ECE421
Assignment 1:
Linear and Logistic Regression

Due date: TBD

Submission: A hard copy of the report should be handed in the lecture
The code should be submitted electronically to: ece421ta2018@gmail.com

Objectives:

In this assignment, you will be investigating the classification performance of linear and logistic regression. In this assignment, you will be implementing the batch Gradient Descent optimization algorithm for a linear regression classifier and logistic regression classifer. After, you will compare the performance of your algorithm against a state-of-the-art optimization technique, ADAM using Stochastic Gradient Descent. For both linear and logistic regression, implementations will be done in Numpy. For comparison with ADAM, you will be allowed to use a Tensorflow implementation. You are encouraged to look up TensorFlow APIs for useful utility functions, at: `https://www.tensorflow.org/api_docs/python/`.

General Note:

- Full points are given for complete solutions, including justifying the choices or assumptions you made to solve each question. A written report should be included in the final submission.
- Homework assignments are to be solved in groups of two. You are encouraged to discuss the assignment with other students, but you must solve it within your own group. Make sure to be closely involved in all aspects of the assignment. Please indicate the contribution percentage from each group member at the beginning of your report.

1 Linear Regression [18 points]

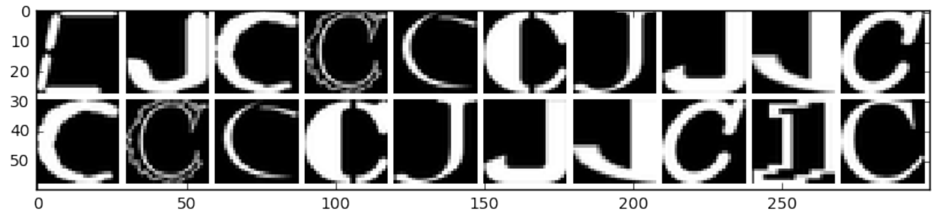
Linear regression can also be used for classification in which the training targets are either 0 or 1. Once the model is trained, we can determine an input's class label by thresholding the model's prediction. We will consider the following Mean Squared Error (MSE) loss function $\mathcal{L}_{\mathcal{D}}$ and weight decay loss $\mathcal{L}_W$ for training a linear regression model, in which the goal is to find the best total loss,

$$\begin{aligned}\mathcal{L} =& \mathcal{L}_\mathcal{D} + \mathcal{L}_W \\ =& \sum_{n=1}^{N} \frac{1}{2N} \|W^T\mathbf{x}^{(n)} + b - y^{(n)}\|_2^2 + \frac{\lambda}{2}\|W\|_2^2\end{aligned}$$

You will train linear regression model for classification on the *two-class notMNIST* dataset (described in the next section) by implementing the batch gradient descent algorithm. You will be using the validation set to tune the hyper-parameter of the weight decay regularizer.

Two-class notMNIST dataset

We use the following script to generate a smaller dataset that only contains the images from two letter classes: "C"(the positive class) and "J"(the negative class). This smaller subset of the data contains 3500 training images, 100 validation images and 145 test images.



```
with np.load('notMNIST.npz') as data :
    Data, Target = data ['images'], data['labels']
    posClass = 2
    negClass = 9
    dataIndx = (Target==posClass) + (Target==negClass)
    Data = Data[dataIndx]/255.
    Target = Target[dataIndx].reshape(-1, 1)
    Target[Target==posClass] = 1
    Target[Target==negClass] = 0
    np.random.seed(521)
    randIndx = np.arange(len(Data))
    np.random.shuffle(randIndx)
    Data, Target = Data[randIndx], Target[randIndx]
    trainData, trainTarget = Data[:3500], Target[:3500]
    validData, validTarget = Data[3500:3600], Target[3500:3600]
    testData, testTarget = Data[3600:], Target[3600:]
```

