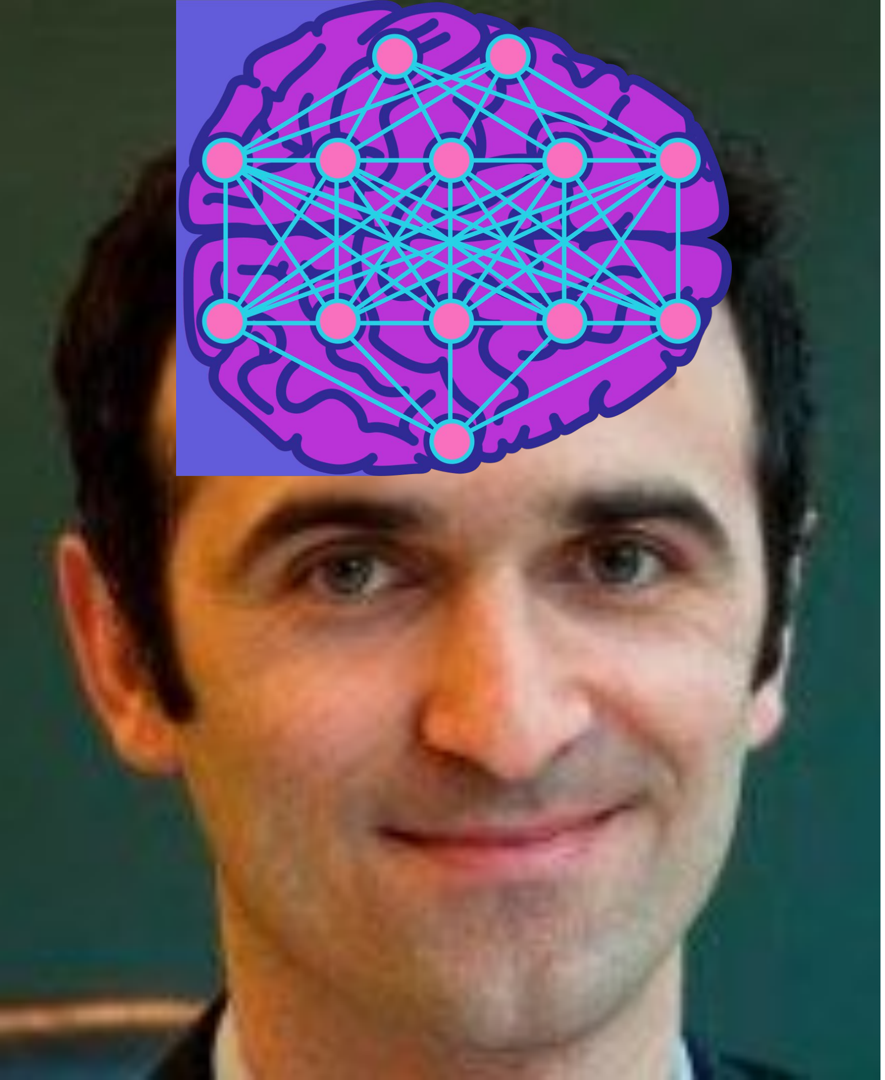


Optimization of neural networks

Kenwan Cheung, Joan Lee, Chantel Miller, Tony Smaniotto

Agenda

- Background
- Neural Networks and Optimization
- Hand Coded Neural Net Implementation
- Overview of NN Optimizers
  - Gradient Descent: Batch, Mini-Batch, Stochastic, AdaGrad
- Optimizer Comparison
- Conclusion

Background: Neural Networks

Artificial neural networks were introduced in 1943 and after periods of falling in and out of favor have now entered into “a virtuous cycle of funding and progress.” Thanks to the availability of huge quantities of data to train neural networks and increases in computing power, which reduce the time to train, neural networks now “frequently outperform other [machine learning] techniques on very large and complex problems.”[1]

As neural networks have gotten deeper and more complex, parameter estimation and the relevant optimization methodology is now a critical step to building neural networks.

