leq \beta < 1$

- Motivation for momentum is to avoid getting stuck in a local minimum
- Can show:

$v_{epoch=1} = \nabla_W L_{epoch=0}$

$v_{epoch=2} = \nabla_W L_{epoch=1} + \beta \nabla_W L_{epoch=0}$

$v_{epoch=i} = \nabla_W L_{epoch=i-1} + \beta \nabla_W L_{epoch=i-2} + \cdots + \beta^{i-1} \nabla_W L_{epoch=0}$

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 and initial guess

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

- Iteration 1:

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

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

$W_{epoch=1} = W_{cpoch=0} - \alpha v_{epoch=1} = [2 \quad 2] - 0.1 * [8 \quad 4] = [1.2 \quad 1.6]$

$L(W_{epoch=1}) = 2 * 1.2^2 + 1.6^2 = 5.44$

- Iteration 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 number of iterations increase
- 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

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

Iteration 1:

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

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

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

$= [1.6838 \quad 1.6838]$

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

Component-wise square root and division

RmsProp - Example

Iteration 2:

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

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

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

$= [1.4739 \quad 1.4739]$

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

Component-wise square root and division

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
- m represents moving average of gradient
- "Effective" learning rate decreases as number of iterations increase
- ε included to avoid division by 0
- 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 initial guess

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

Iteration 1:

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

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

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

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

$= [1.6838 \quad 1.6838]$

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

Component-wise
square root and division

Adam - Example

Iteration 2:

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

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

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

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

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

Component-wise
square root and division

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
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 $m_{epoch=0} = 0$ $v_{guess=i} = \beta_2 v_{epoch=i-1} + (1-\beta_2) \nabla_W L^2_{epoch=i-1}$ start with $v_{epoch=0} = 0$ $Update_{epoch=i} = -\alpha \frac{m_{epoch=i}}{\sqrt{v_{epoch=i}} + \varepsilon}$

Derivative Testing – Jupyter Notebook Demo

- Open files
  - IntroML/Examples/Chapter4/Chapter4.2_Momentum.ipynb
  - IntroML/Examples/Chapter4/Chapter4.2_RmsProp.ipynb
  - IntroML/Examples/Chapter4/Chapter4.2_Adam.ipynb

Chapter 4.3 Code Walkthrough Version 3.1

4.3 Coding Walkthrough: Version 3.1

Goal of this Section:

- Walkthrough addition of mini-batch Gradient Descent
- 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 train method perform mini-batch optimization
Optimizer	Add derived classes for Momentum, RmsProp, and Adam optimizers
Optimizer	Update constructor to accommodate Momentum, RmsProp, and Adam

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 (X_batch,Y_batch) 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
RmsProp: 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 $W^{[2]}$

Chapter 4.4 Validation and Accuracy

4.4 Accuracy and Validation

Goal of this Section:

- Review of validation and testing approaches
- Review metrics for measuring accuracy of neural networks

Effectiveness of Neural Network Models

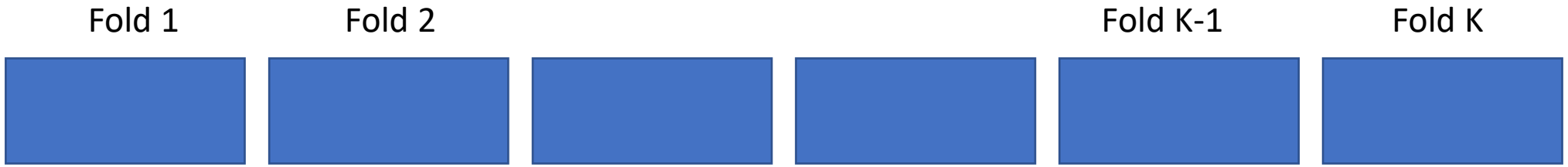
- In code examples, we have used accuracy of prediction for the training set as a measure of effectiveness of the Neural Network
- This provides some information but is not a reliable measure of model effectiveness
  - In trivial case, can create a model that is simply a look-up table based on training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

Train, Validation, and Test Data Sets

Dataset	Description
Train	Data used to fit the model (Feature Matrix X and Label Vector Y in notation of this course)
Validation	Separate dataset from training dataset used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration. (Hyperparameters include number of layers, number of units, optimization parameters, $\alpha, \beta, \beta_1, \beta_2, \varepsilon$)
Test	The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.
	Source: Towards Data Science https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7

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 batches 1,2,…,K-1 and validation with batch K
  - Repeat: train with batches 2,3,…,K and validation with batch K
  - Repeat will 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
- In case of binary or multiclass classification, ensure that validation and train data sets have same distribution of outputs
  - 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

Accuracy Calculation

- Accuracy by itself is not always useful 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 model, 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?

