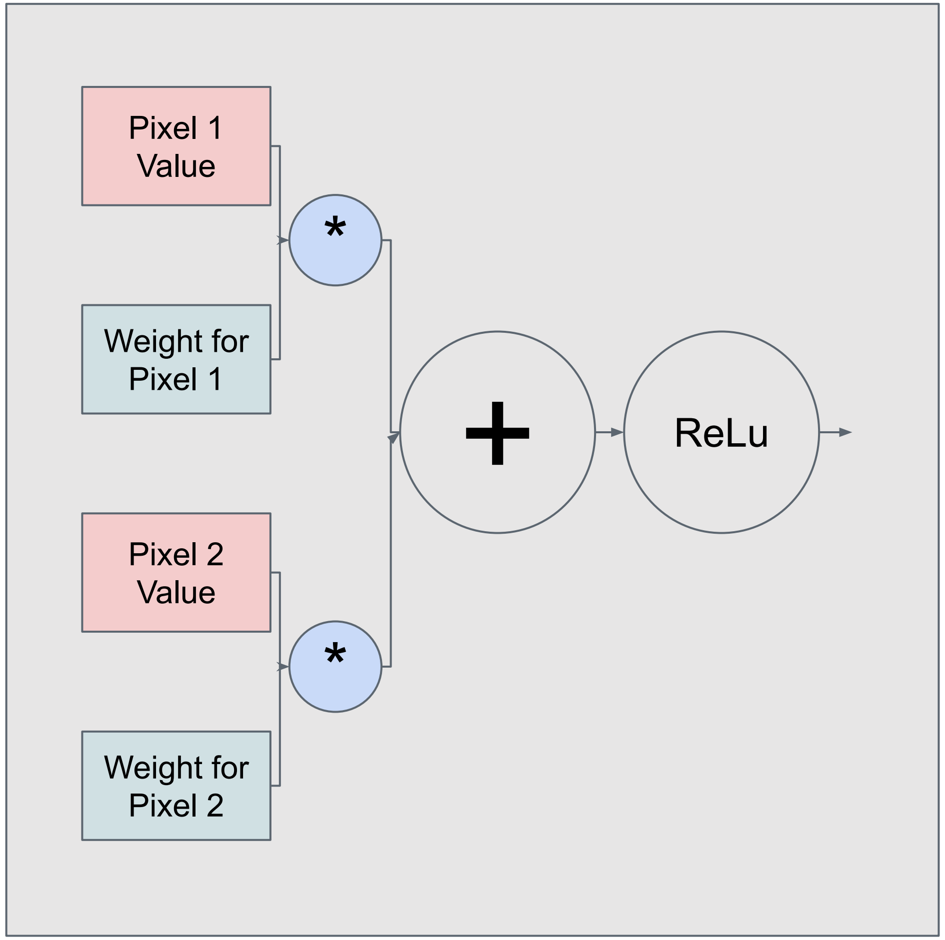
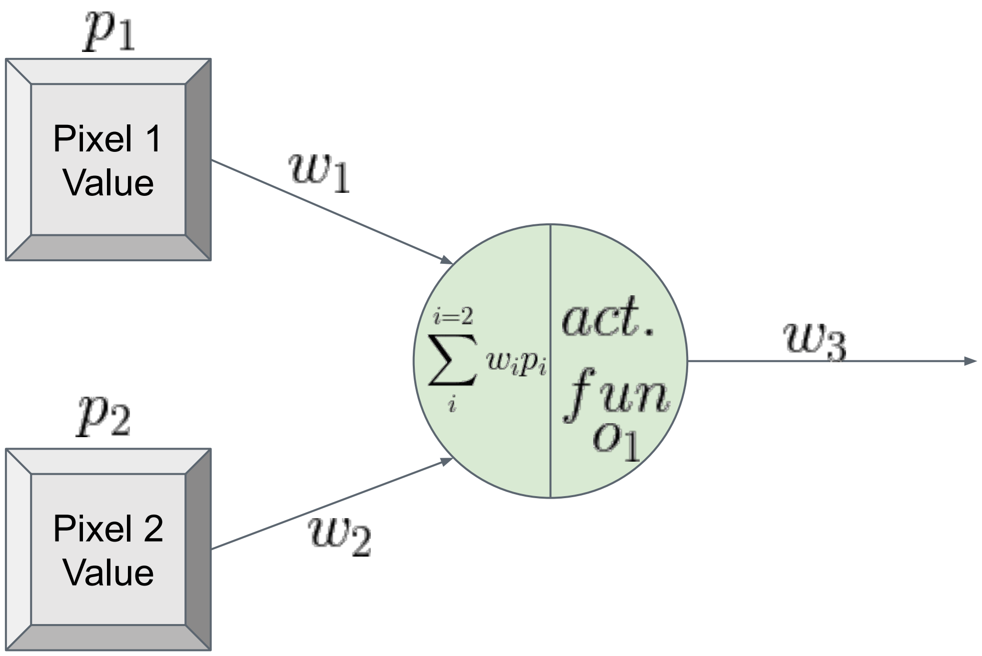
DATA 442: Neural Networks & Deep Learning