Our goal:

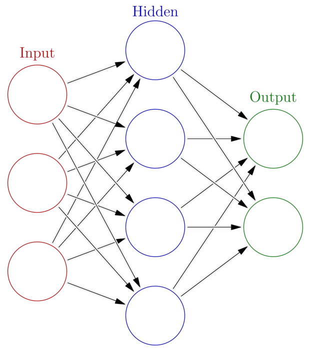
Our team will seek to hand-code neural networks while implementing several of the most common optimization methods such as: gradient descent, batch GD, stochastic GD, and AdaGrad. We will seek to gain a deeper understanding over how they converge by leveraging the age old Francisco™ method: simulation, development of our own functions, and visualization. We will complete this in Python.

1. Hands-On Machine Learning with Scikit-Learn and TensorFlow

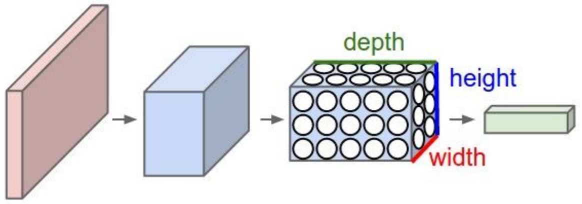
How are neural networks related to optimization?

The number of parameters to be estimated is calculated by the sum of the product of the numbers of nodes in connected layers.

Example below: Parameters:
(3×4)+(4×2)=20



In a convolutional neural network:
Parameter estimation 100M+!

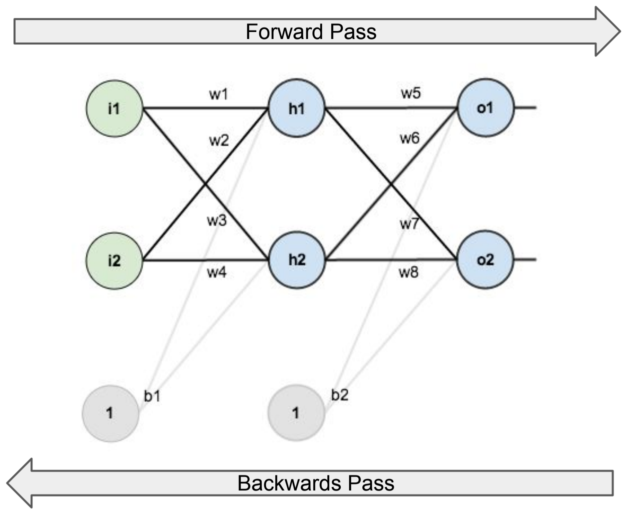


Optimization of millions of parameters is non trivial!

How does the optimization happen?

Forward and backwards passing:

Weights for the parameters are estimated at multiple layers. Errors are then passed ***backwards*** to identify the gradient of the error with respect to ***each layer*** through the chain rule.



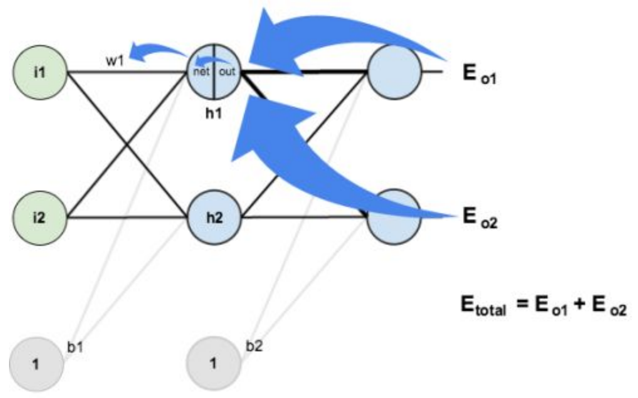
How is this actually accomplished?

The error is calculated at the final layer, and passed back with respect to the layer preceding it, and the layer preceding it etc…