Precision, Recall and F1 Score

- Accuracy is fraction of positives and negatives predicted correctly

$$Accuracy = \frac{\#TruePositive + \#TrueNegative}{\#Total}$$

- Precision is fraction of predicted positives that are correct

$$Precision = \frac{\#TruePositive}{\#TruePositive + \#FalsePositive}$$

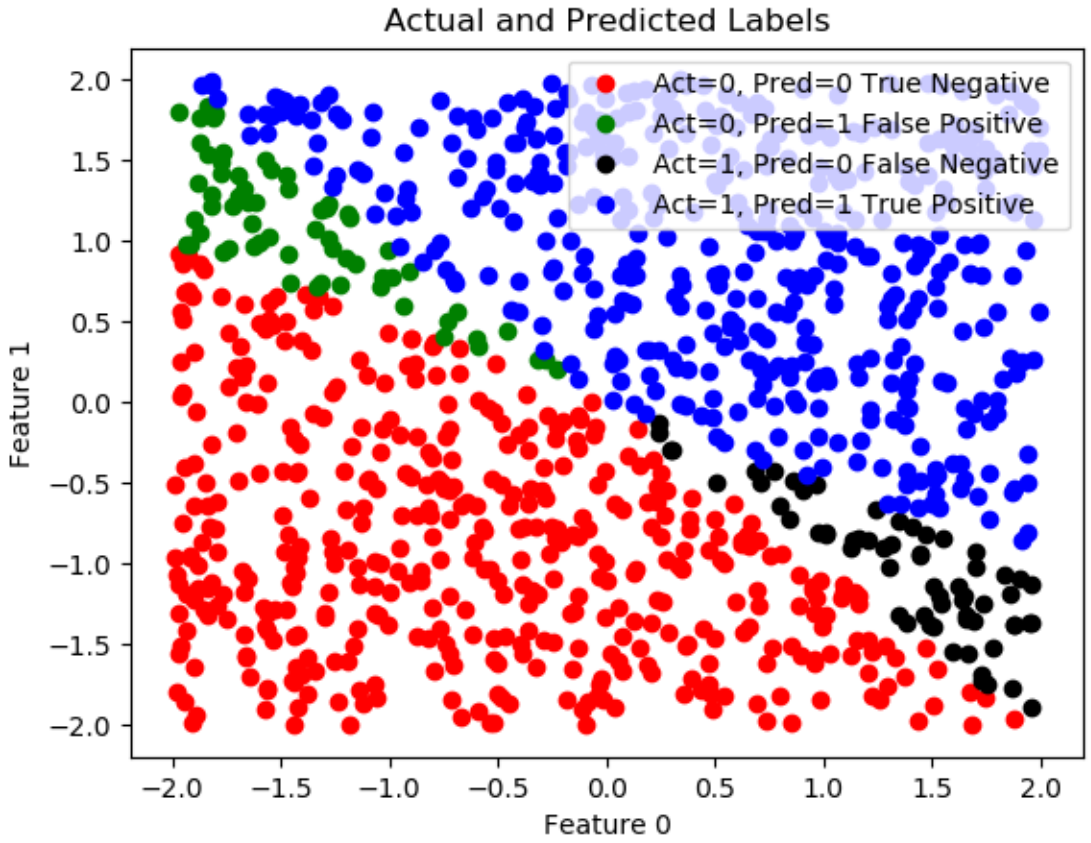
- Recall is fraction of actual positives predicted correctly