Dan Runfola – danr@wm.edu

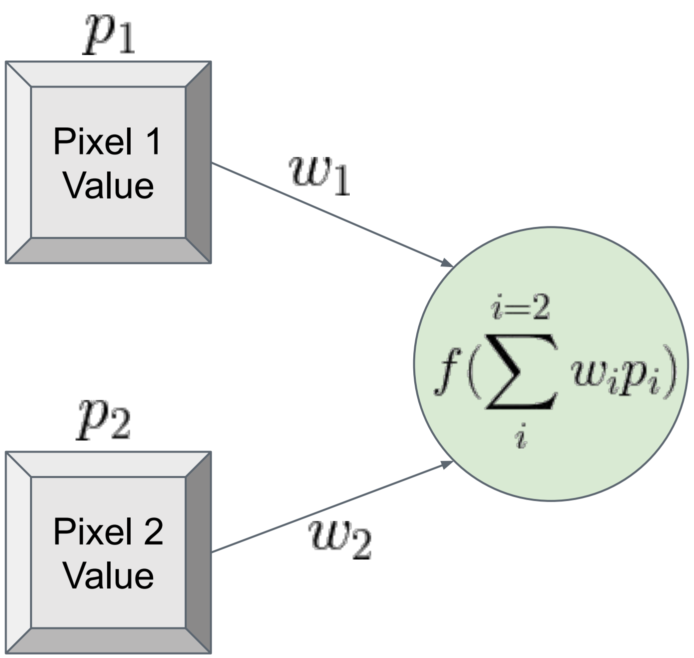
icss.wm.edu/data442/

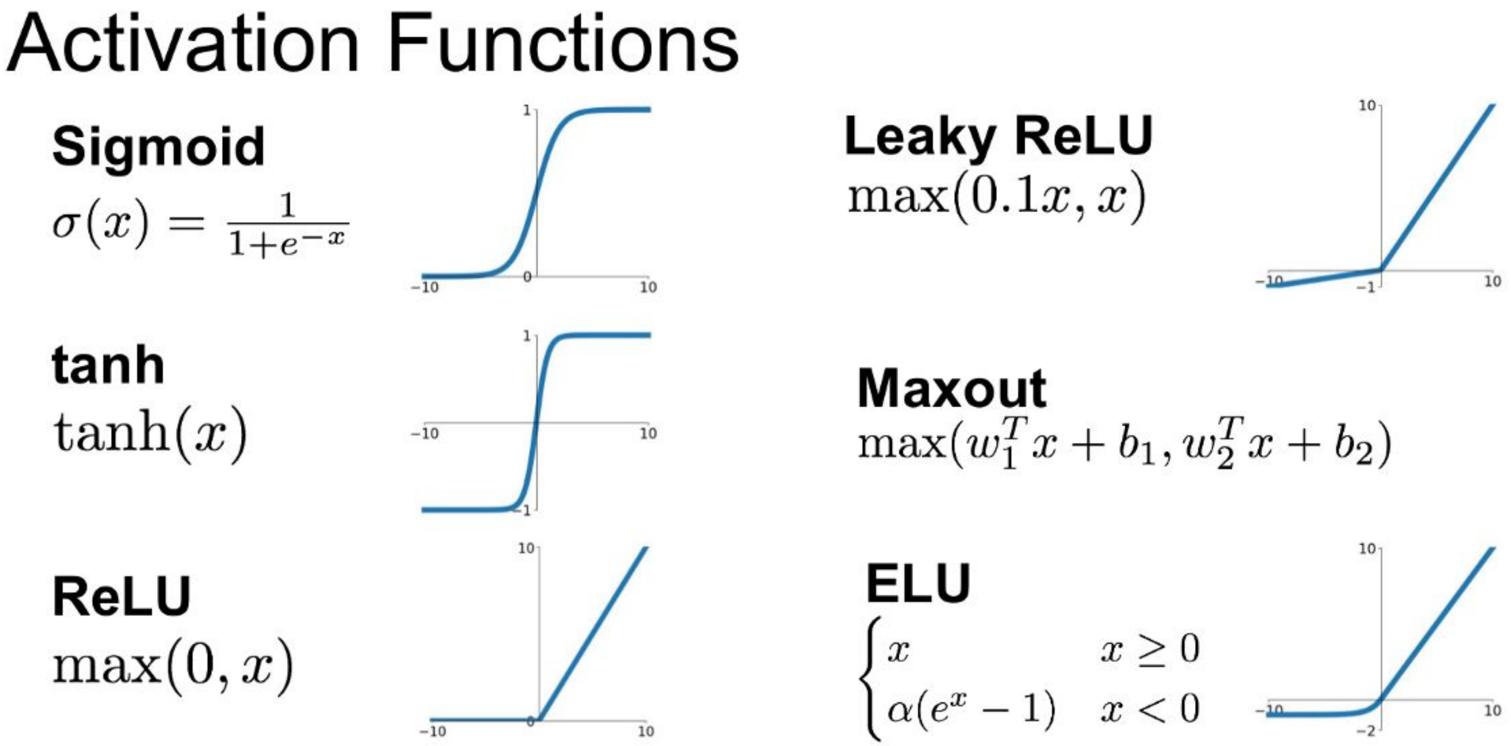
Network Architecture: Fundamentals



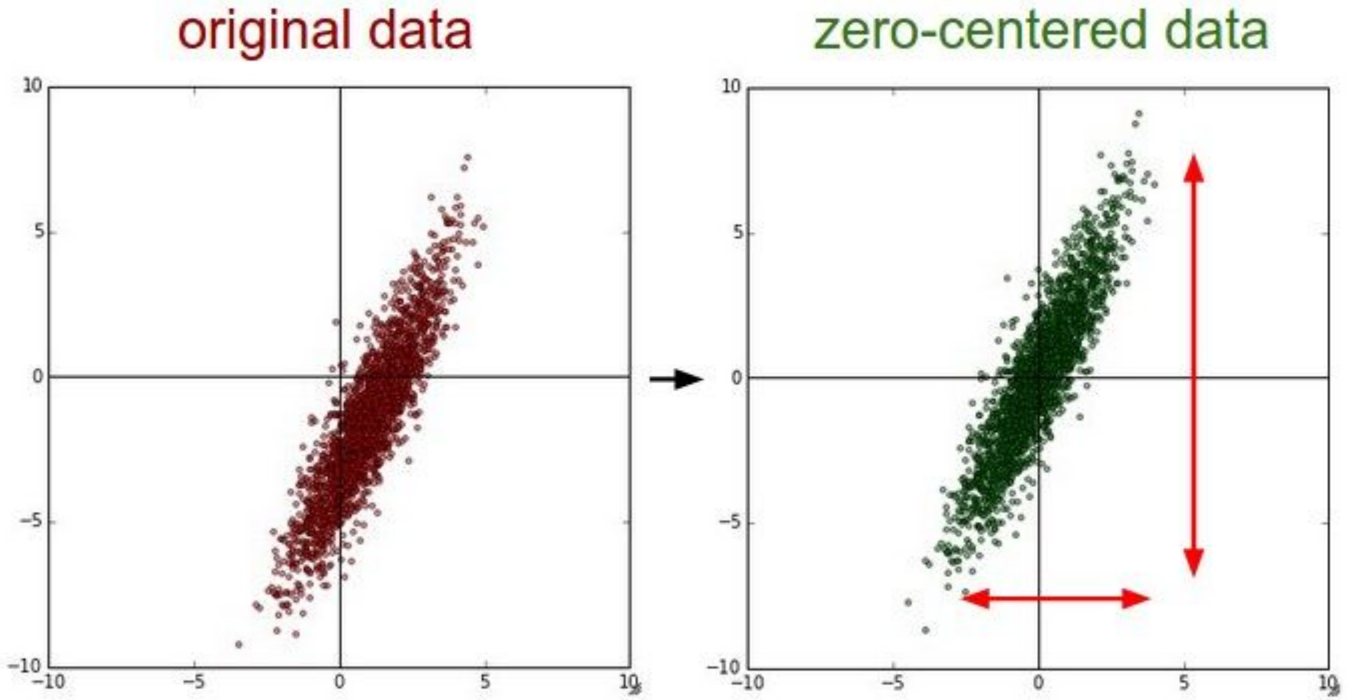


Network Architecture: Activation Function





Network Architecture: Data Preprocessing



Xavier Initialization

Original:

```
W = np.random.randn(3072, 10) * .0001
```

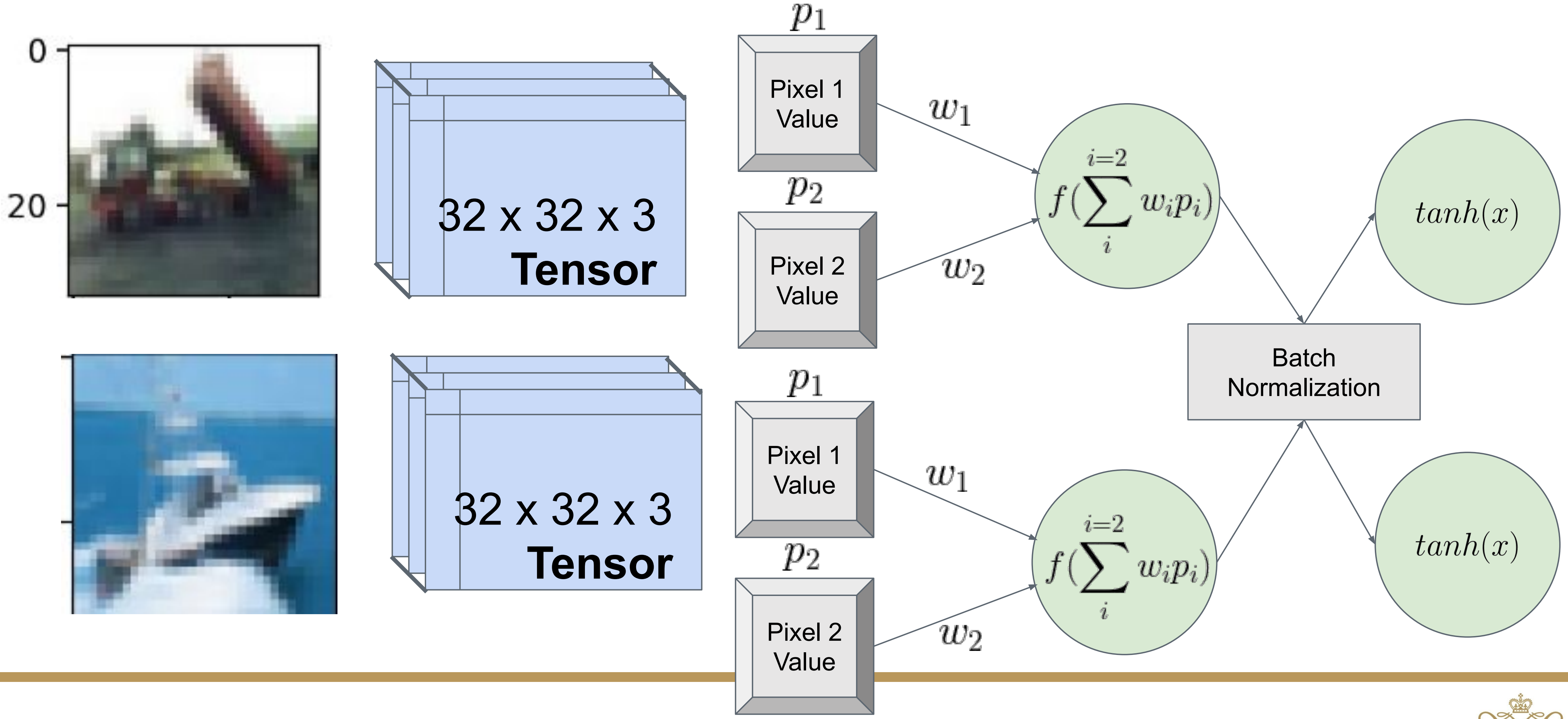
Xavier:

```
W = np.random.randn(3072, 10) / np.sqrt(3072)
```

He:

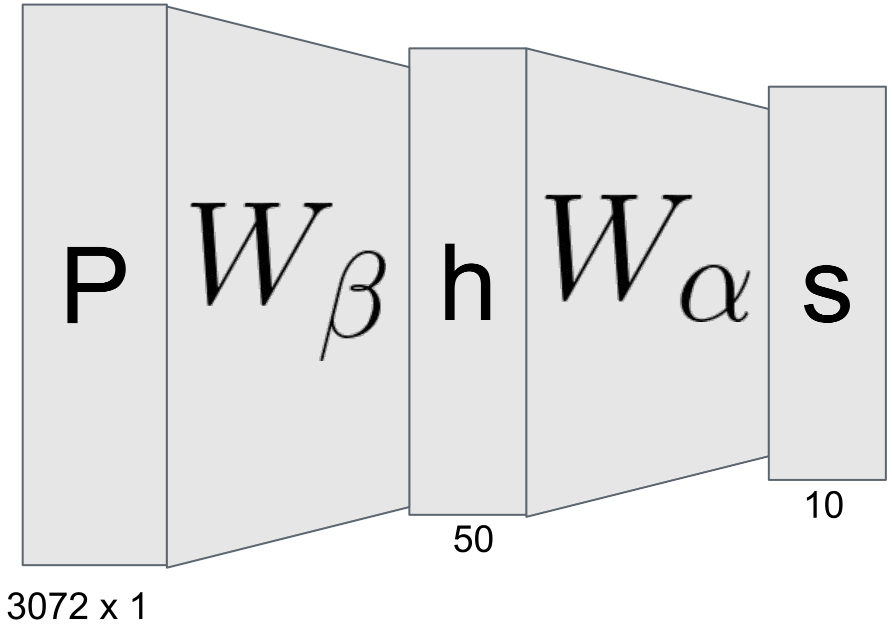
```
W = np.random.randn(3072, 10) / np.sqrt(3072 / 2)
```

Another Strategy: Batch Normalization



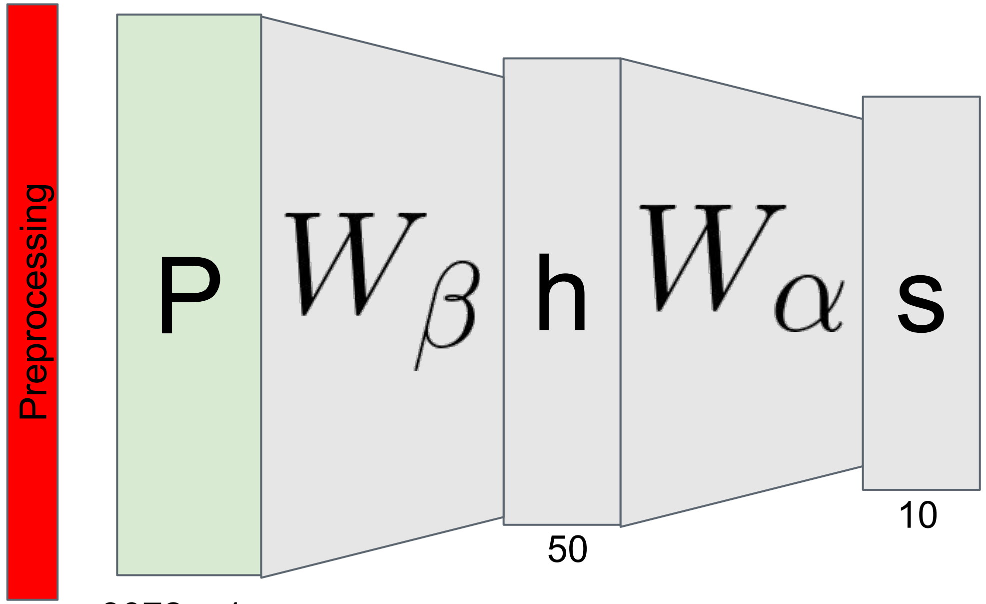
Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

Architecture:



Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

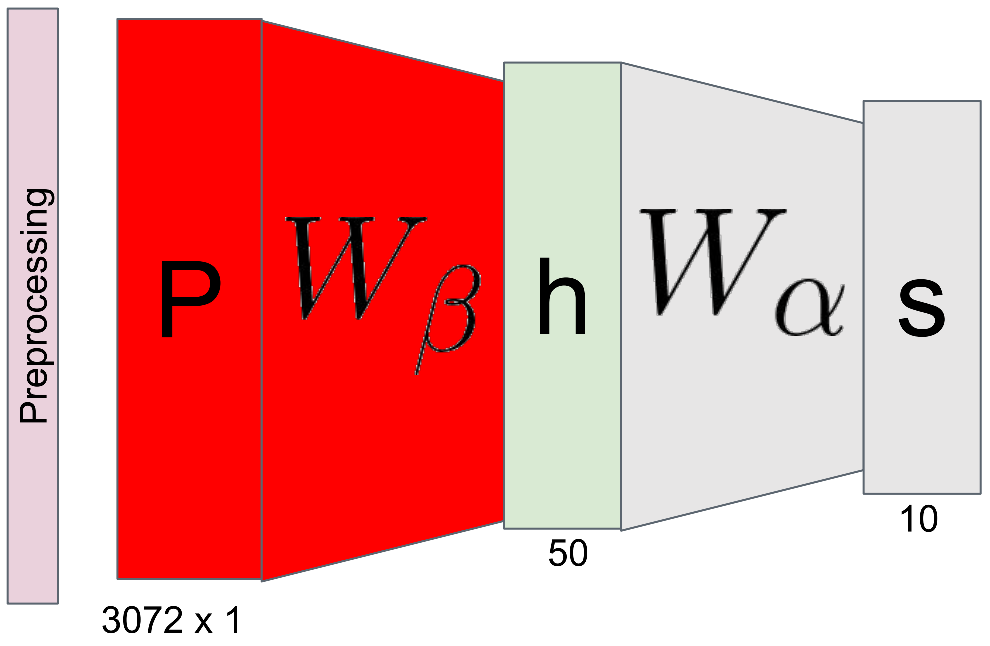
Architecture:




```
def preProcessing(train, test, arrayReshape=True, zeroShift=True, zeroShiftVis = True):
    if(zeroShift == True):
        #First, we're going to calculate the overall mean image across our training dataset.
        mean_image = np.average(train, axis=0)
        if(zeroShiftVis == True):
            plt.figure(figsize=(4,4))
            plt.imshow(mean_image.reshape((32,32,3)).astype('uint8'))
            plt.show()
        #And - we subtract!  That's all there is to this.
        train -= mean_image
        test -= mean_image

    if(arrayReshape == True):
        #Here, we're reshaping CIFAR-10 from 32x32 to a 1x3024 array.
        #We are moving this into preprocessing, as once we get to
        #convolutional nets, we won't want to do this anymore.
        train = np.reshape(train, (train.shape[0], -1))
        test = np.reshape(test, (test.shape[0], -1))

    return(train, test)
```

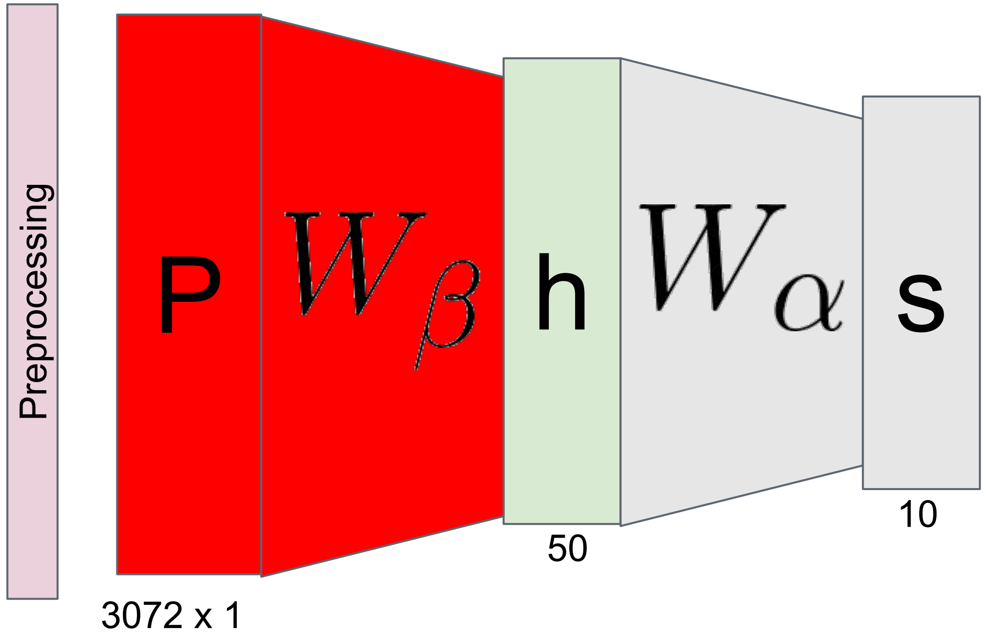
Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