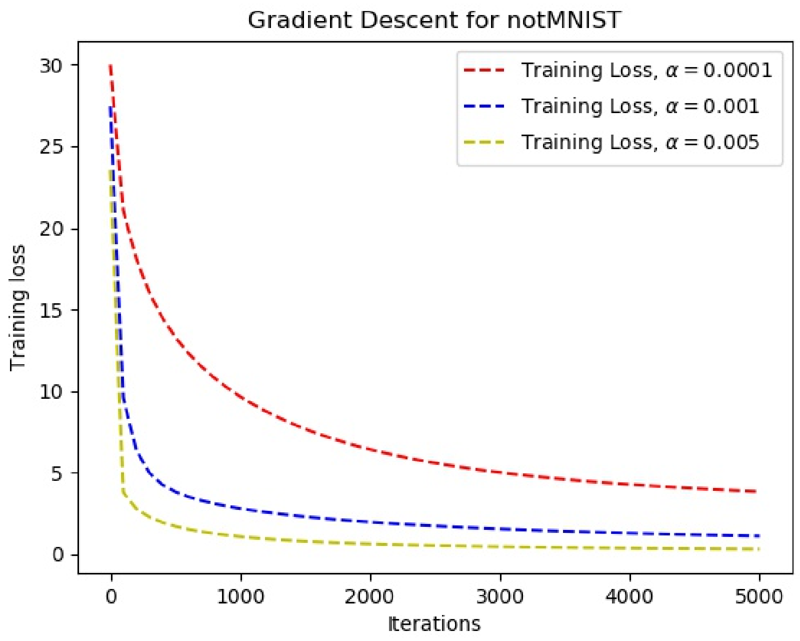
1. **Loss Function and Gradient [4 pts]:**

```
def MSE(W, b, x, y, reg):

    x = np.transpose(x)
    pred = np.dot(np.transpose(W), x)+b
    return np.squeeze(np.add(np.mean((y - pred) ** 2), np.dot(np.transpose(W), W)
        * reg / 2), axis=0)

def gradMSE(W, b, x, y, reg)
    x = np.transpose(x)
    pred = np.dot(np.transpose(W), x)+b
    grad_W = -2*np.mean(np.multiply(y-pred, x), axis=1)
    grad_W = np.expand_dims(grad_W, axis=1)
    grad_W += 2*reg*W
    # grad_W = np.expand_dims(grad_W, axis=1)
    grad_b = -2*np.mean(y-pred)
    grad_b = np.expand_dims(grad_b, axis=1)
    return grad_W, grad_b
```

2. **Gradient Descent Implementation [6 pts]:**

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol):
def grad_descent(W, b, x, y, alpha, iterations, reg, EPS, lossType="None"):
    #initialize the old weight matrix
    if lossType == None:
        print("No loss specified")
    training_losses = []

    iters = []
    W_old = W - 10*EPS
    W_new = W.copy()
    b_new = b.copy()
    current_iter = 0
    while np.linalg.norm(W_new-W_old) > EPS and current_iter < iterations:
        W_old = W_new.copy()
        if lossType == "MSE":
            grad_W, grad_b = gradMSE(W_new, b_new, x, y, reg)
            W_new -= alpha*grad_W
            b_new -= alpha*grad_b
            if current_iter % 100 == 0:
                training_loss = MSE(W_new, b, x, y, reg)
                print("Current Iteration: %d" % current_iter)
                print("Average MSE: %.4f" % training_loss)
                iters.append(current_iter)
                training_losses.append(training_loss)
            current_iter += 1
        elif lossType == "CE":
            grad_W, grad_b = gradCE(W_new, b_new, x, y, reg)
            W_new -= alpha*grad_W
            b_new -= alpha*grad_b
            if current_iter % 100 == 0:
                training_loss = crossEntropyLoss(W_new, b, x, y, reg)
                print("Current Iteration: %d" % current_iter)
                print("Average CE: %.4f" % training_loss)
                iters.append(current_iter)
                training_losses.append(training_loss)
            current_iter += 1

    return W_new, b_new, iters, training_losses
```

3. **Tuning the Learning Rate[3 pts]:**
   The plot should look something like the one below.



4. **Generalization [3 pts]:**

λ	Validation Accuracy	Test Accuracy
0.001	0.7400	0.8000
0.01	0.8300	0.8690
0.1	0.9700	0.9655

5. **Comparing Batch GD with normal equation [2 pts]:**