$$Recall = \frac{\#TruePositive}{\#TruePositive + \#FalseNegative}$$

- F1 score combines into a single figure

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

- For each of these measures, 1 is optimal score

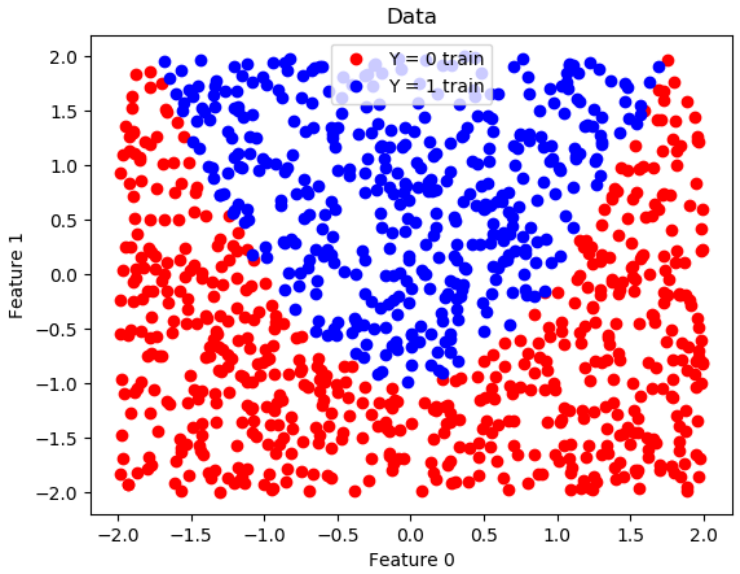


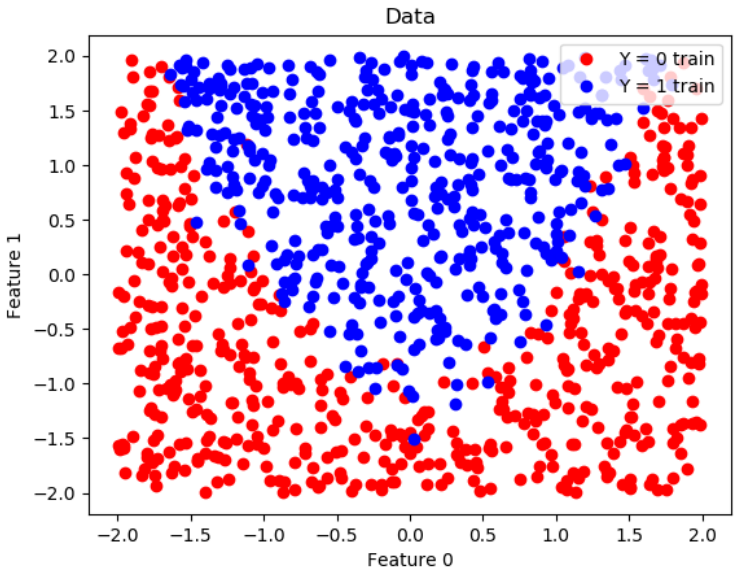
Target Accuracy

- What accuracy or (F1 score) should we expect from a machine learning system? Is it always 100%?
- Target accuracy should be what is expected from a human (or expert)
  - Example: for classification of cats and dogs based on photos, expect human to be able to achieve 100% accuracy.
  - Example: for medical image classification (x-rays, scans), target accuracy is that achieved by expert or specialist in the field
- 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
  - Expect target accuracy to be less than 100%





Confusion Matrix

- For classification (binary or multiclass), Confusion Matrix is a matrix/table listing counts of actual and predicted for each class
- Useful for determining patterns in mis-classification
- To produce
  - Perform training
  - For dataset where labels are known, use model to predict classes
  - For each class, tally number of actual and predicted
- Example: 3 classes
  - First column indicates Actual = 0 for 49 cases. Of these, 45 were predicted class 0, 3 were predicted class 1, and 1 was predicted class 2

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

Algorithm for Producing Confusion Matrix

- 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

Chapter 4.5 Code Walkthrough Version 3.2

4.5 Coding Walkthrough: Version 3.2

Goal of this Section:

- Add calculation of 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 accuracy for validation data set for each epoch in fit method
metrics	Add function to calculate precision, recall and F1 score
metrics	Add function to print confusion matrix

NeuralNetwork_Base Class – Methods

Method/Function	Input	Description
fit	X (numpy array) Y (numpy array) epochs (integer) **kwargs batch_size (integer) validation_data (tuple containing feature matrix and labels)	Update the existing train method to compute accuracy for validation dataset if provided Return: dictionary containing loss and accuracy histories for train and validation dataset (if provided)
f1store	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

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

Chapter 4.6 Regularization

4.6: Regularization

Goal of this Section:

- Discuss purpose of regularization
- Present approach for L2 Regularization

Purpose of Regularization

Regularization: technique for solving ill-posed problems/avoid overfitting

- Approaches for Regularization:
  - Add “information” to problem
    - “Ridge Regression” L1 regularization
    - “Lasso Regression” L2 regularization
  - Stop training algorithm early
  - Dropout (remove some nodes in layers randomly)

L1 and L2 Regularization

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

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

- L2 Regularization:

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

- 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
- Penalty term forces training algorithm to tend to smaller parameter values $W_{ij}^{[k]}$

L2 Regularization - Example

- Consider a case of 2 features and 3 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 that layer 1 has 2 units and that layer 2 has 1 unit
- Assume parameter matrices

$$W^{[1]} = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & -0.5 \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \qquad W^{[2]} = \begin{bmatrix} -1 & 1 \end{bmatrix} \qquad b^{[2]} = \begin{bmatrix} -0.1 \end{bmatrix}$$

- 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:

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

Impact of L2 Regularization on Back Propagation

- L2 Regularization has no impact on Forward Propagation
- L2 Regularization does affect Loss function and 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

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

- 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 4.7 Hyperparameter Searches

4.7 Hyperparameter Search

Goal of this Section:

- Describe hyperparameter searches

Hyperparameters

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 Routine	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 constants $\lambda^{[k]}$ for layers k=1,…,N

Hyperparameter Search

Machine Learning Process:

- Define training, validation, and final test datasets
- Create Neural Network Model with various hyperparameters
- Search for “Best” Hyperparameter combination
  - Perform training for each hyperparameter combination
  - Measure accuracy for training and validation datasets
- Choose best Hyperparameter combination
  - Determine final accuracy using test dataset

Chapter 4.8 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	Update these drivers to include input of lambda value for L2 regularization
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_propagation	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)
NeuralNetwork_Base Fit	X (numpy array) Y (numpy array) epochs (integer) **kwargs batchsize (integer) validation_data (tuple containing feature matrix and labels) verbose (boolean)	Add True/False input verbose to determine if epoch level results should be printed Return: dictionary containing loss and accuracy histories for train and validation dataset (if provided)

Hyperparameter Search

File/Component	Input	Description
write_csv		

Chapter 4.9 Underfitting and Overfitting

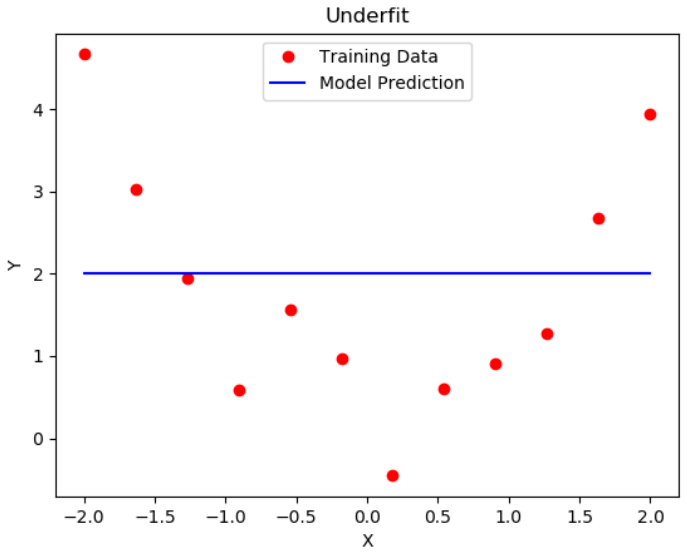
Underfitting and Overfitting

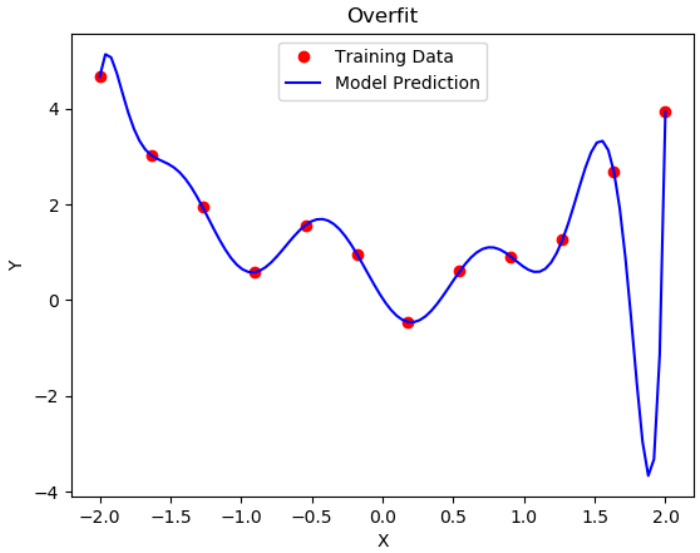
Goal of this Section:

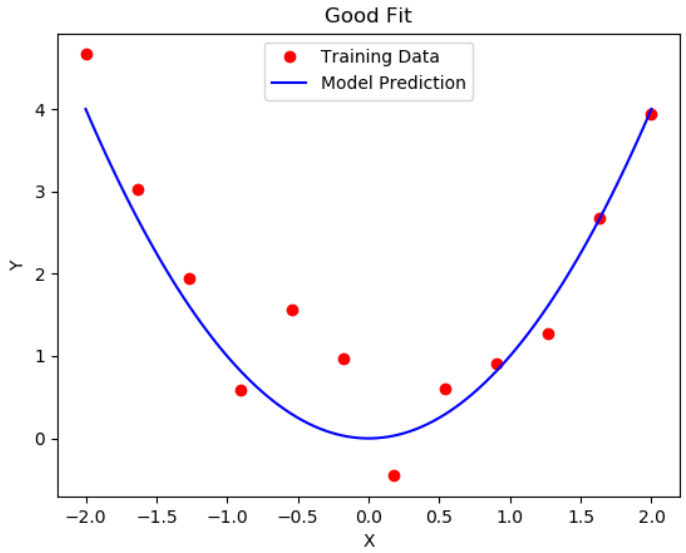
- Define and provide examples of Underfitting and Overfitting
- Identify symptoms and remedies for addressing Underfitting and Overfitting