<u>Architecture:</u>



Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

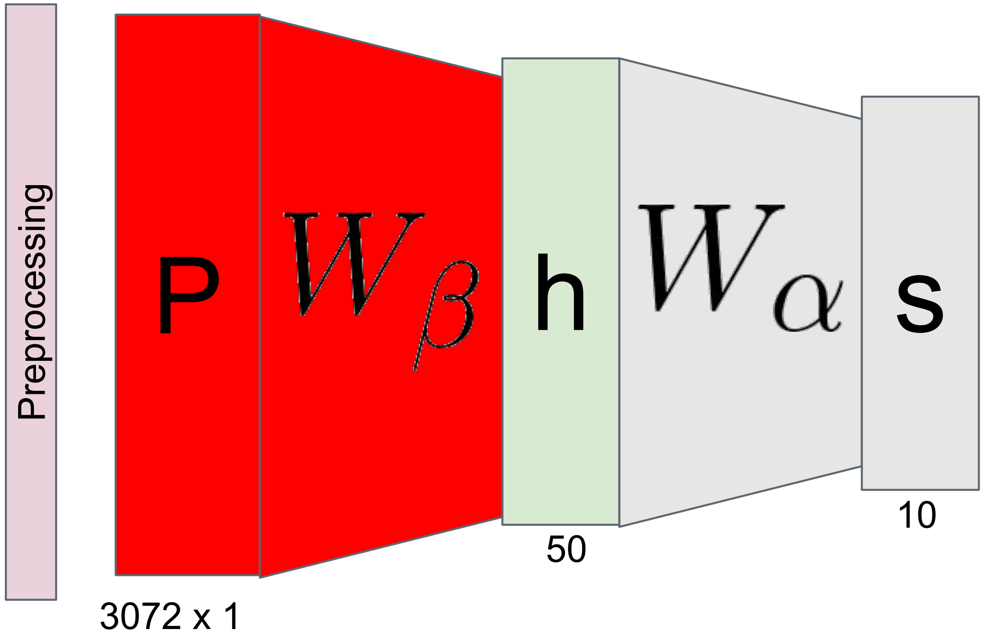
Architecture:




```
def affineForward(x, w, b):
    out = np.dot(x, w) + b
    cache = (x, w, b)
    return(out, cache)
```

Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

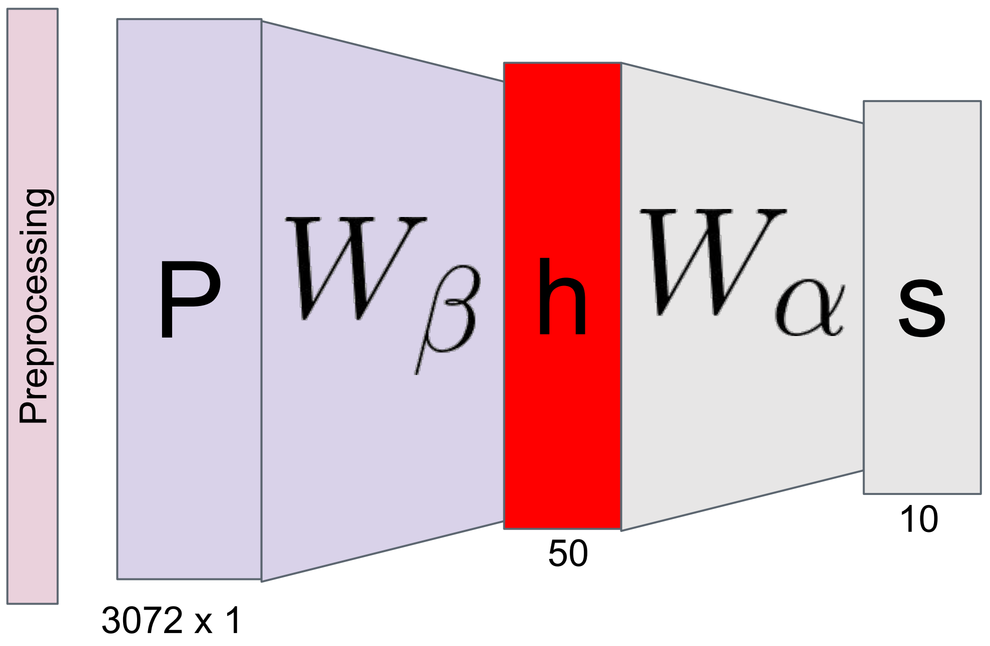
Architecture:




```
def affineForward(X, W, B):
    #Total number of observations:
    N = X.shape[0]

    #Number of dimensions - in this example, 3072 (i.e., each observation has 3072 values)
    D = np.prod(X.shape[1:])

    #Reshape our inputs to be (N,D), matching our expectation for the weights dot product.
    xReshape = np.reshape(X, (N, D))

    #Calculate the dot product:
    out = np.dot(xReshape, W) + B

    #Save a cache for use later in the backprop:
    cache = (x, w, b)

    return(out, cache)
```

Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

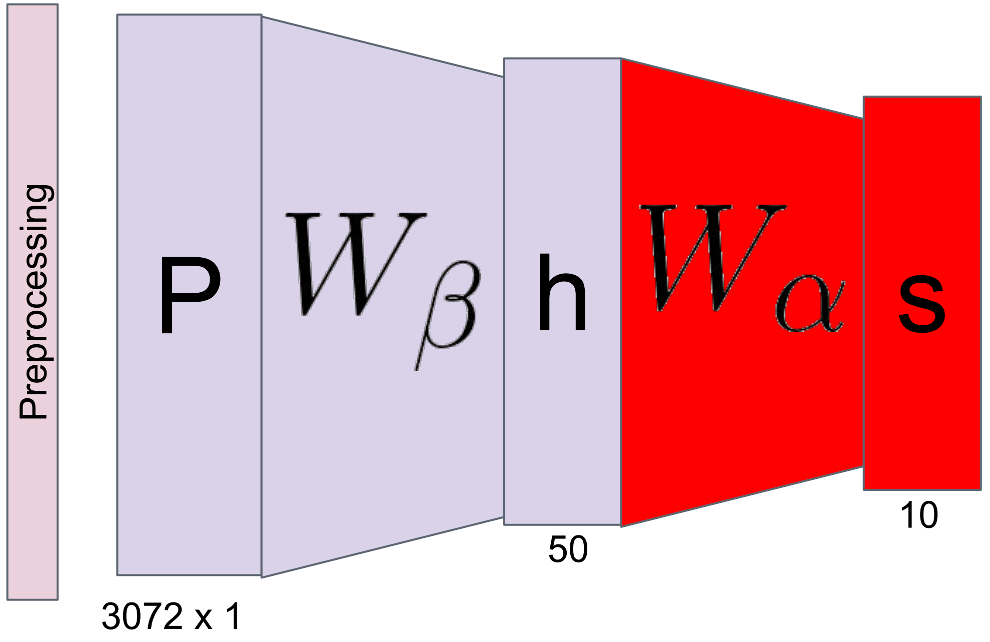
Architecture:




```python
def reluForward(reluInput):
    out = np.maximum(reluInput, 0)
    cache = reluInput
    return(out, cache)
```

Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

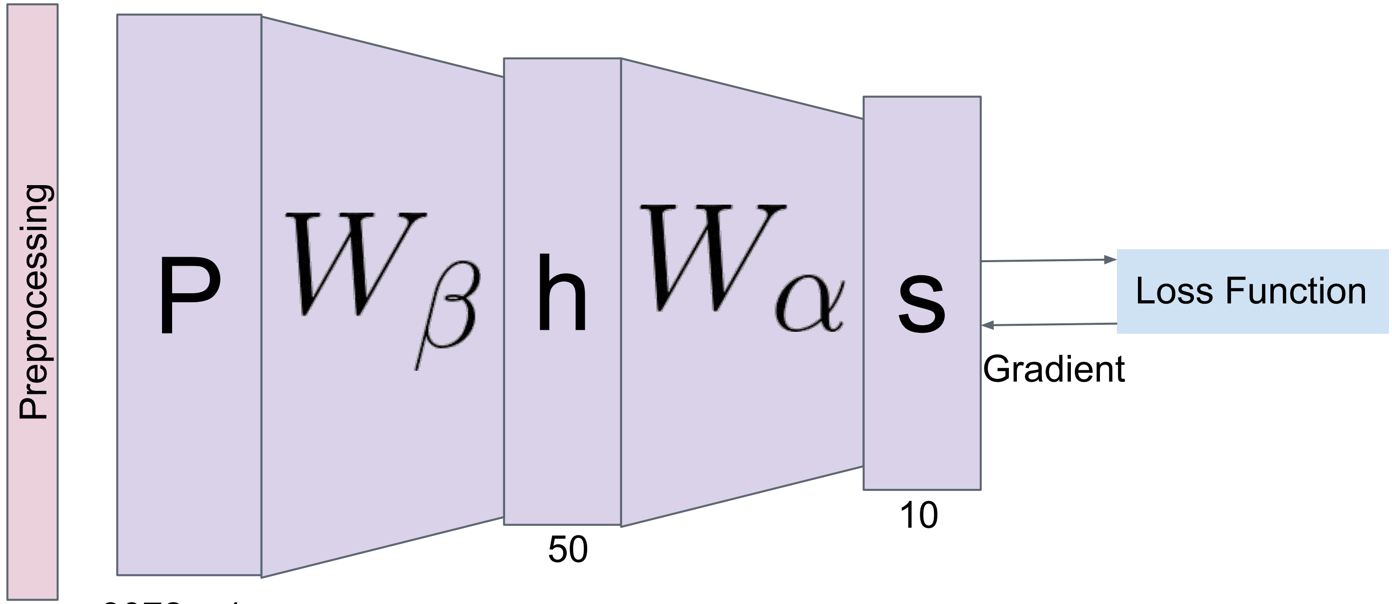
Architecture:




```python
def affineForward(X, W, B):
    #Total number of observations:
    N = X.shape[0]

    #Number of dimensions - in this example, 3072 (i.e., each observation has 3072 values)
    D = np.prod(X.shape[1:])

    #Reshape our inputs to be (N,D), matching our expectation for the weights dot product.
    xReshape = np.reshape(X, (N, D))

    #Calculate the dot product:
    out = np.dot(xReshape, W) + B

    #Save a cache for use later in the backprop:
    cache = (x, w, b)

    return(out, cache)
```

Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

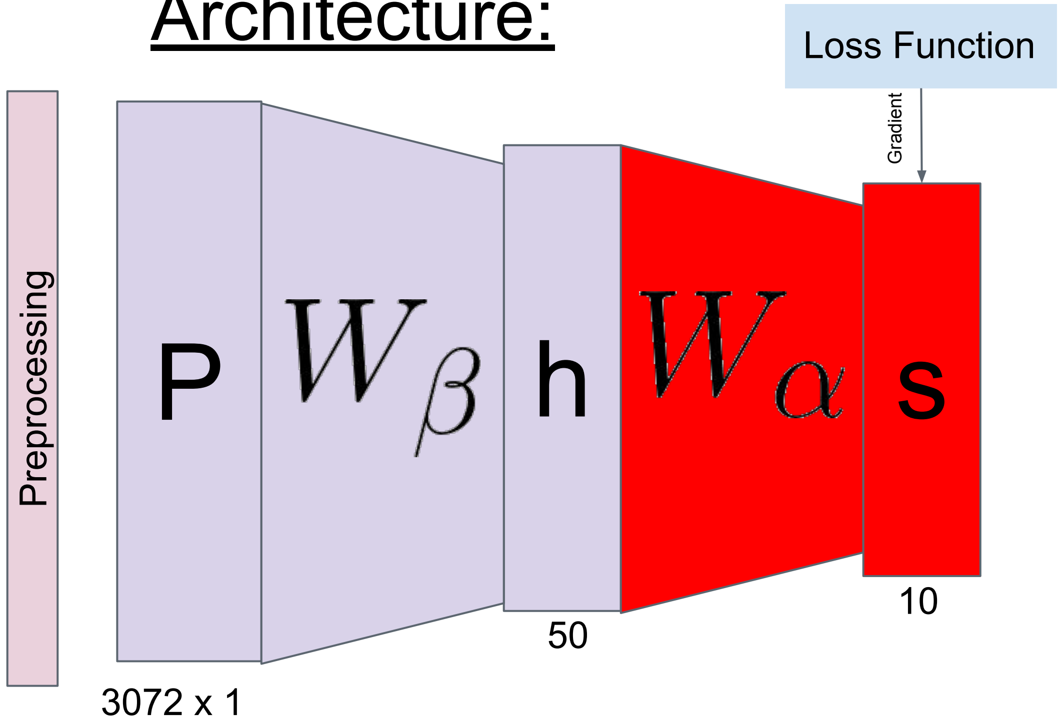
Architecture:




```python
def svmLoss(y, estimatedScores, e):
    N = estimatedScores.shape[0]

    #This takes the estimated score for the correct class, y.
    #correctClassScore will have one entry per observation.
    correctClassScore = estimatedScores[np.arange(N), y]

    #Now we calculate SVM loss, adding a new dimension to correctClassScore.
    margin = np.maximum(0, estimatedScores-correctClassScore[:,np.newaxis] + e)

    #Set our correct cases to 0 as per the SVM Loss function:
    margin[np.arange(N), y] = 0

    #Calculate the total loss
    loss = np.sum(margin)

    #Now we want to solve for our gradients.
    #Because SVM loss only changes if the value is greater than 0,
    #first we need to identify those cases.
    positiveCount = np.sum(margin>0, axis=1)

    #Now let's solve for dx - first create an empty matrix
    #the same size as our inputs (estimatedScores).
    dx = np.zeros_like(estimatedScores)

    #Identify each case with a postiive value
    dx[margin > 0] = 1

    #Because the true cases result in a negative change, we subtract
    #the total positive cases from the y entries:
    dx[np.arange(N), y] -= positiveCount

    #And, finally, divide by our sample size
    dx /= N

    return loss, dx
```

Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

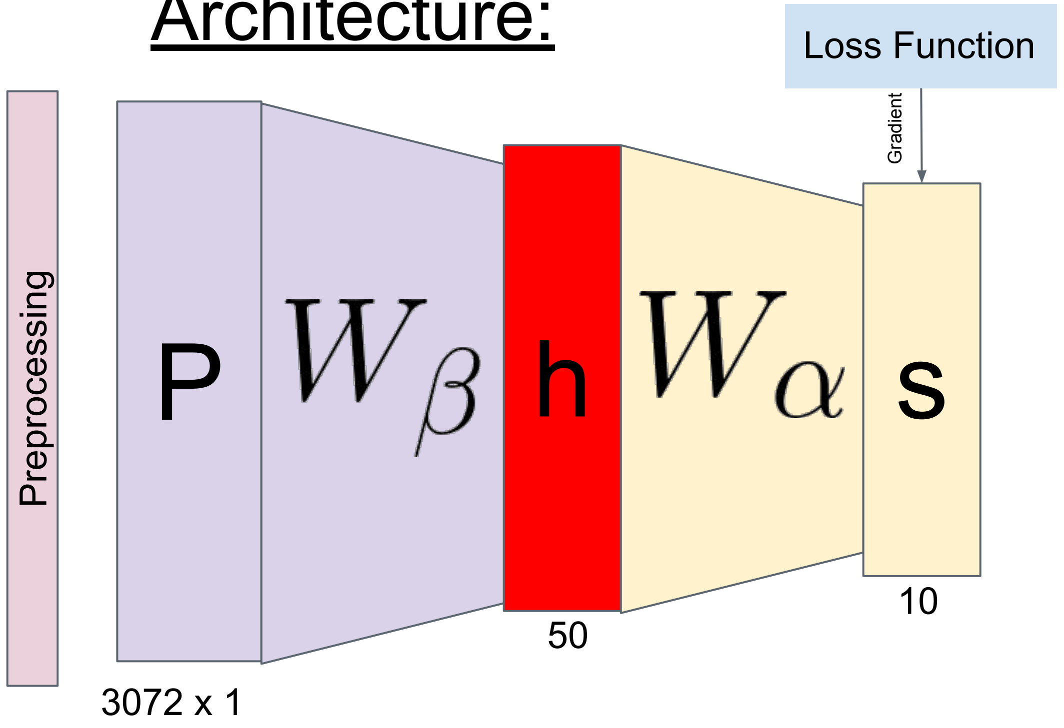
Architecture:




```
def affineBackward(dUpstream, cache):
    X, W, B = cache

    #Same steps as the forward pass:
    N = X.shape[0]
    D = np.prod(X.shape[1:])
    xReshape = np.reshape(X, (N, D))

    #Gradient calculations for the affine case - nothing you haven't
    #seen before!
    dx = np.reshape(np.dot(dUpstream, W.T), X.shape)
    dw = np.dot(xReshape.T, dUpstream)
    db = np.dot(dUpstream.T, np.ones(N))

    return(dx, dw, db)
```

Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

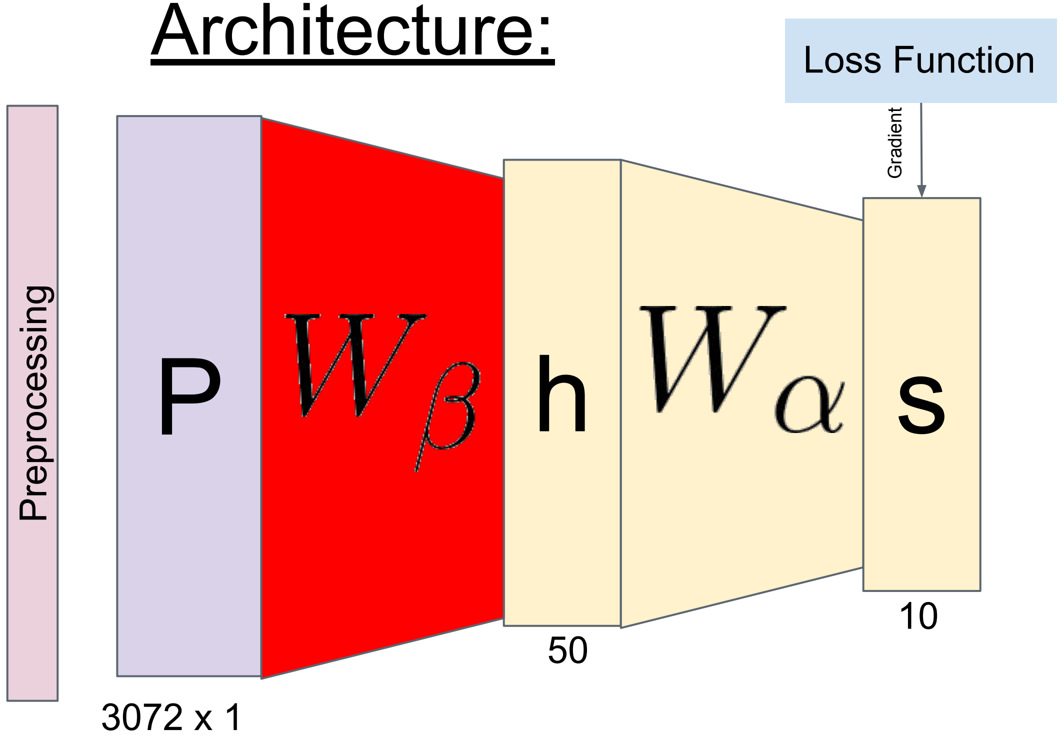
<u>Architecture:</u>




```python
def reluBackward(upstreamGradient, cache):
    x = cache

    #Remember this gradient is just copying our incoming,
    #and then setting anything less than 0 to 0!
    dx = np.array(upstreamGradient, copy=True)
    dx[x <= 0] = 0

    return(dx)
```

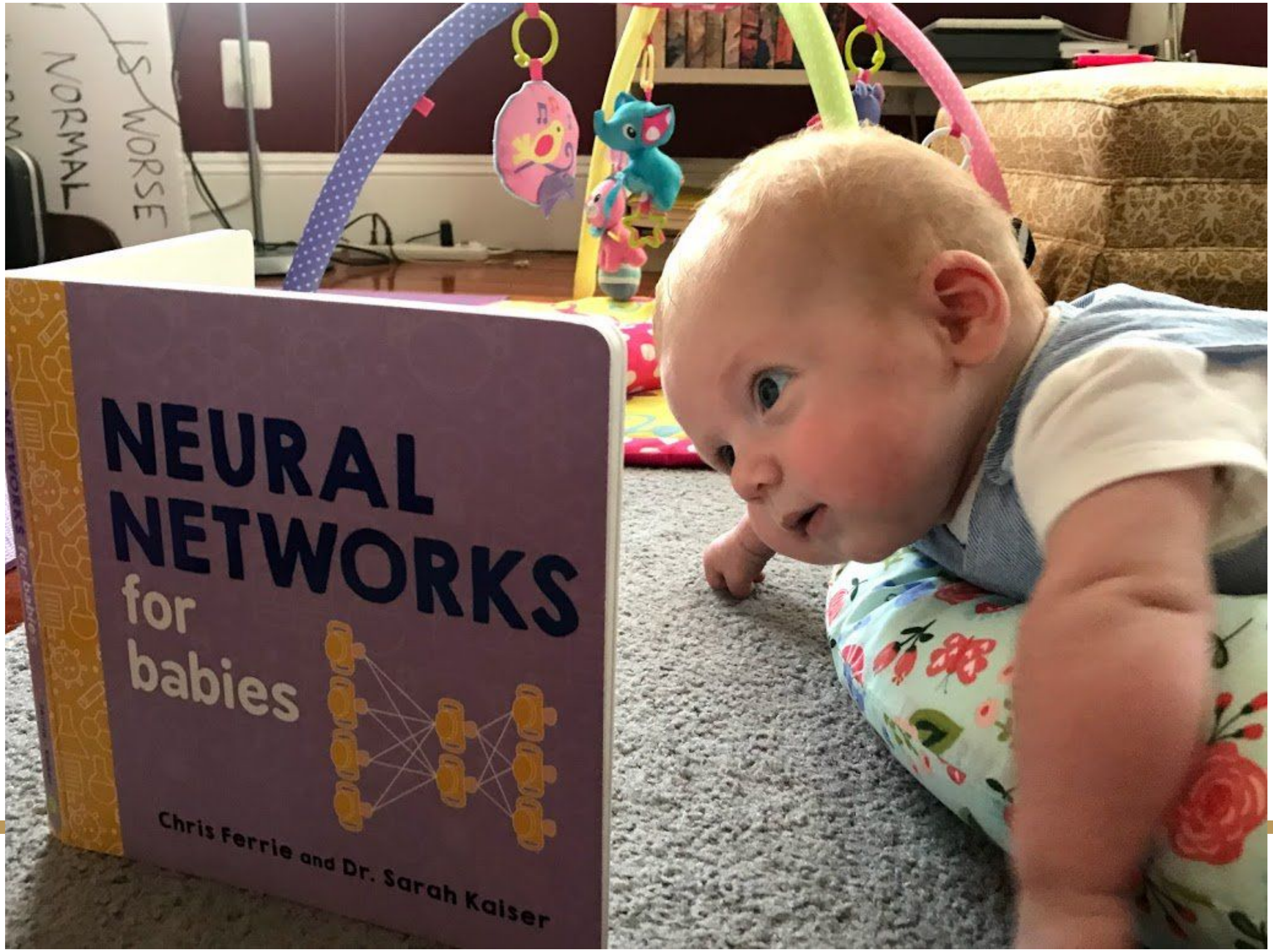
Preprocessing: Zero Centered Data
Weights Initialization: He
Activations: ReLU

Architecture:

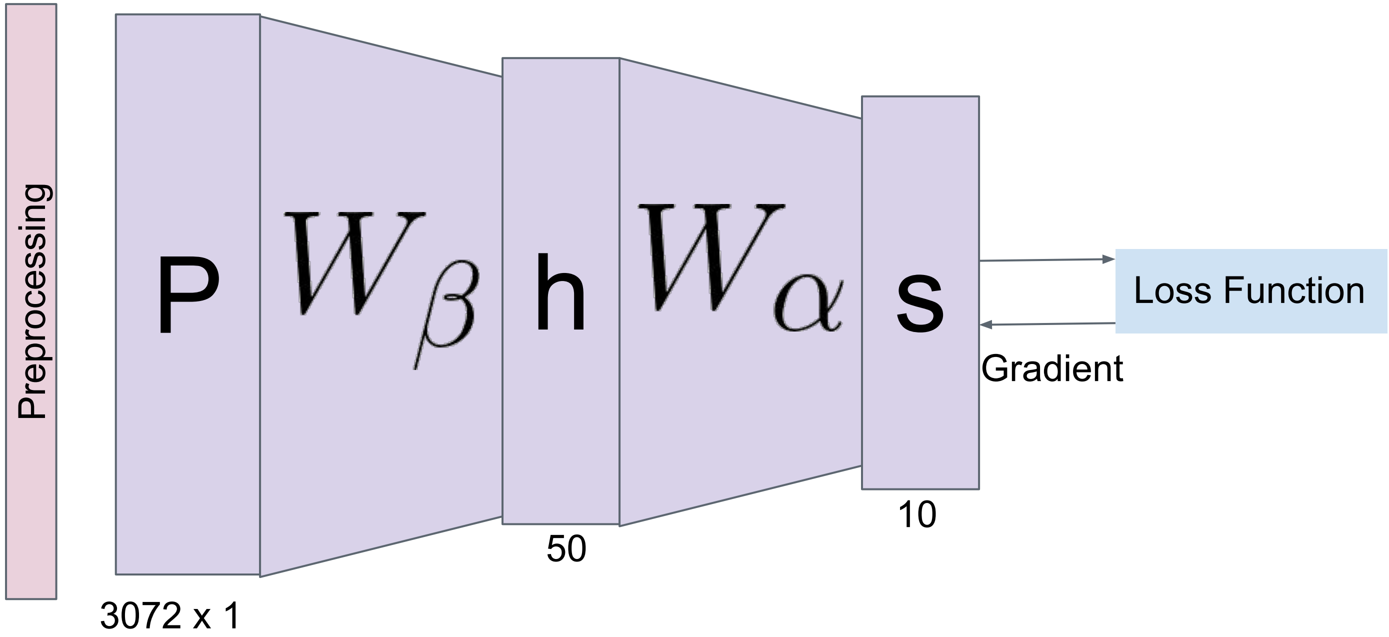



```
def affineBackward(dUpstream, cache):
    X, W, B = cache

    #Same steps as the forward pass:
    N = X.shape[0]
    D = np.prod(X.shape[1:])
    xReshape = np.reshape(X, (N, D))

    #Gradient calculations for the affine case - nothing you haven't
    #seen before!
    dx = np.reshape(np.dot(dUpstream, W.T), X.shape)
    dw = np.dot(xReshape.T, dUpstream)
    db = np.dot(dUpstream.T, np.ones(N))

    return(dx, dw, db)
```

