Lab 3 Report
HPC - DL

By Shivani Patel

Exercise 1 - Optimization problem

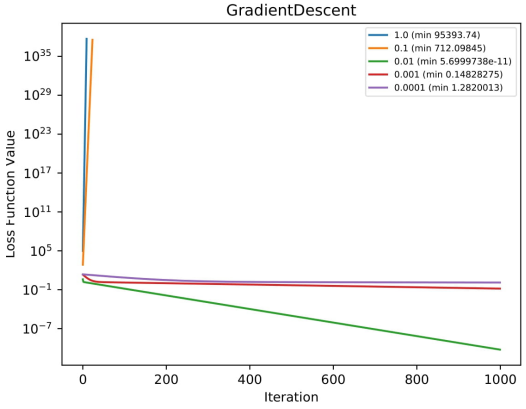
For the given optimization problem, we tried the following ones available in TF1, along with varying LRs on the log scale from 1 to 0.0001 and also tried tweaking the hyperparameters available for that optimizer around the given defaults (if any):

#	Optimizer	Tweakable hyperparams	Significance	Default value	Optimal value after tweaking
1	Gradient Descent (given, baseline)	LR	Learning rate	NA	0.01
2	Adam	LR	Learning rate	0.001	0.1
		beta1	Exponential decay rate for the 1st moment estimates	0.9	0.9
		beta2	Exponential decay rate for the 2nd moment estimates	0.999	0.994
		epsilon	Small constant for numerical stability	1e-8	1e-7
3	RMSprop	LR	Learning rate	NA	0.01
		decay	Discounting factor for the history/coming gradient	0.9	0.9
		momentum	Inertia in the direction in the search space to overcome the oscillations of noisy gradients	0.0	0.1
		epsilon	Small value to avoid zero denominators	1e-10	1e-7
		centred	If True, gradients are normalized by the estimated variance of the gradient. If False, by the uncentered second moment.	False	True
4	Momentum	LR	Learning rate	NA	0.01
		momentum	Inertia in the direction in the search space to overcome the oscillations of noisy gradients	0.9	0.95
5	FTRL	LR	Learning rate	NA	1.0
		LR_power	Controls how the learning rate decreases during training (<=0; 0 for constant LR)	-0.5	-0.4
		initial_accumulator_value	The starting value for accumulators (>=0)	0.1	0.1

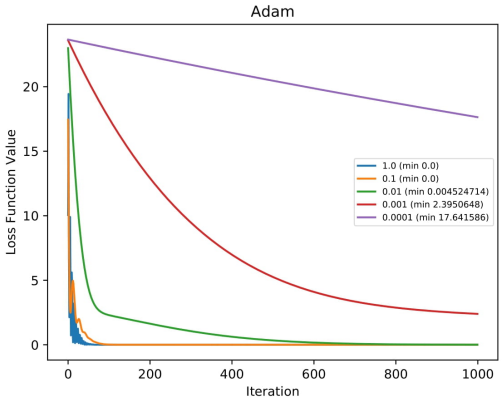
		l1_reg_strength	L1 stabilization penalty	0.0	0.0
		l2_reg_strength	L2 stabilization penalty	0.0	0.15
		l2_shrinkage_regularization_strength	L2 magnitude penalty, must be >=0	0.0	0.0
6	AdaGrad	LR	Learning rate	NA	1.0
		initial_accumulator_value	The starting value for accumulators (>=0)	0.1	0.1
7	AdaDelta	LR	Learning rate	0.001	1.0
		rho	Decay rate	0.95	0.95
		epsilon	Constant used to better conditioning the gradient update	1e-8	1e-7
8	Proximal Gradient Descent	LR	Learning rate	NA	0.01
		l1_reg_strength	L1 stabilization penalty	0.0	0.0
		l2_reg_strength	L2 stabilization penalty	0.0	0.15
9	Proximal AdaGrad	LR	Learning rate	NA	1.0
		initial_accumulator_value	The starting value for accumulators (>=0)	0.1	0.1
		l1_reg_strength	L1 stabilization penalty	0.0	0.0
		l2_reg_strength	L2 stabilization penalty	0.0	0.2

The loss graphs obtained were as follows - for this optimization problem, every loss function converged but this may not be the case for other problems (as we'll see in MNIST later):