```
def normal():
   normal_W = np.dot(np.dot(np.linalg.inv(np.dot(np.transpose(trainData),
       trainData)), np.transpose(trainData)), np.transpose(trainTarget))
    t_prediction = np.dot(np.transpose(normal_W), np.transpose(trainData)) + b
    t_prediction = np.where(t_prediction > 0.5, 1, 0)
    print("Normal Equation MSE: %.4f" % np.mean((np.equal(t_prediction,
        trainTarget))))
```

Accuracy should be 98.43%. For small datasets, analytical solution is possible, but computing matrix inverse for larger datasets is infeasible.

2 Logistic Regression [10 points]

2.1 Binary cross-entropy loss

The MSE loss function works well for typical regression tasks in which the model outputs are real values. Despite its simplicity, MSE can be overly sensitive to mislabelled training examples and to outliers. A more suitable loss function for the classification task is the cross-entropy loss, which compares the log-odds of the data belonging to either of the classes. Because we only care about the probability of a data point belonging to one class, the real-valued linear prediction is first fed into a sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ that "squashes" the real-valued output to fall between zero and one. The model output is therefore $\hat{y}(\mathbf{x}) = \sigma(W^T\mathbf{x} + b)$. The cross-entropy loss is defined as:

$$\begin{aligned}\mathcal{L} =& \mathcal{L}_{\mathcal{D}} + \mathcal{L}_W \\ =& \sum_{n=1}^{N} \frac{1}{N}\Big[- y^{(n)} \log \hat{y}(\mathbf{x}^{(n)}) - (1 - y^{(n)}) \log(1 - \hat{y}(\mathbf{x}^{(n)}))\Big] + \frac{\lambda}{2}\|W\|_2^2\end{aligned}$$

The sigmoid function is often called the logistic function and hence a linear model with the cross-entropy loss is named "logistic regression".

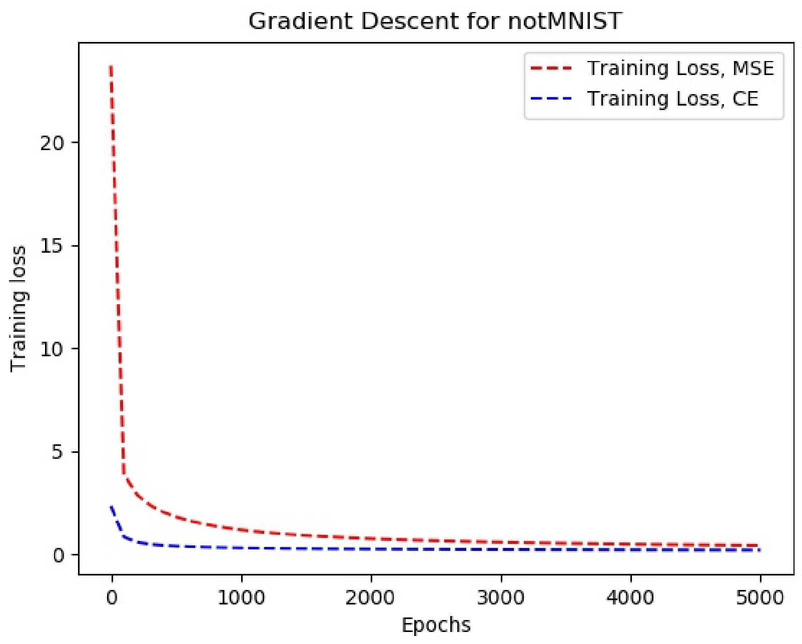
1. **Loss Function and Gradient** [4 pts]:

```
#Helper function
def sigmoid(x):
    return (1/(1+np.exp(-x)))

def crossEntropyLoss(W, b, x, y, reg):
    x = np.transpose(x)
    pred = sigmoid(np.dot(np.transpose(W), x) + b)
    return np.squeeze(np.add(np.mean((-y*np.log(pred)-(1-y)*np.log(1-pred))),
        np.dot(np.transpose(W), W)*reg/2), axis=0)
```