Practical Considerations for your Nets



Network Architecture & Learning



Make sure your Weights Matrix isn't 0s

One of the most common problems you'll run into is that your gradients are all 0 - i.e., no changes are being made. Print your matrix to check this; this can be because you've saturated, or a poor weights initialization scheme, or just a bug in your code.

```
fullPass = twoLayerNet(X = X_train[0:1], y = y_train[0:1], modelParameters = modelParametersInit)
print("Loss: " + str(fullPass[0]))
print("Gradient of W2 (example): \n" + str(fullPass[1]['W2']))
```

```
Loss: 3224.327062904839
Gradient of W2 (example):
[[   0.            0.            0.            0.           95.23316494
     0.          -95.23316494    0.            0.            0.        ]
 [   0.            0.            0.            0.           50.30978818
     0.          -50.30978818    0.            0.            0.        ]
 [   0.            0.            0.            0.           75.59835015
     0.          -75.59835015    0.            0.            0.        ]
 [   0.            0.            0.            0.            0.
     0.            0.            0.            0.            0.        ]
```

Double Check your Loss Function

Another common issue is a miscalculated loss function - i.e., you coded it wrong, or the loss function you chose isn't appropriate for your distribution of data / outcome goals. Always print it to confirm the value makes sense!

```
fullPass = twoLayerNet(X = X_train[0:1], y = y_train[0:1], modelParameters = modelParametersInit)
print("Loss: " + str(fullPass[0]))
print("Gradient of W2 (example): \n" + str(fullPass[1]['W2']))

Loss: 3224.327062904839
Gradient of W2 (example):
[[   0.           0.           0.           0.          95.23316494
     0.         -95.23316494   0.           0.           0.        ]
 [   0.           0.           0.           0.          50.30978818
     0.         -50.30978818   0.           0.           0.        ]
 [   0.           0.           0.           0.          75.59835015
     0.         -75.59835015   0.           0.           0.        ]
 [   0.           0.           0.           0.           0.
     0.           0.           0.           0.           0.        ]
```

Trade Note: It is helpful to disable any regularization while doing this debugging.

Double Check your Loss Function

You can also solve for the expected values to make sure you're getting the magnitude right.

$$L_i = -log(\frac{e^s_k}{\sum_{j=1}^{J} e^s_j})$$

Debugging Regularization

$$R(W) = \sum_{k=1}^{K} W_k^2$$

Creating a Dev Dataset

Always, always, always do this before any real runs.

Everything is working! Now what?

Learning Rate = .00001

```
while currentIteration < maxIterations:
    randomSelection = np.random.randint(len(X_train), size=batchSize)
    xBatch = X_train[randomSelection,:]
    yBatch = y_train[randomSelection]

    iterationModel = twoLayerNet(X = xBatch, y = yBatch, modelParameters = modelParameters)
    plotData['iterationLoss'].append(iterationModel[0])
    plotData['correctlyClassifiedImagesPercent'].append(iterationModel[2])


    modelParameters['W1'] += -learningRate * iterationModel[1]['W1']
    modelParameters['W2'] += -learningRate * iterationModel[1]['W2']
    modelParameters['B1'] += -learningRate * iterationModel[1]['B1']
    modelParameters['B2'] += -learningRate * iterationModel[1]['B2']

    currentIteration = currentIteration + 1

    print("Iteration: "+ str(currentIteration) + ": ")
    print("Average Weight 1: " + str(iterationModel[1]['W1'].mean()))
    print("Average Change in Weights Paramter 1" + str((-learningRate * iterationModel[1]['W1']).mean()))
    print("=============")

#plotFit(plotData = plotData, title="Network Gradient Descent Optimization")
```

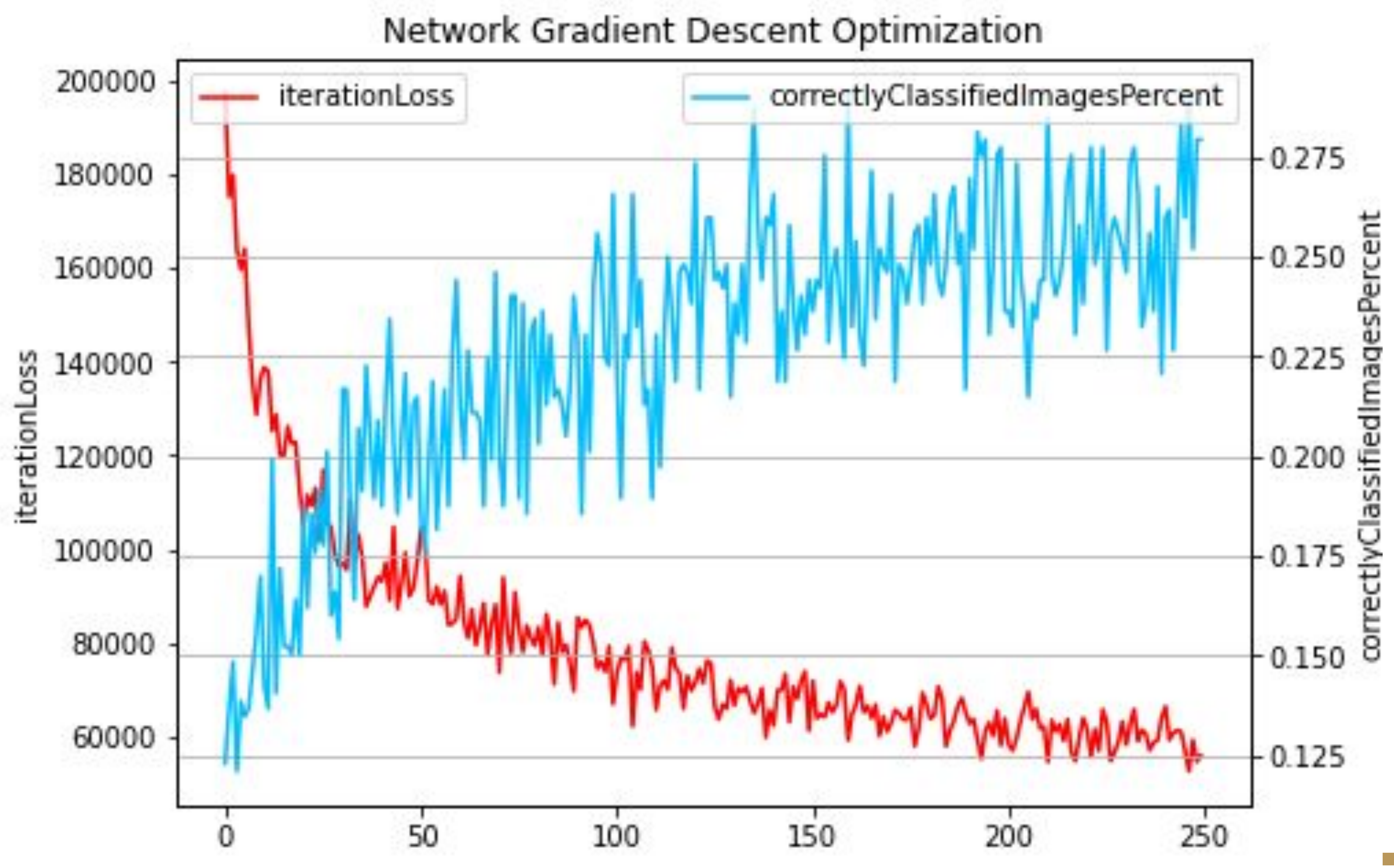
```
Iteration: 1:
Average Weight 1: 0.4004396529218216
Average Change in Weights Paramter 1-4.004396529218216e-06
=============
Iteration: 2:
Average Weight 1: 0.410183971775558424
Average Change in Weights Paramter 1-4.101839717755843e-06
=============
Iteration: 3:
Average Weight 1: 0.11339039257647404
Average Change in Weights Paramter 1-1.1339039257647405e-06
=============
```

Programmatically Searching for LR

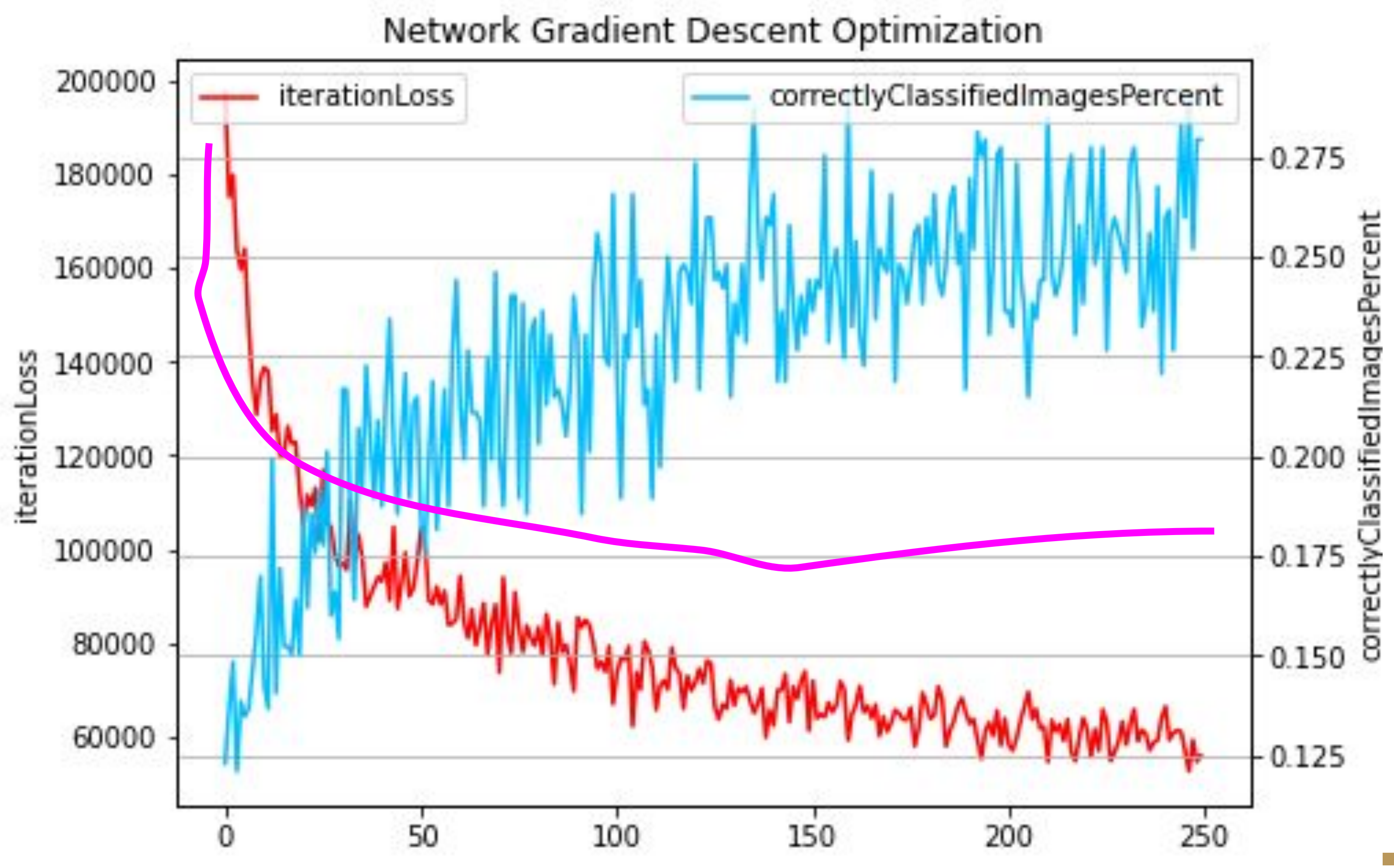
You can easily write a loop that automatically tests different learning rates - i.e., starting with .0001 and searching all values from .0001 to .01. Use a small number of epochs for this test. Iterate over smaller regions to find optimal cases.

```
for lr in rates:
    m.compile(optimizer=SGD(learning_rate = lr),
          metrics=['categorical_accuracy'],
          loss='categorical_hinge')
    m.fit(x=X_train, y=y_train,
      batch_size=64,
      epochs=5,
      validation_data=(X_val,y_val),
      verbose = 0)
    iterationLoss = m.evaluate(x=X_test, y=y_test)
    print("LR: " + str(lr) + " Loss: " + str(iterationLoss[1]))
```