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

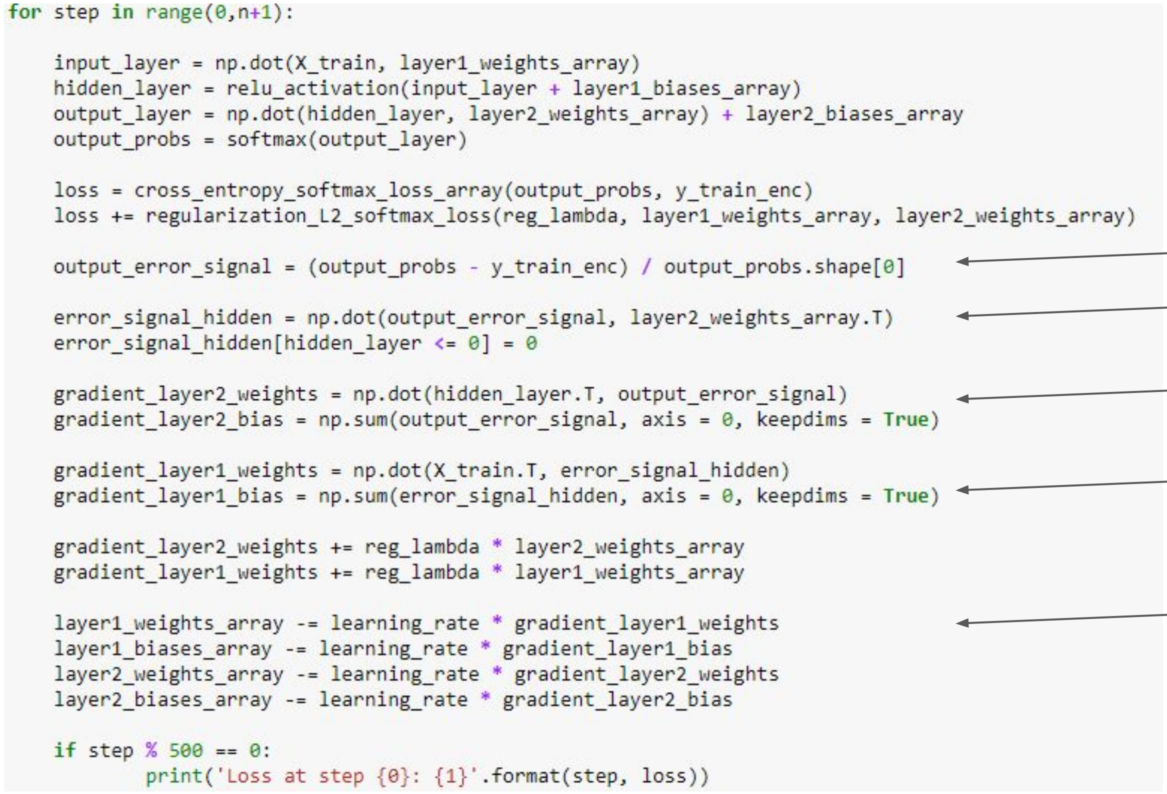
$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



$E_{total} = E_{o1} + E_{o2}$

Let’s take a look at the implementation



```python
for step in range(0,n+1):

    input_layer = np.dot(X_train, layer1_weights_array)
    hidden_layer = relu_activation(input_layer + layer1_biases_array)
    output_layer = np.dot(hidden_layer, layer2_weights_array) + layer2_biases_array
    output_probs = softmax(output_layer)

    loss = cross_entropy_softmax_loss_array(output_probs, y_train_enc)
    loss += regularization_L2_softmax_loss(reg_lambda, layer1_weights_array, layer2_weights_array)

    output_error_signal = (output_probs - y_train_enc) / output_probs.shape[0]

    error_signal_hidden = np.dot(output_error_signal, layer2_weights_array.T)
    error_signal_hidden[hidden_layer <= 0] = 0

    gradient_layer2_weights = np.dot(hidden_layer.T, output_error_signal)
    gradient_layer2_bias = np.sum(output_error_signal, axis = 0, keepdims = True)

    gradient_layer1_weights = np.dot(X_train.T, error_signal_hidden)
    gradient_layer1_bias = np.sum(error_signal_hidden, axis = 0, keepdims = True)

    gradient_layer2_weights += reg_lambda * layer2_weights_array
    gradient_layer1_weights += reg_lambda * layer1_weights_array

    layer1_weights_array -= learning_rate * gradient_layer1_weights
    layer1_biases_array -= learning_rate * gradient_layer1_bias
    layer2_weights_array -= learning_rate * gradient_layer2_weights
    layer2_biases_array -= learning_rate * gradient_layer2_bias

    if step % 500 == 0:
            print('Loss at step {0}: {1}'.format(step, loss))
```

Error at final layer

Update at hidden layer

Update at layer 2

Update at layer 1

Updating new weights

Backwards Pass

Gradient Descent

The most popular Optimization algorithm used in optimizing a Neural Network.