```
def gradCE(W, b, x, y, reg):

    x = np.transpose(x)
    pred = sigmoid(np.dot(np.transpose(W), x)+b)
    grad_W = np.mean(np.multiply(pred-y, x), axis=1)
    grad_W = np.expand_dims(grad_W, axis=1)
    grad_W += 2*reg*W
    grad_b = np.mean(pred-y)
    grad_b = np.expand_dims(grad_b, axis=1)
    return grad_W, grad_b
```

2. **Learning [4 pts]:**
   See above

3. **Comparison to Linear Regression [2 pts]:**
   Plot of MSE and CE vs. Epochs is below. Overall, cross-entropy trains much faster over



   linear regression.

3 Batch Gradient Descent vs. SGD and Adam [25 points]

3.1 SGD

In the exercises above, you implemented the Batch Gradient Descent Algorithm. For large datasets however, obtaining the gradient for *all* of the training data may be infeasible. Stochastic Gradient Descent, or Mini-batch gradient descent is aimed at solving this problem. You will be implementing the SGD algorithm and optimizing the training process using the Adaptive Moment Estimation technique (Adam), using Tensorflow.

1. **Building the Computational Graph** [5 pts]:

```
def buildGraph():
    tf.set_random_seed(421)
    W = tf.Variable(tf.truncated_normal(shape=[28*28,1], stddev=0.5),
       name='weights')
    b = tf.Variable(0.0, name='biases')
    X = tf.placeholder(tf.float32, [None, 28, 28], name='input_x')
    X_flatten = tf.reshape(X, [-1, 28*28])
    y_target = tf.placeholder(tf.float32, [None,1], name='target_y')
    Lambda = tf.placeholder("float32", name='Lambda')

    # Graph definition
    y_predicted = tf.matmul(X_flatten, W) + b

    # Error definition
    meanSquaredError = tf.losses.mean_squared_error(predictions=y_predicted,
       labels=y_target)
    weight_loss = tf.nn.l2_loss(W)*Lambda
    loss = meanSquaredError + weight_loss
    # Training mechanism
    optimizer = tf.train.AdamOptimizer(learning_rate= 0.001)
    train = optimizer.minimize(loss=loss)
    return W, b, X, y_target, y_predicted, meanSquaredError, train, Lambda
```

2. **Implementing Stochastic Gradient Descent** [5 pts.]

```
def stochasticGD(batch_size):
def stochasticGD(batch_size, epochs, lossfilename, accfilename, lossType=None):
    if lossType == None:
        W, b, X, y_target, y_predicted, meanSquaredError, train, Lambda =
            buildGraph(lossType="MSE")
    elif lossType == "CE":
        W, b, X, y_target, y_predicted, meanSquaredError, train, Lambda =
            buildGraph(lossType="CE")
    # Initialize session
    init = tf.global_variables_initializer()
    sess = tf.InteractiveSession()
    sess.run(init)
    trainData, validData, testData, trainTarget, validTarget, testTarget =
       loadData()
    print(trainData.shape)

    ## Training hyper-parameters
    B = batch_size
    wd_lambda = 0
```

```
wList = []
trainLoss_list = []
trainAcc_list = []
validLoss_list = []
validAcc_list = []
testLoss_list = []
testAcc_list = []
epochs = epochs
current_epoch = 0
i = 0
while current_epoch <= epochs:
    if (i*B) % trainData.shape[0] == 0 and i != 0:
        i = 0
        randIdx = np.arange(len(trainData))
        np.random.shuffle(randIdx)
        trainData = trainData[randIdx]
        trainTarget = trainTarget[randIdx]

        err = meanSquaredError.eval(feed_dict={X: trainData, y_target:
            trainTarget, Lambda: wd_lambda})
        acc = np.mean((y_predicted.eval(feed_dict={X: trainData}) > 0.5) ==
            trainTarget)
        trainLoss_list.append(err)
        trainAcc_list.append(acc)

        err = meanSquaredError.eval(feed_dict={X: validData, y_target:
            validTarget, Lambda: wd_lambda})
        acc = np.mean((y_predicted.eval(feed_dict={X: validData}) > 0.5) ==
            validTarget)
        validLoss_list.append(err)
        validAcc_list.append(acc)

        err = meanSquaredError.eval(feed_dict={X: testData, y_target:
            testTarget, Lambda: wd_lambda})
        acc = np.mean((y_predicted.eval(feed_dict={X: testData}) > 0.5) ==
            testTarget)
        testLoss_list.append(err)
        testAcc_list.append(acc)
        current_epoch += 1

    feeddict = {X: trainData[i * B:(i + 1) * B], y_target: trainTarget[i *
        B:(i + 1) * B], Lambda: wd_lambda}
    ## Update model parameters
    sess.run([train, meanSquaredError, W, b, y_predicted], feed_dict=feeddict)
    i += 1

validation_dict = {"train":(trainData, trainTarget), "valid":(validData,
    validTarget), "test":(testData, testTarget)}
```