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: 11^{th} degree polynomial used to fit 12 data points – fits training data exactly, but large discrepancy at end point for points not in training data set
- 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







Underfitting/Overfitting and Bias/Variance

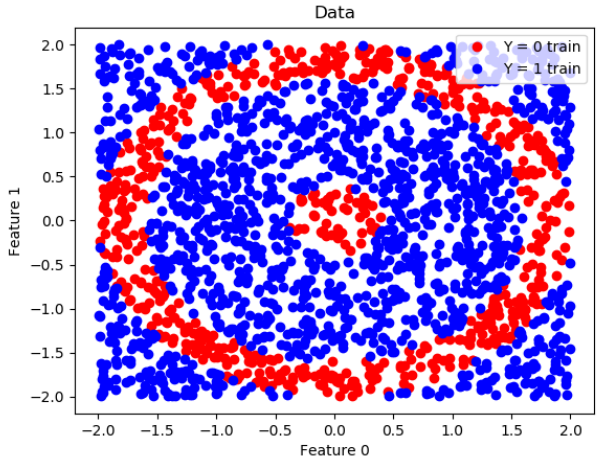
Terms Bias and Variance used for Underfitting and Overfitting

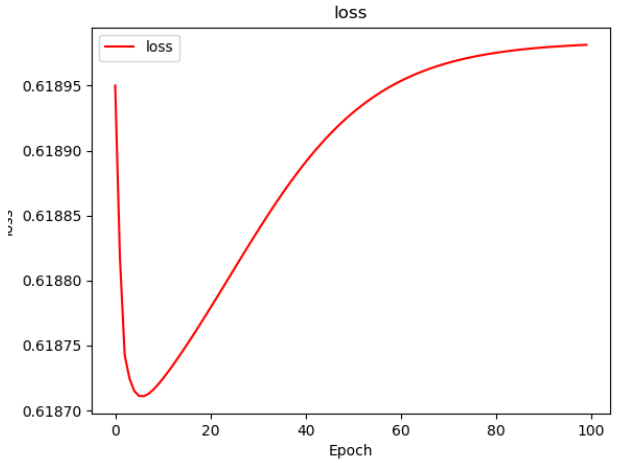
- Bias (Error) = Underfitting
  - Occurs from erroneous assumptions in learning algorithm. High bias can cause algorithm to miss relationship between input features and target outputs.
- Variance = Overfitting
  - Sensitivity to small fluctuations in training data. High variance can cause algorithm to capture random noise in training data as opposed to general trend.
- Source

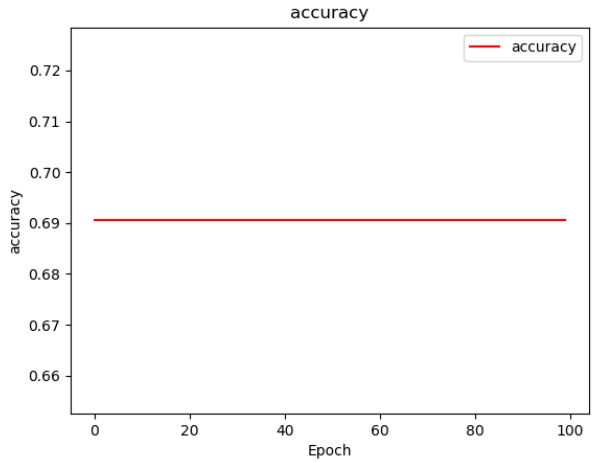
https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff

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 swells above 0 – roughly 0.62 in example below
  - Training Accuracy well below target accuracy – stays at 0.69 in example below