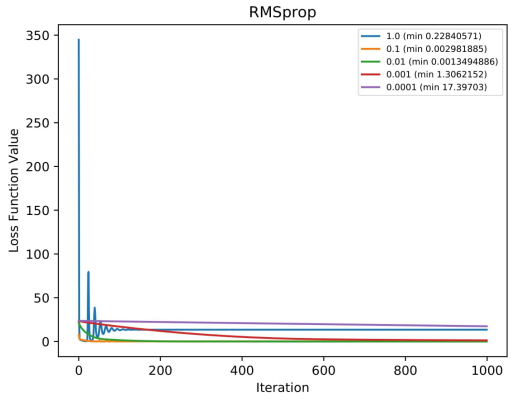
1. Gradient Descent



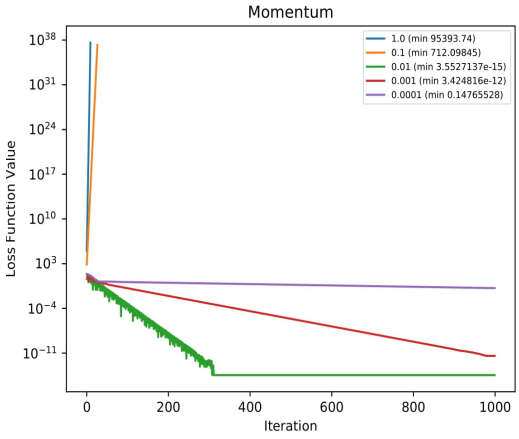
2. Adam



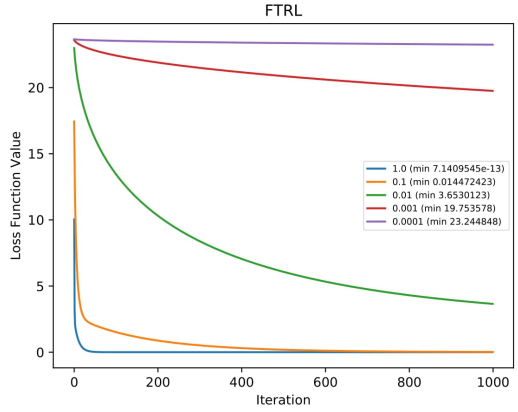
3. RMSprop



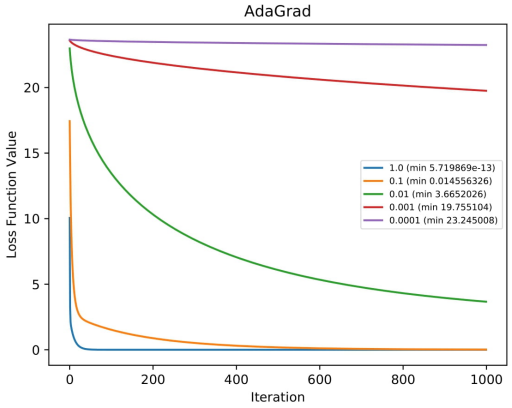
4. Momentum



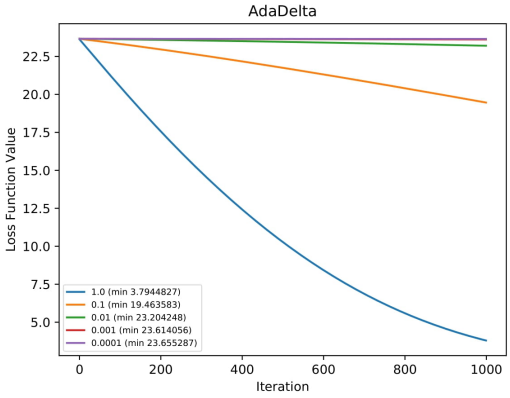
5. FTRL



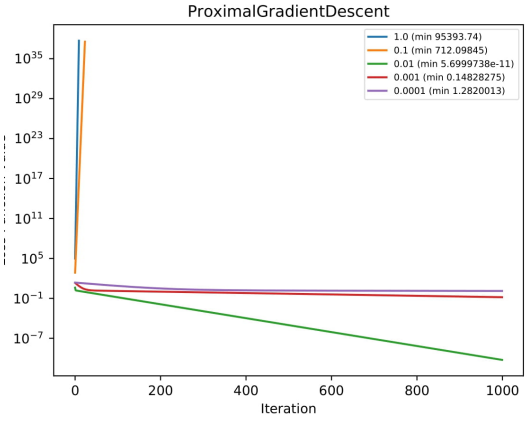
6. AdaGrad



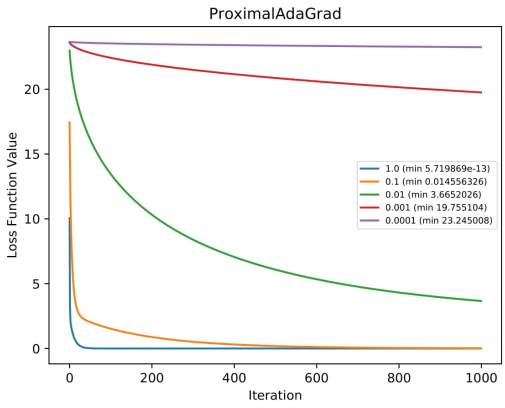
7. AdaDelta



8. Proximal Gradient Descent



9. Proximal AdaGrad