```
for dataset in validation_dict:
    data, target = validation_dict[dataset]
    err = sess.run(meanSquaredError, feed_dict={X: data, y_target: target,
        Lambda: wd_lambda})
    acc = np.mean((y_predicted.eval(feed_dict={X: data}) > 0.5) == target)
    print("Final %s MSE: %.3f, acc: %.3f"%(dataset, err, acc))

plt.xlabel("Epochs")
if lossType == None:
    plt.ylabel("L2 Regularized MSE Loss")
elif lossType == "CE":
    plt.ylabel("L2 Regularized Mean CE Loss")
plt.title("Stochastic Gradient Descent for notMNIST")
plt.plot(trainLoss_list, 'r--', label="Training Loss")
plt.plot(validLoss_list, 'g--', label="Validation Loss")
plt.plot(testLoss_list, 'y--', label="Test Loss")
plt.legend()
plt.savefig(lossfilename)
plt.clf()
plt.xlabel("Epochs")
plt.ylabel("Classification Accuracy")
plt.title("Stochastic Gradient Descent for notMNIST")
plt.plot(trainAcc_list, 'r--', label="Training Accuracy")
plt.plot(validAcc_list, 'g--', label="Validation Accuracy")
plt.plot(testAcc_list, 'y--', label="Test Accuracy")
plt.legend()
plt.savefig(accfilename)
sess.close()
```

3. **Batch Size Investigation [2 pts.]** For MSE, plots are below.

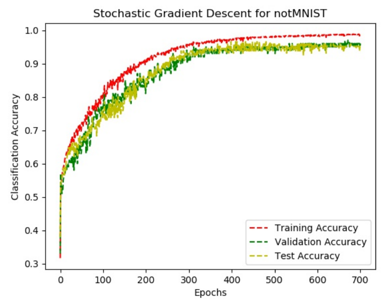


Figure 1: MSE Accuracy for $B = 100$

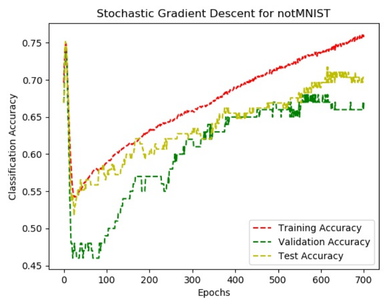


Figure 2: MSE Accuracy for $B = 700$

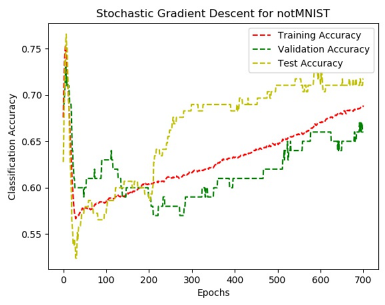


Figure 3: MSE Accuracy for $B = 1750$

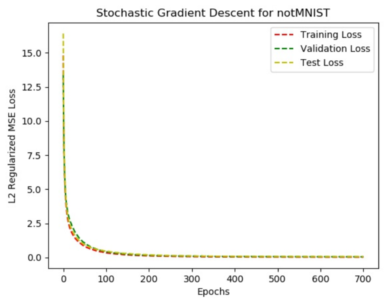


Figure 4: MSE Loss Curve for $B = 100$

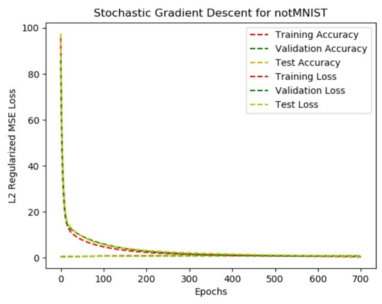


Figure 5: MSE Loss Curve for $B = 700$

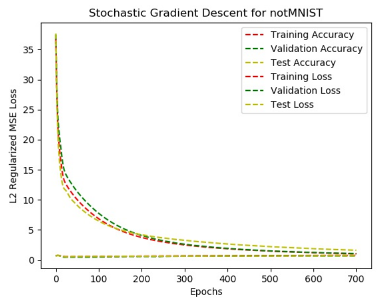


Figure 6: MSE Loss Curve for $B = 1750$

Plots for CE are below.

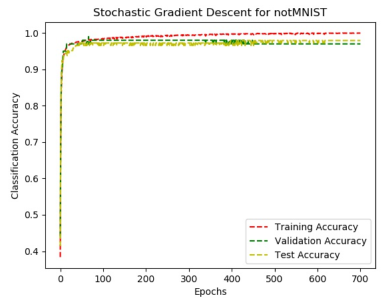