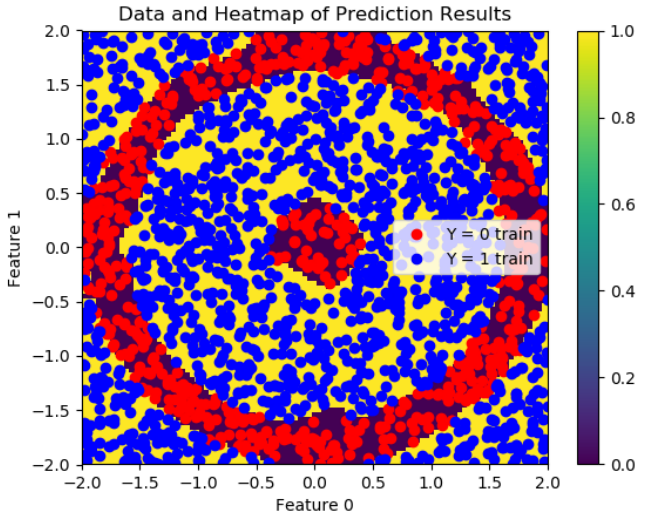


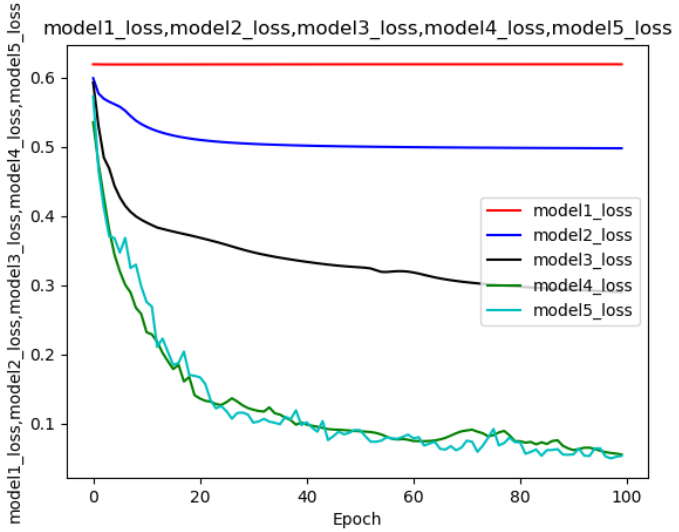


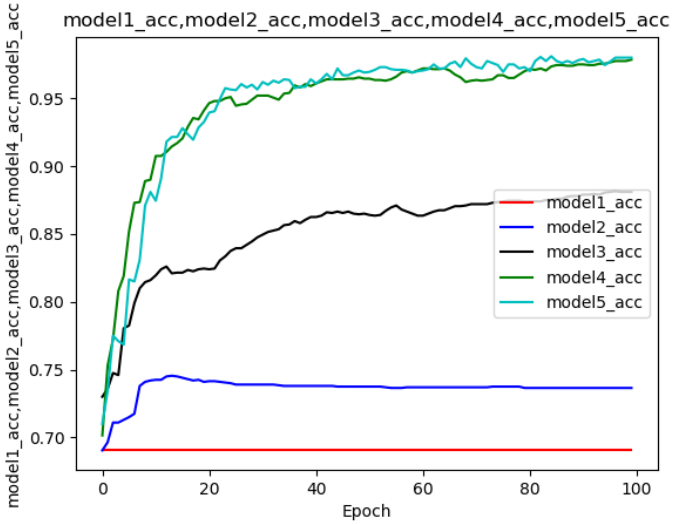


Underfitting - Remedy

- Remedy for underfitting: use neural network with more layers
- 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%