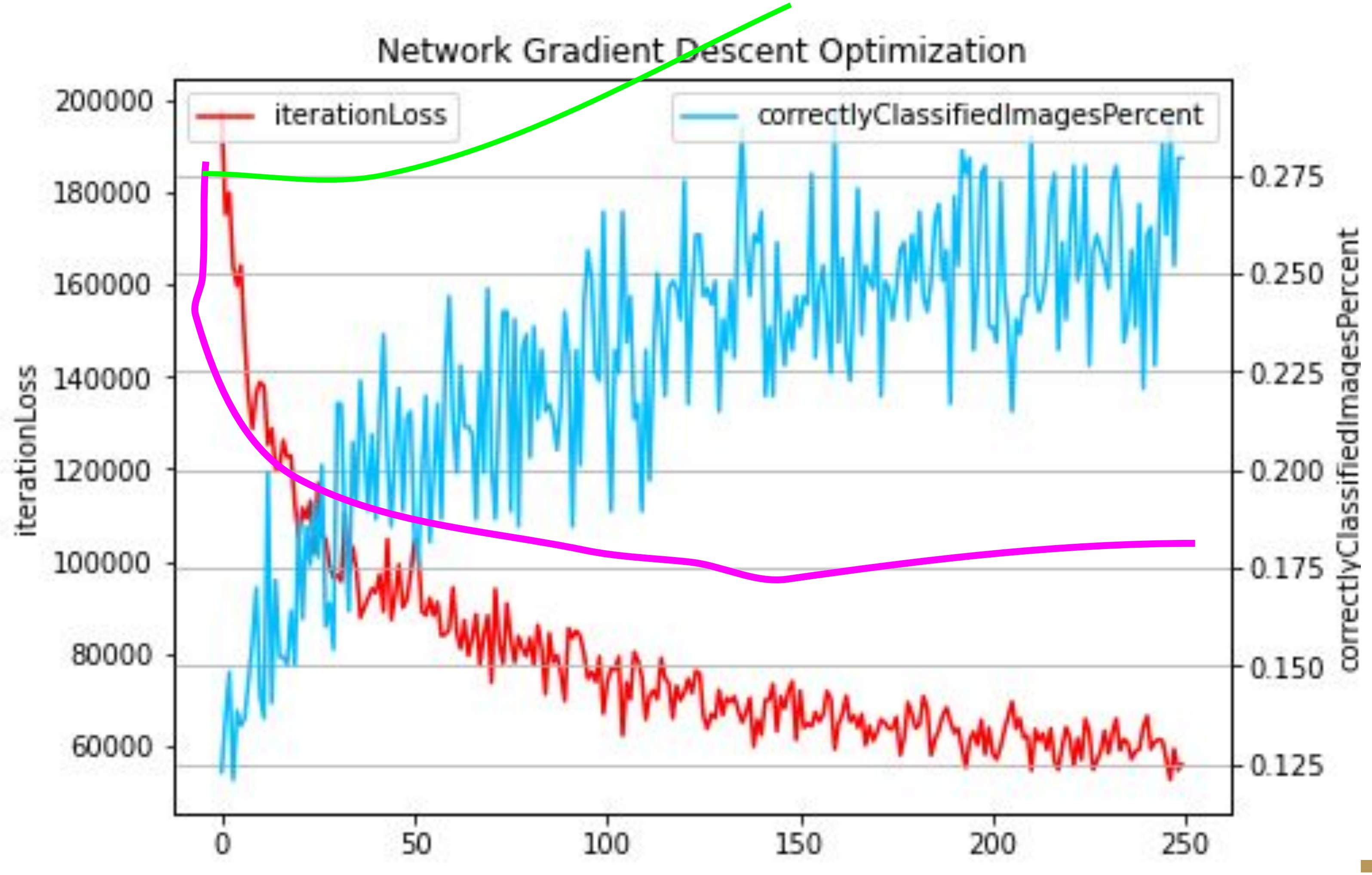
What is a "Good" Learning Rate?



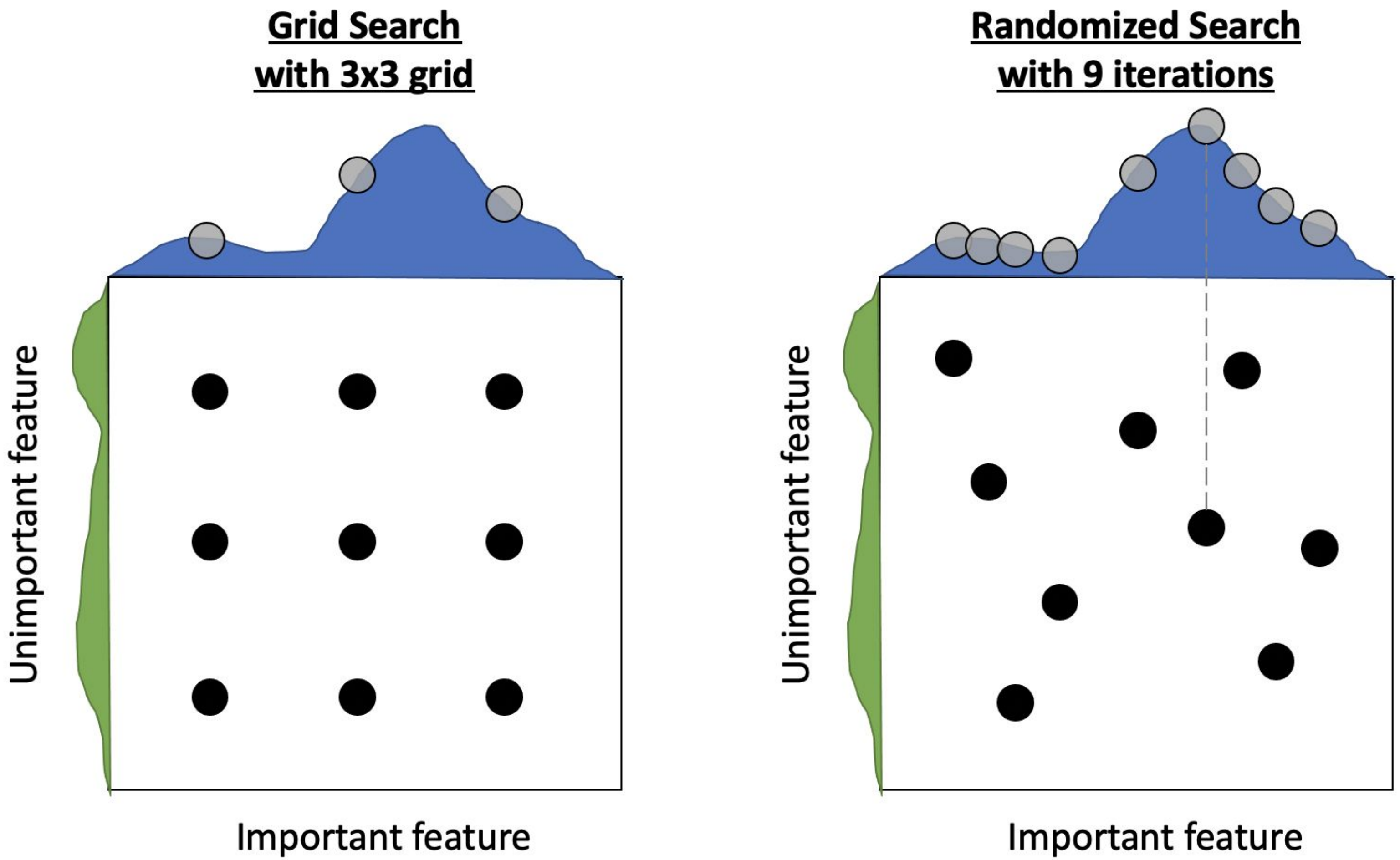
What is a “Good” Learning Rate?



What is a “Good” Learning Rate?

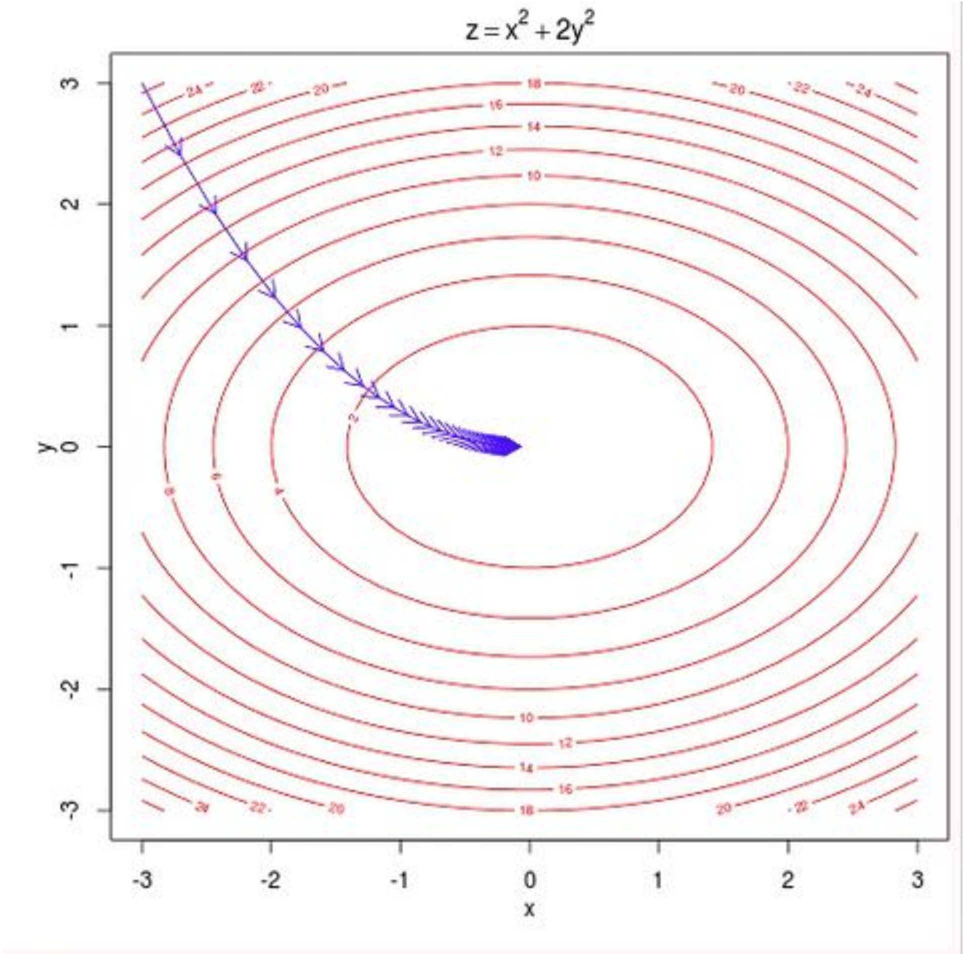


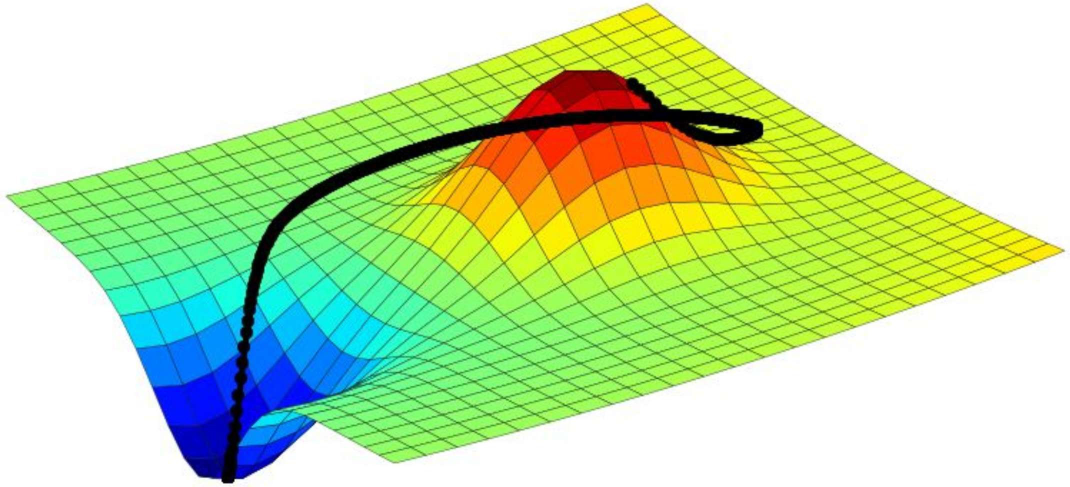
More effective programmatic searches



https://towardsdatascience.com/gridsearch-vs-randomizedsearch-vs-bayesiansearch-cfa76de27c6b?gi=e49a04a83798