Figure 7: CE Accuracy Curve for $B = 100$

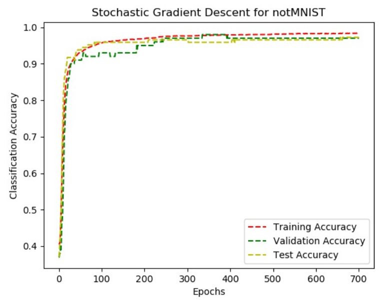


Figure 8: CE Accuracy Curve for $B = 700$

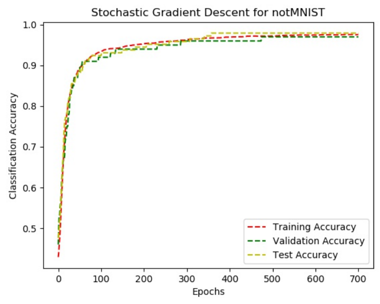


Figure 9: CE Accuracy Curve for $B = 1750$

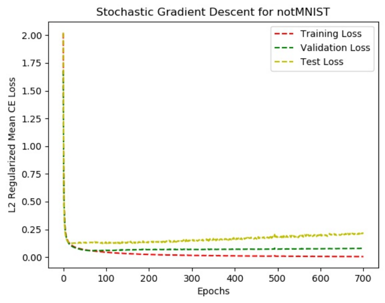


Figure 10: CE Loss Curve for $B = 100$

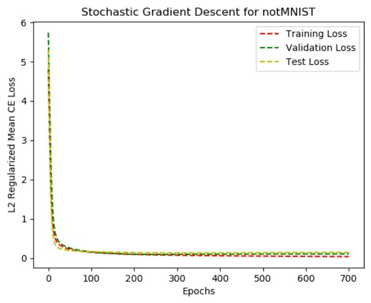


Figure 11: CE Loss Curve for $B = 700$

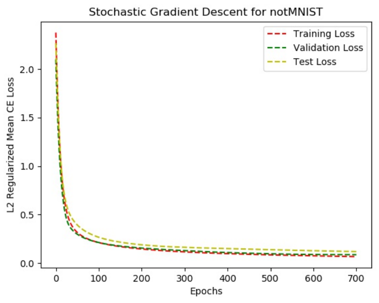


Figure 12: CE Loss Curve for $B = 1750$

4. **Hyperparameter Investigation [4 pts.]** For MSE:

Accuracy	$\beta_1 = 0.95$	$\beta_1 = 0.99$	$\beta_2 = 0.99$	$\beta_2 = 0.9999$	$\epsilon = 10^{-9}$	$\epsilon = 10^{-4}$
Training	0.906	0.806	0.953	0.834	0.850	0.883
Validation	0.830	0.710	0.890	0.750	0.820	0.860
Test	0.855	0.760	0.910	0.786	0.862	0.828

For CE:

Accuracy	$\beta_1 = 0.95$	$\beta_1 = 0.99$	$\beta_2 = 0.99$	$\beta_2 = 0.9999$	$\epsilon = 10^{-9}$	$\epsilon = 10^{-4}$
Training	0.990	0.995	0.997	0.980	0.992	0.990
Validation	0.960	0.960	0.890	0.987	0.970	0.99
Test	0.972	0.979	0.972	0.986	0.979	0.972

5. **Cross Entropy Loss Investigation [6 pts.]** See above

6. **Comparison against Batch GD [3 pts.]** Comment on the overall performance of the SGD algorithm with Adam vs. the batch gradient descent algorithm you implemented earlier. Additionally, comment on the plots of the losses and accuracies of the SGD vs. batch gradient descent implementation. What do you notice about the curves? Why is this happening?

   SGD + Adam performs significantly better. Noisy gradients due to updates based on a sample of training data and not the entire set. Also due to modifications on optimization hyperparameters from the Adam solver (e.g. how exponential decay is handled for example)