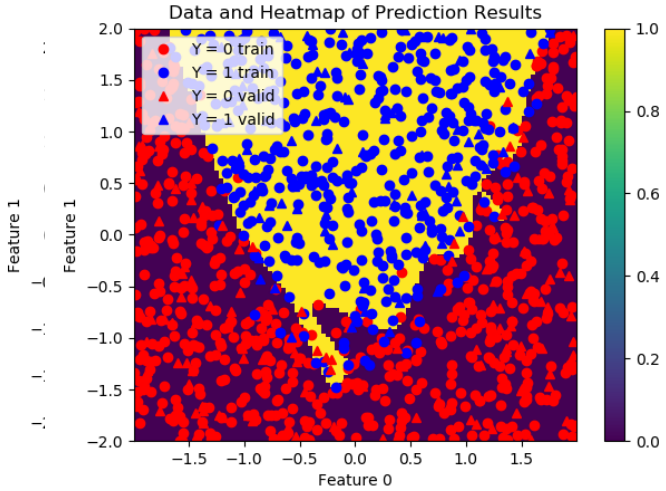


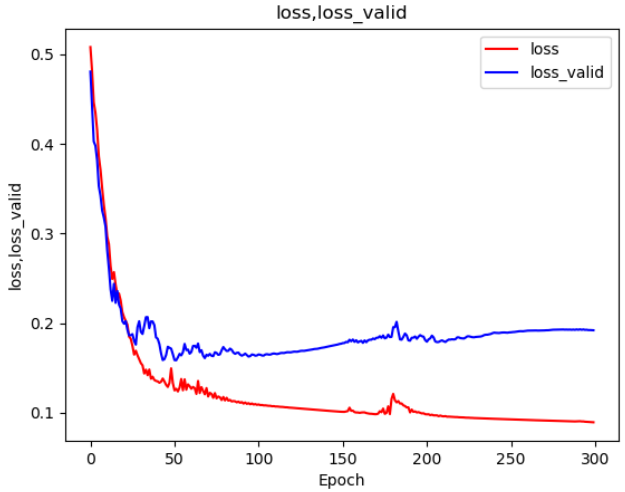


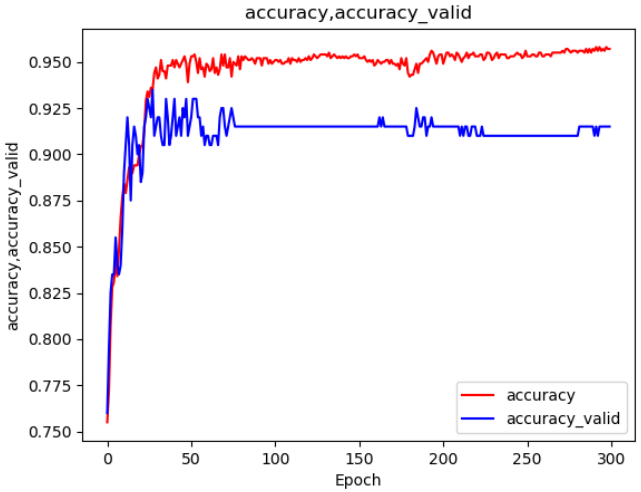


Overfitting - Example

- 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)







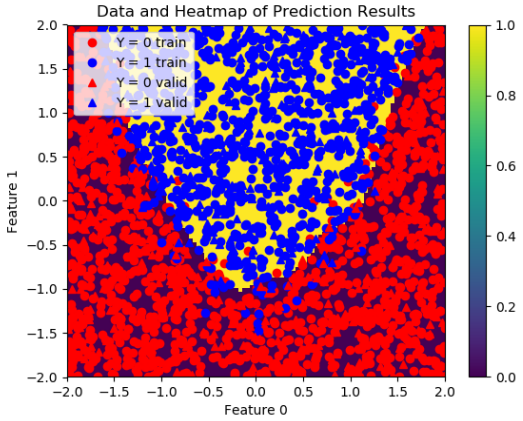
Overfitting - Remedies

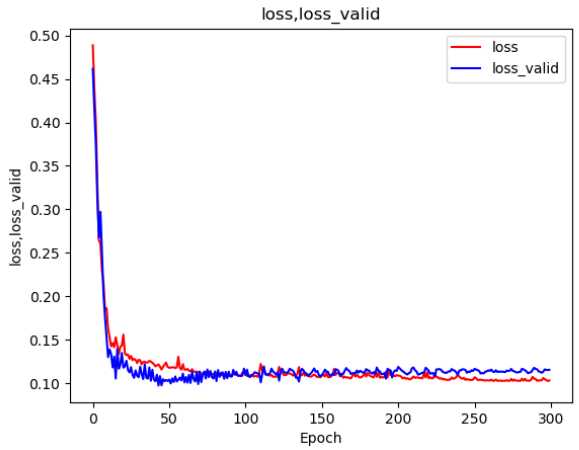
Overfitting Remedies

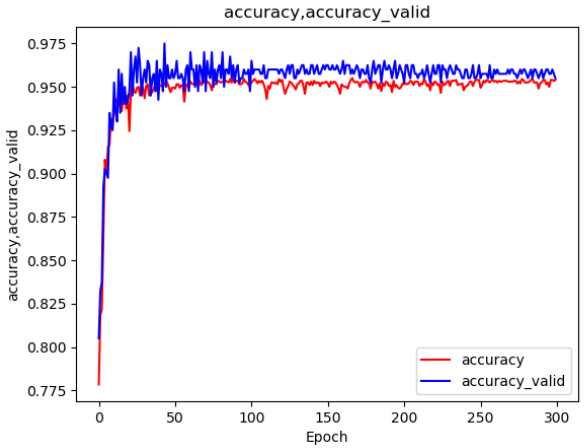
- Use more data points
  - This is not always possible for data in real world setting
- Use simpler function structure
  - In previous example, see “fingers” in model prediction to attempt to fit training points near boundary
  - Fewer layers in neural network should lead to simpler boundary between 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