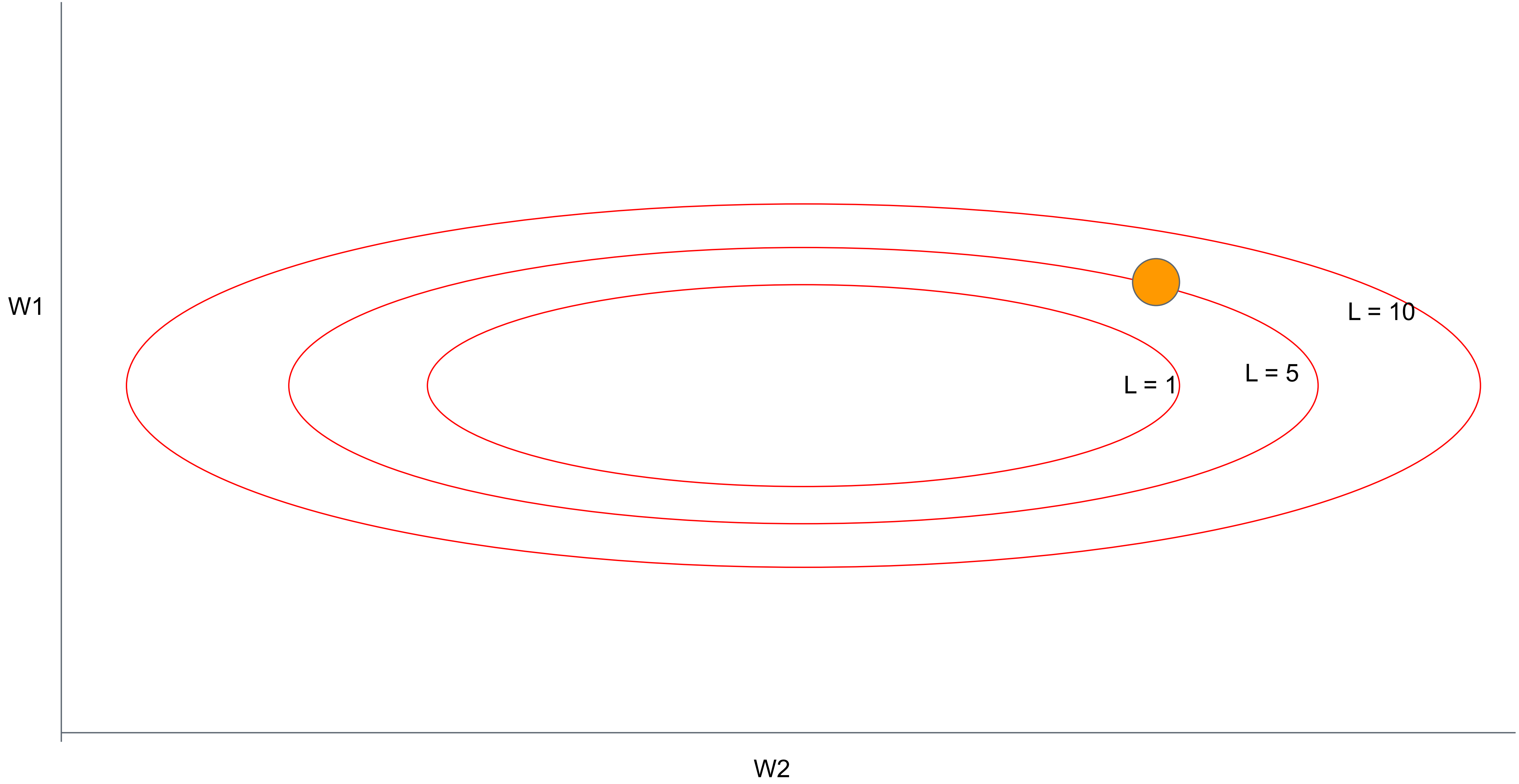
More Advanced Optimization

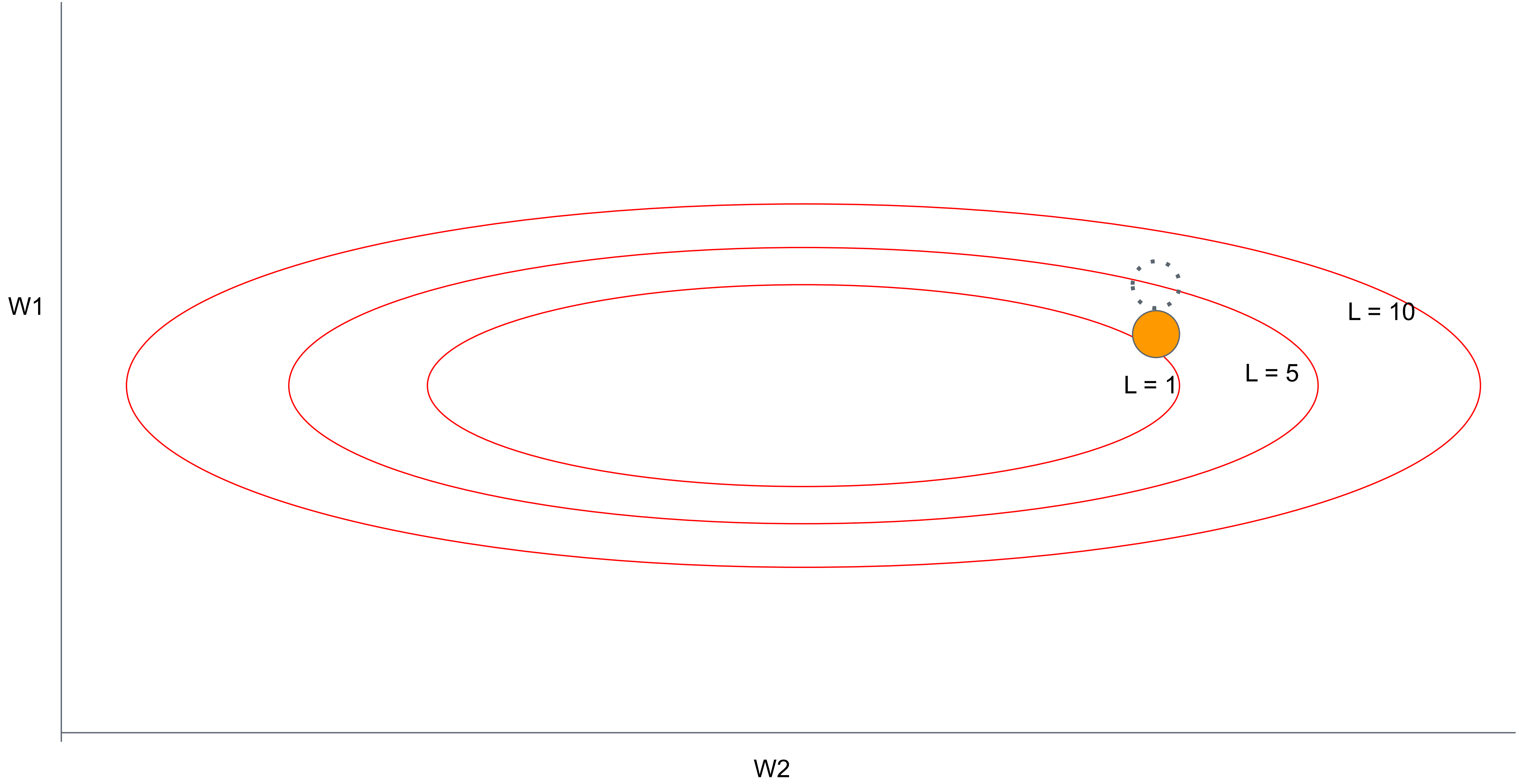


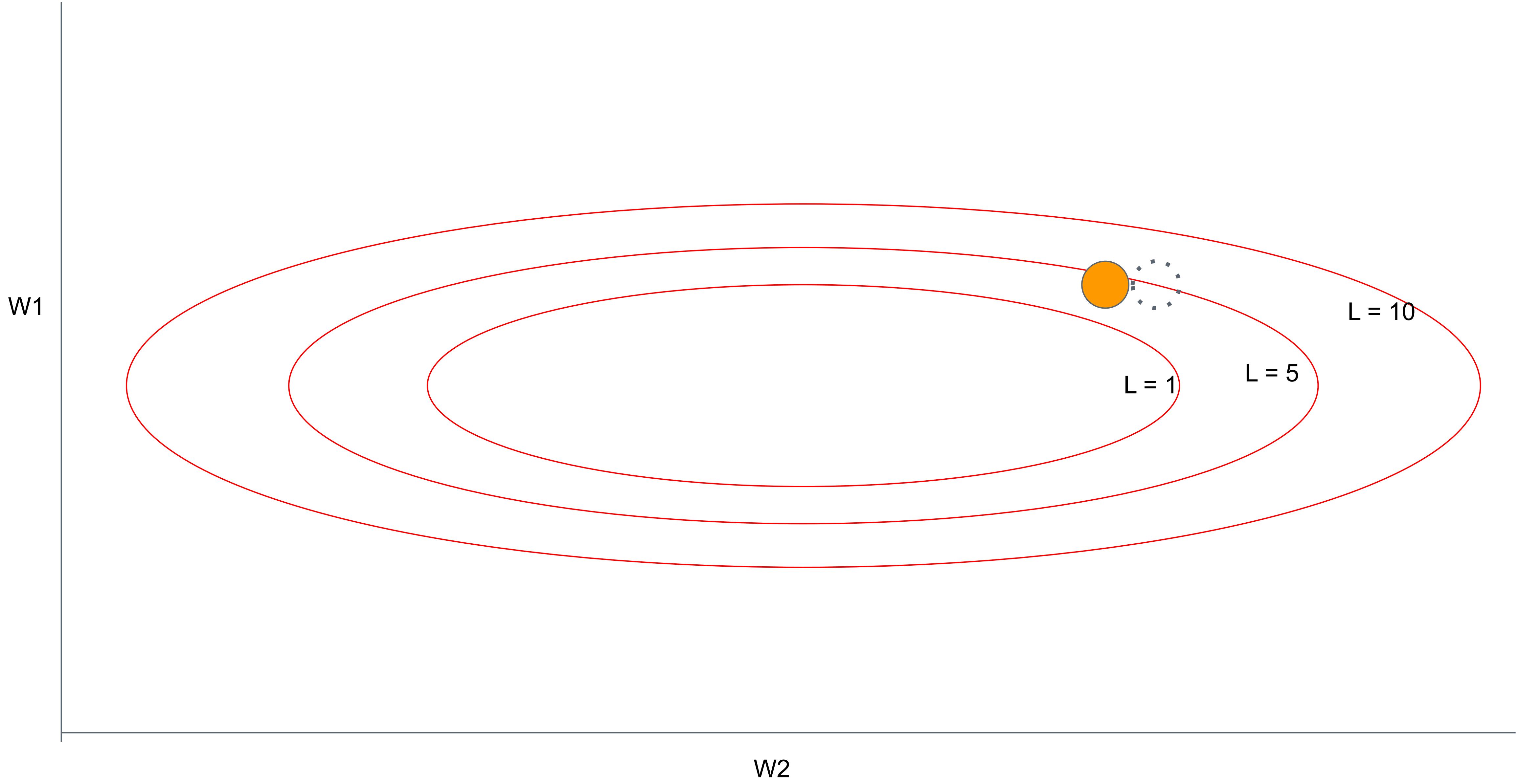


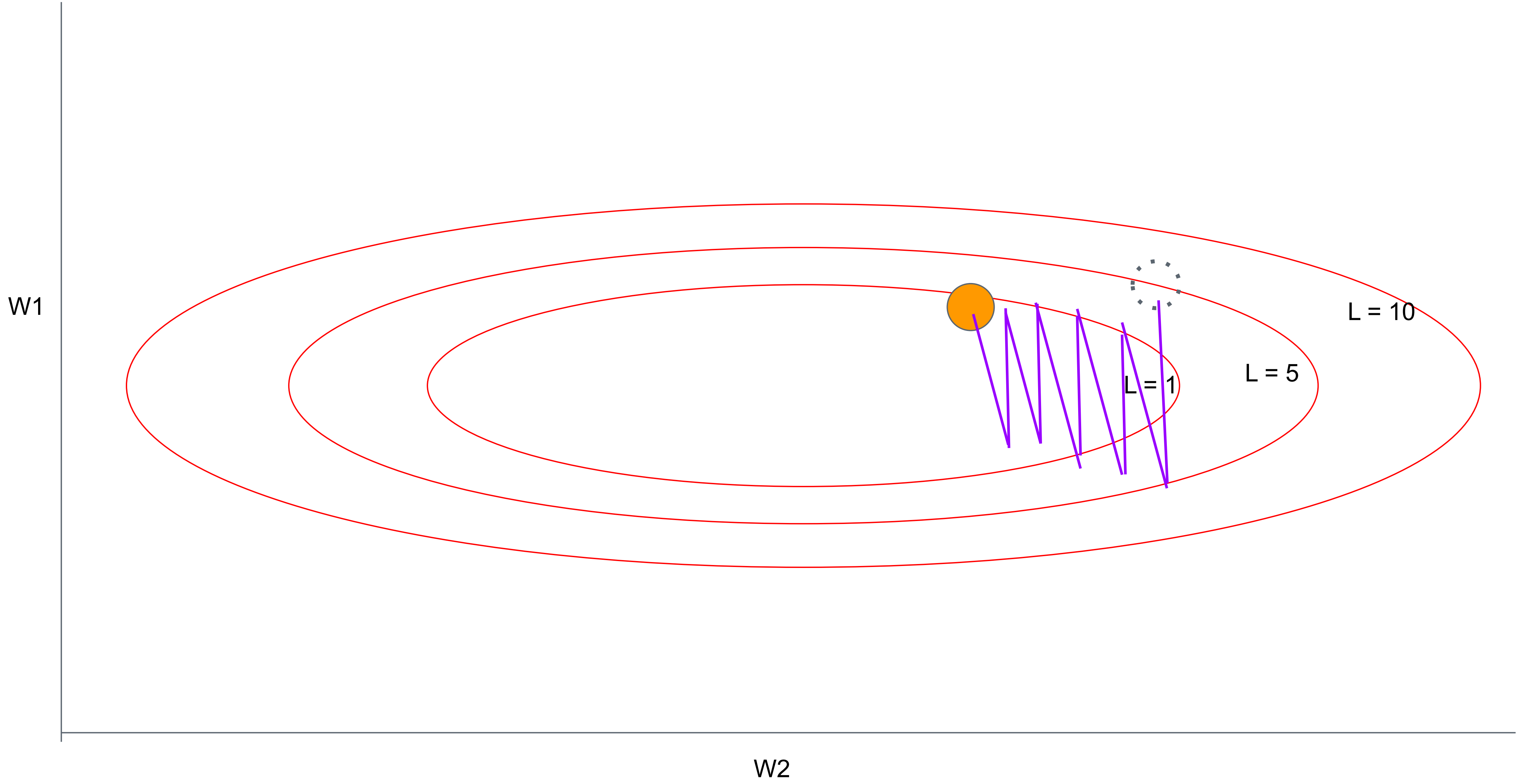
http://hduongtrong.github.io/2015/11/23/coordinate-descent/