Formula:

$W = W - \alpha \nabla J(W,b)$

where 'α**'** is the learning rate ,'$\nabla J(W,b)$**'** is the **Gradient** of ***Loss function*-J(**W,b**)** w.r.t changes in the weights.

Challenges:

1. How do I pick the learning rate?
2. The same learning rate applies to all parameters… can foresee a scenario where that might not be the case!
3. Local minima

Batch Gradient Descent

Traditional implementation of gradient descent. Calculates gradient of the entire dataset and will perform a single update.

Formula:

$$W = W - \alpha \nabla J(W,b) \qquad J(W,b) = \frac{1}{m}\sum_{z=0}^{m} J(W,b,x^{(z)},y^{(z)})$$

Where **W** are the weights, **α** is the learning rate ,**∇J(W,b)** is the gradient of the loss function **J(W,b)** w.r.t changes in the weights calculated on all of the **m** training samples

Challenges:

1. Can be slow and hard to control for large datasets that may not fit in memory
2. Can't be efficiently parallelised
3. Smooth nature of the reducing cost function tends to lead to the neural network training getting stuck in local minimums

http://adventuresinmachinelearning.com/stochastic-gradient-descent/

Stochastic Gradient Descent

Updates weight parameters after evaluating the cost function for each sample, usually faster than BGD.

Formula:

$$W = W - \alpha \nabla J(W, b, x^{(z)}, y^{(z)})$$

Where **W** are the weights, **α** is the learning rate ,**∇J(W,b)** is the gradient of the loss function **J(W,b)** w.r.t changes in the weights calculated for each individual sample

Challenges:

1. Parameter updates have high variance and cause the loss function to fluctuate
2. Due to the intense fluctuations, can complicate convergence to the exact minimum
3. Sensitive to feature scaling

http://adventuresinmachinelearning.com/stochastic-gradient-descent/

Mini-batch Gradient Descent

Updates weight parameters after evaluating the cost function for every batch of training examples.

Formula:

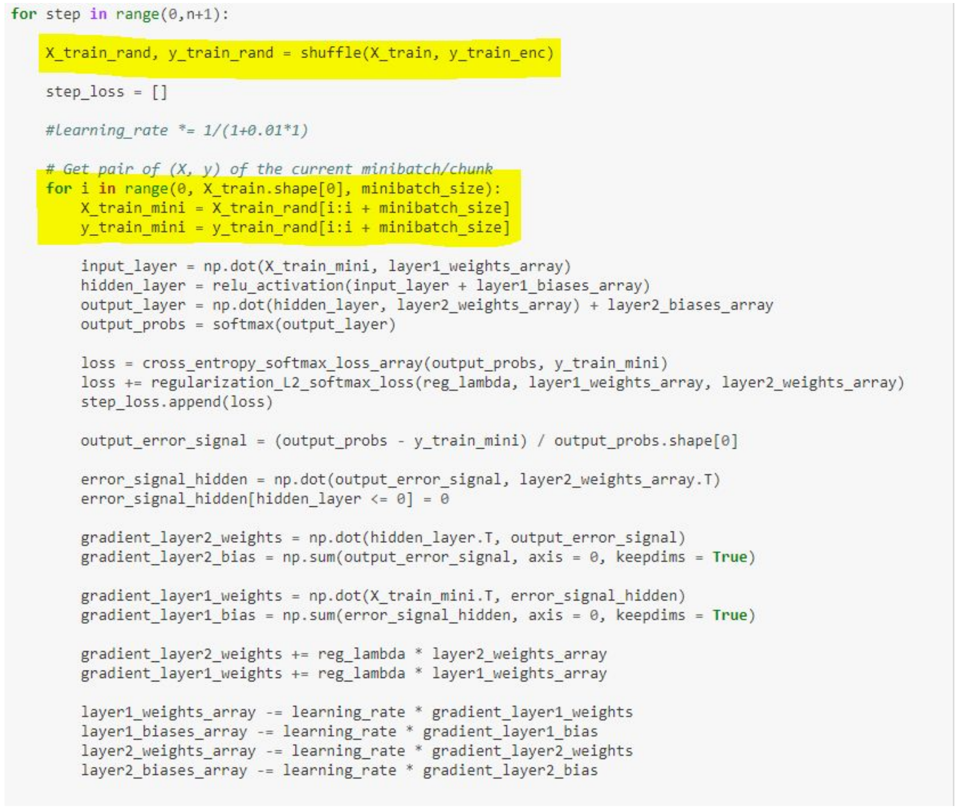
$$W = W - \alpha \nabla J(W, b, x^{(z:z+bs)}, y^{(z:z+bs)}) \qquad J(W, b, x^{(z:z+bs)}, y^{(z:z+bs)}) = \frac{1}{bs}\sum_{z=0}^{bs} J(W, b, x^{(z)}, y^{(z)})$$