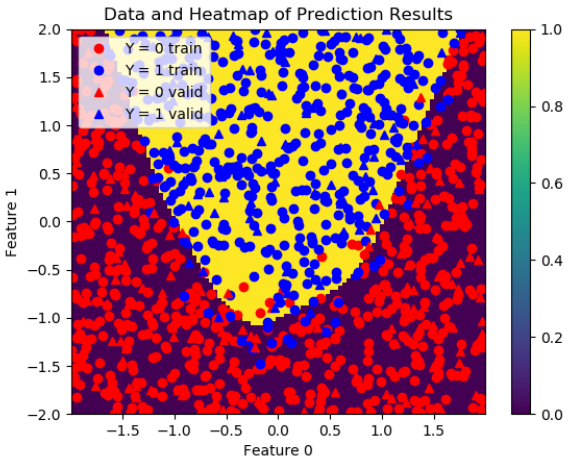


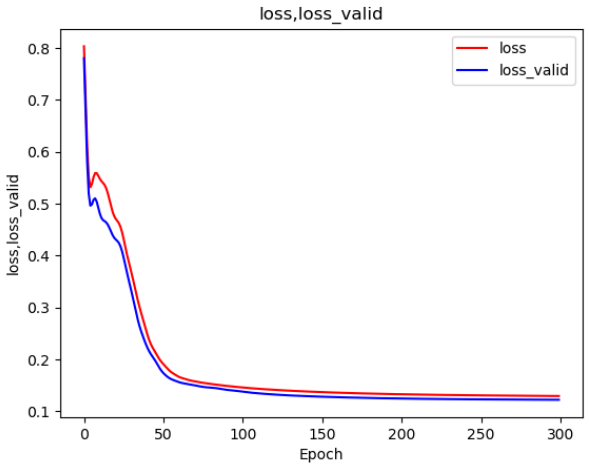


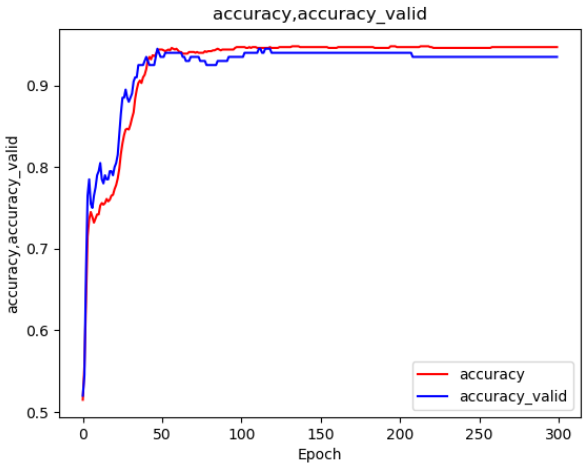


Overfitting Remedy – Simpler Neural Network

- 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

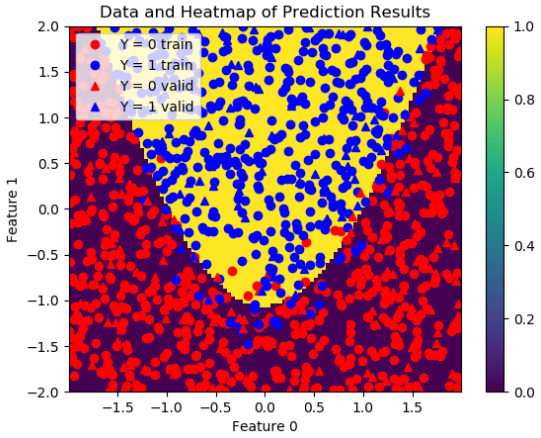


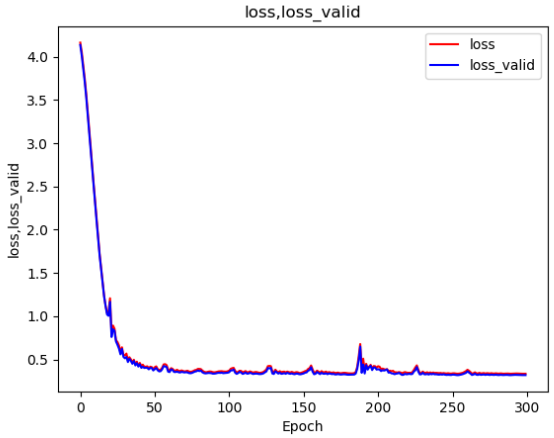


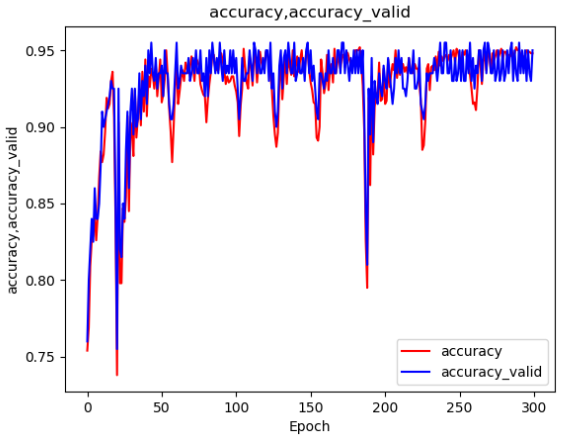


Overfitting Remedy – L2 Regularization

- 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 batchsize = 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