https://ozzieliu.com/2016/02/09/gradient-descent-tutorial/

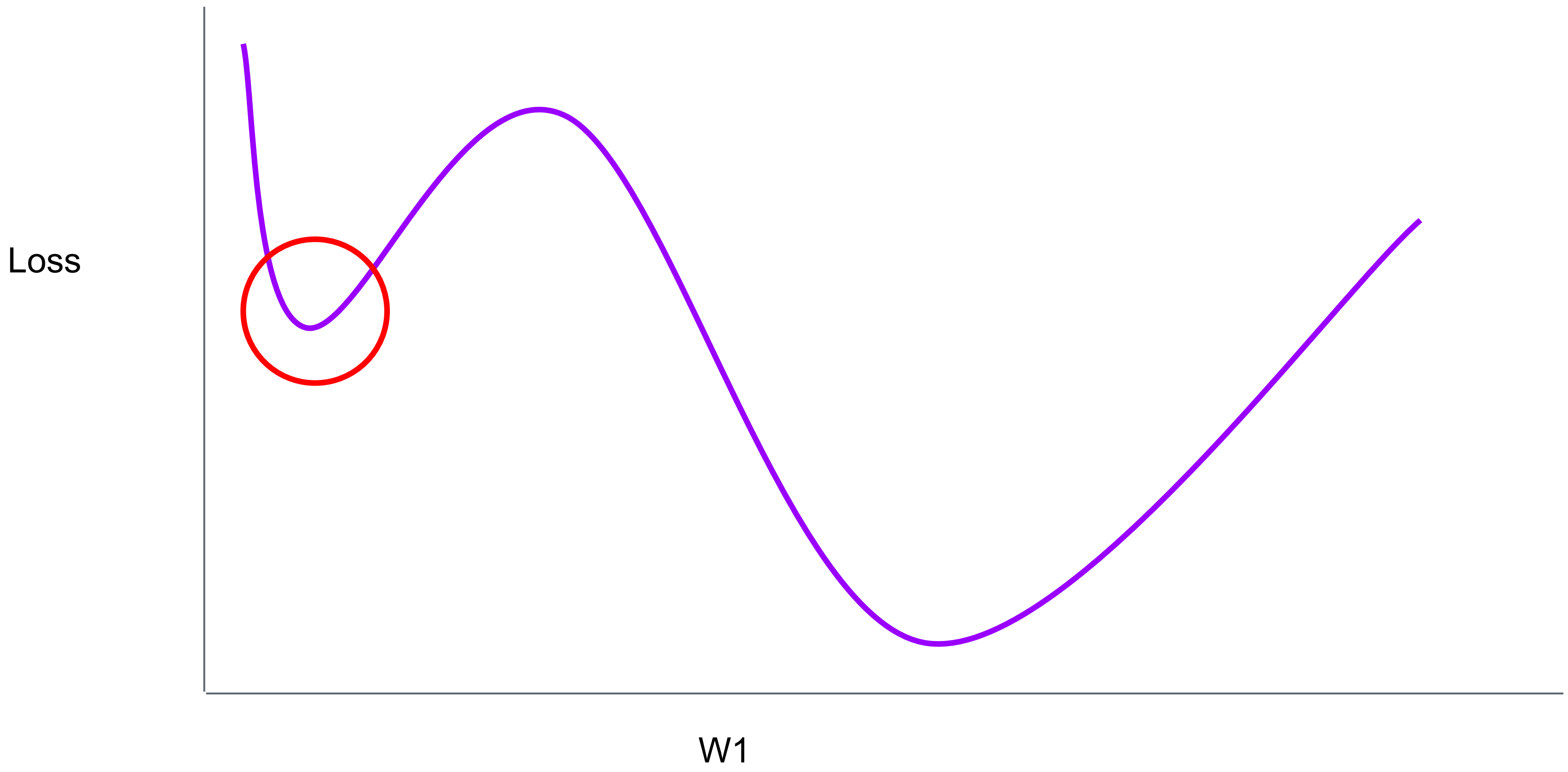

W1
L = 10
L = 5
L = 1
W2


W1
L = 10
L = 5
L = 1
W2

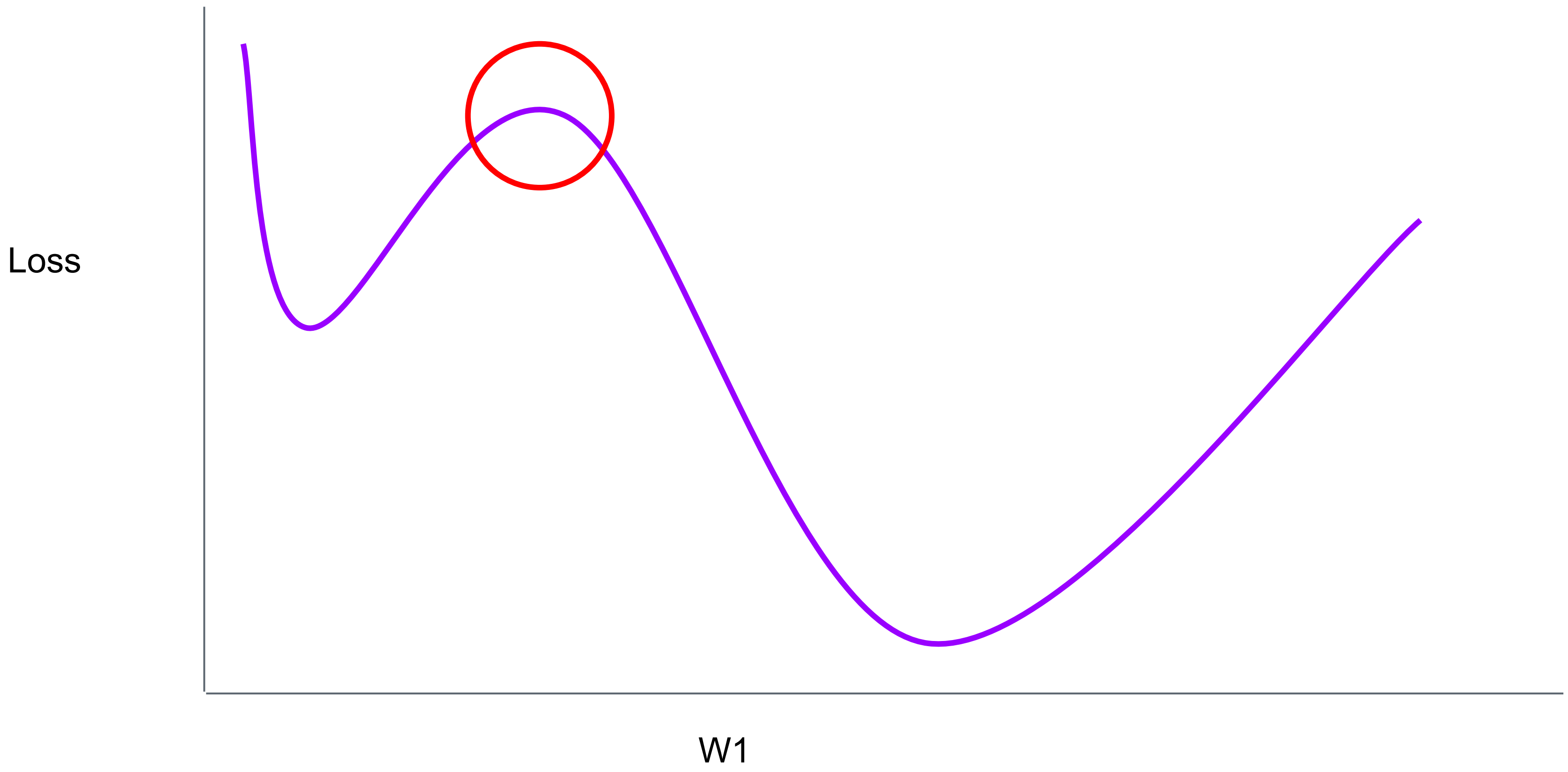

W1
L = 10
L = 5
L = 1
W2


W1
L = 10
L = 5
L = 1
W2

Local Minima



Saddle Point



SGD: $$W_{iteration+1} = W_{iteration} - \alpha \Delta f(W_{iteration})$$

SGD: $$W_{iteration+1} = W_{iteration} - \alpha \Delta f(W_{iteration})$$

SGD + Momentum:

$$V_{i+1} = \rho V_i + \Delta f(W_i)$$

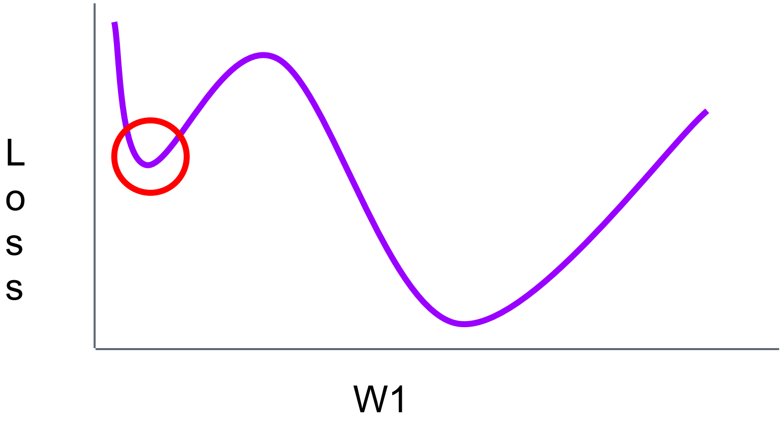
$$W_{i+1} = W_i - \alpha V_{t+1}$$

SGD + Momentum:

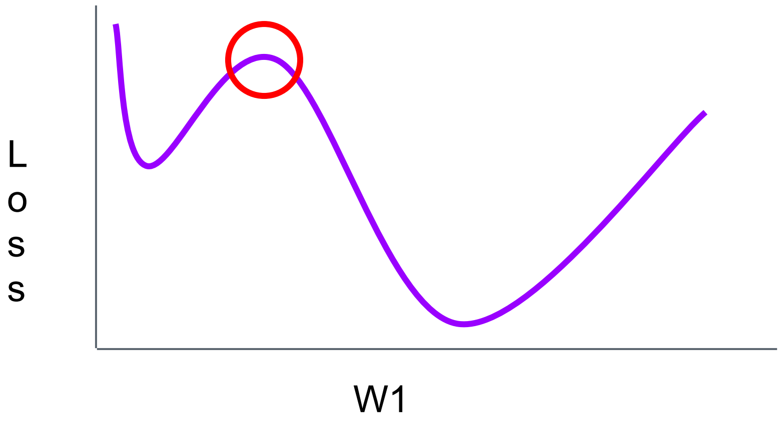
$$V_{i+1} = \rho V_i + \Delta f(W_i)$$

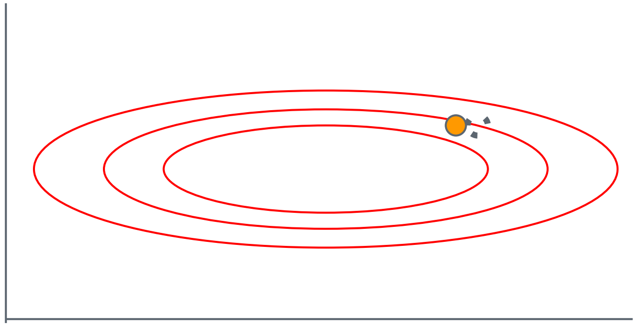
$$W_{i+1} = W_i - \alpha V_{t+1}$$

Local Minima



Saddle Points





Poor Conditioning

AdaGrad (Duchi et al.)

SGD: $$W_{iteration+1} = W_{iteration} - \alpha\Delta f(W_{iteration})$$

AdaGrad:

$$W_{i+1} = W_i - \alpha\Delta/(\sqrt{\gamma} + .0000001)$$

$$\gamma = \sum_i^N \Delta^2$$

RMSProp (Tieleman and Hinton)

AdaGrad:

$$W_{i+1} = W_i - \alpha\Delta/(\sqrt{\gamma} + .00000001)$$

$$\gamma = \sum_i^N \Delta^2$$

RMSProp: $$\gamma = \sum_i^N (\rho\Delta^2 + (1-\rho) * \Delta^2))$$

ADAM (Kingma and Ba)

Beta 1 - Similar to Friction in SGD + Momentum

Beta 2 - Similar to Decay Rate in RMSProp

Practical Tip: Beta1 = 0.9, beta2=0.99, LR = 1e-3 can provide a strong starting condition for tests with Adam.

Summary

- Implementing Networks
- Practical Considerations for Network Fitting
  - Debugging Issues
  - Development Datasets & Loss = 0
  - Learning Rate
  - Grid vs. Randomized Searches for Hyperparameters
- Optimization Algorithms
  - Limitations of SGD
  - SGD + Momentum
  - AdaGrad
  - RMSProp
  - ADAM