Where **W** are the weights, **α** is the learning rate ,**∇J(W,b)** is the gradient of the loss function **J(W,b)** w.r.t changes in the weights calculated for each mini-batch **bs** of samples.

Why mini-batch is the best of both worlds:

1. Reduces the variance in the parameter updates (compared to stochastic gradient descent)
2. Updating weights in batches decreases likelihood of sticking in local minimum (compared to batch gradient descent)
3. Can make use of highly optimized matrix optimizations common to deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient

Implementation of SGD and MBGD



```
for step in range(0,n+1):

    X_train_rand, y_train_rand = shuffle(X_train, y_train_enc)

    step_loss = []

    #learning_rate *= 1/(1+0.01*1)

    # Get pair of (X, y) of the current minibatch/chunk
    for i in range(0, X_train.shape[0], minibatch_size):
        X_train_mini = X_train_rand[i:i + minibatch_size]
        y_train_mini = y_train_rand[i:i + minibatch_size]

        input_layer = np.dot(X_train_mini, layer1_weights_array)
        hidden_layer = relu_activation(input_layer + layer1_biases_array)
        output_layer = np.dot(hidden_layer, layer2_weights_array) + layer2_biases_array
        output_probs = softmax(output_layer)

        loss = cross_entropy_softmax_loss_array(output_probs, y_train_mini)
        loss += regularization_L2_softmax_loss(reg_lambda, layer1_weights_array, layer2_weights_array)
        step_loss.append(loss)

        output_error_signal = (output_probs - y_train_mini) / output_probs.shape[0]

        error_signal_hidden = np.dot(output_error_signal, layer2_weights_array.T)
        error_signal_hidden[hidden_layer <= 0] = 0

        gradient_layer2_weights = np.dot(hidden_layer.T, output_error_signal)
        gradient_layer2_bias = np.sum(output_error_signal, axis = 0, keepdims = True)

        gradient_layer1_weights = np.dot(X_train_mini.T, error_signal_hidden)
        gradient_layer1_bias = np.sum(error_signal_hidden, axis = 0, keepdims = True)

        gradient_layer2_weights += reg_lambda * layer2_weights_array
        gradient_layer1_weights += reg_lambda * layer1_weights_array

        layer1_weights_array -= learning_rate * gradient_layer1_weights
        layer1_biases_array -= learning_rate * gradient_layer1_bias
        layer2_weights_array -= learning_rate * gradient_layer2_weights
        layer2_biases_array -= learning_rate * gradient_layer2_bias
```

Same implementation as previously shown except now within each epoch we are looping through mini-batches of our randomized training set and updating weights based on those batches.

Common mini-batch sizes are between 10-500, often chosen as multiples of 2^n.

Stochastic Gradient Descent would be implemented with a minibatch_size of 1.

Adaptive Gradient Descent (AdaGrad)

The Adaptive Gradient algorithm adapts the learning rate to its parameters. AdaGrad differs from typical Stochastic Gradient Descent in that instead of updating all our parameters using one learning rate, **AdaGrad updates each parameter using a different learning rate**.

In AdaGrad, we calculate the **sums of the squares of the gradients** for each iterative step (represented by the G diagonal matrix below), and then update our parameters.

AdaGrad is well-suited for sparse data (i.e. data with lots of boolean fields -- could extend to large Mixed-Integer LP problems). It is also convenient in that it is no longer necessary to fine-tune the learning rate in the first step.

However, as the algorithm iterates, the denominator gets infinitely small, which means that the performance of the optimizer plateaus at a certain point.

Formula:

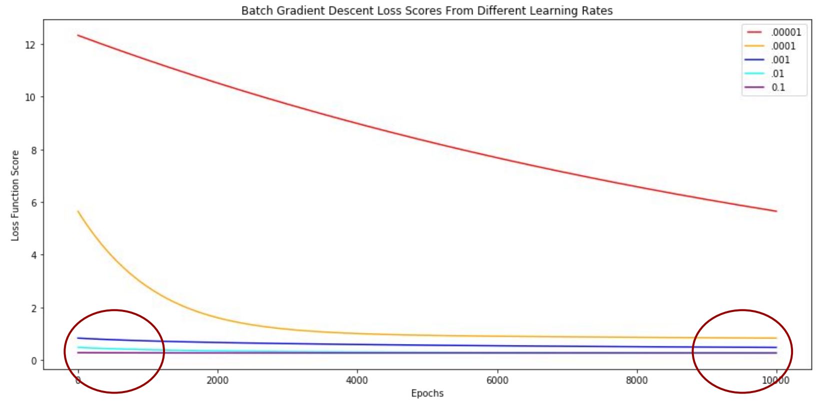
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} .$$

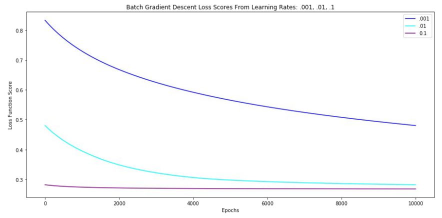
Where g is our gradient

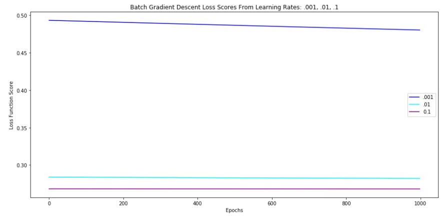
$$g_{t,i} = \nabla_\theta J(\theta_{t,i}) .$$

Parameter Tuning and Comparing Optimizers

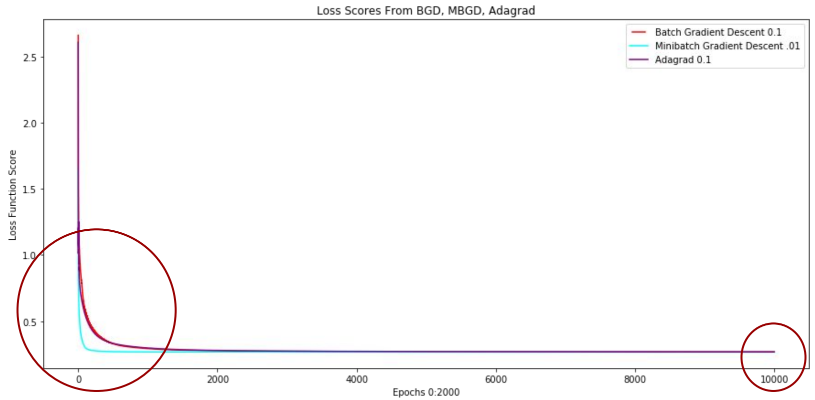
1. Choosing a proper learning rate is important, can be challenging
   a. Ex.: Batch gradient descent with learning rates: 1e-5...0.1

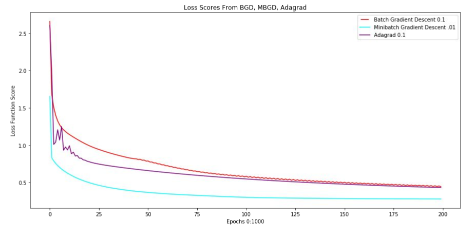


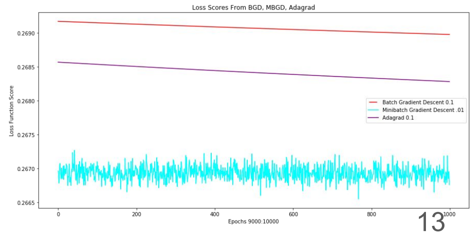




2. Choosing the right optimizer; i.e. one that converges quickly, learns properly and tunes internal parameters to minimize the loss function.
   a. Depends on your data







Conclusions

- Neural networks are optimization problems; there are many parameters to estimate and optimize
- For Gradient Descent and its variants, choosing the appropriate learning rate is important, can be difficult
- Choosing an appropriate optimizer depends on your data
  - SGD would perform poorly on sparse data, Adagrad would work better

Future work

- Finetuning parameters
- Explore more of the adaptive learning rate techniques, e.g. AdaDelta, Adam, etc.