Accuracies across the optimizers when the LR is kept fixed at 0.01 and other hyperparams set at their default value, we obtained the following results:

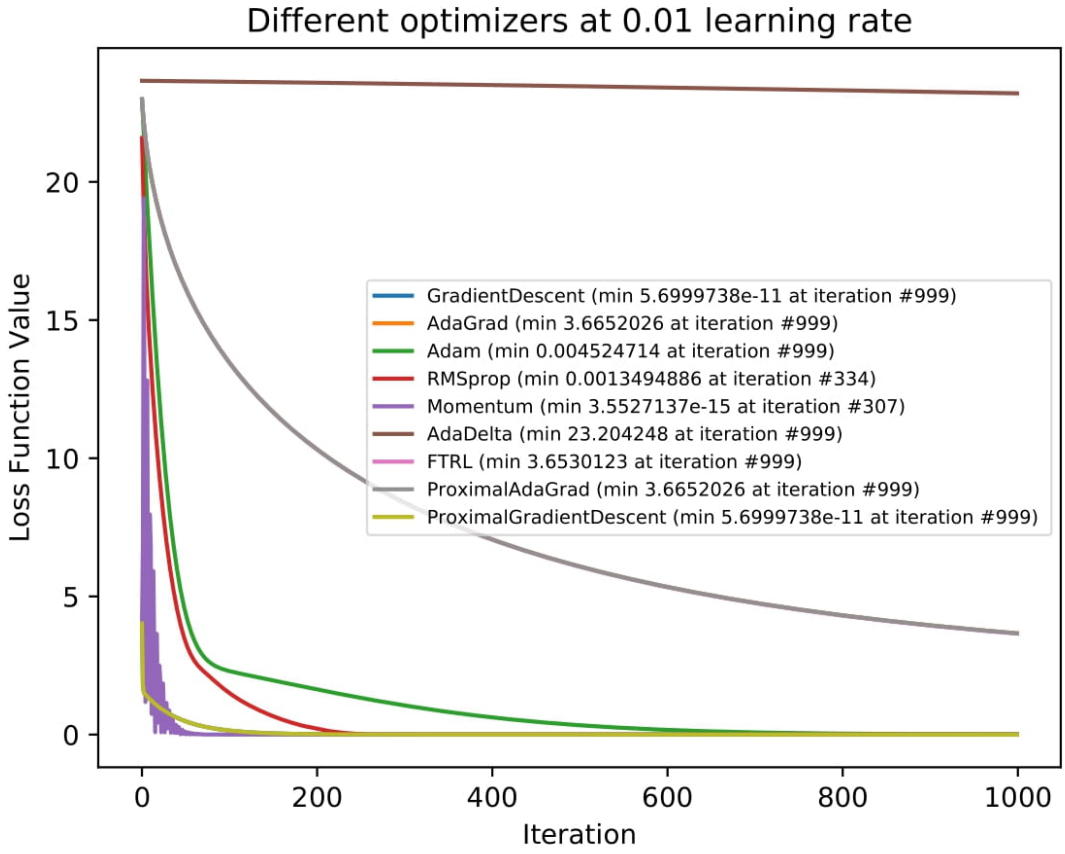


Fig. Loss function plots for various optimizers with LR = 0.01 and default hyperparameters

The overall best performance was in the case of Gradient Descent and Proximal GD at LR 0.01. However, their initial losses were quite high (hence their individual graphs in the log scale for the y-axis). Close second best and best for most cases was RMSprop with a loss of 0.001349 after 330 iterations.

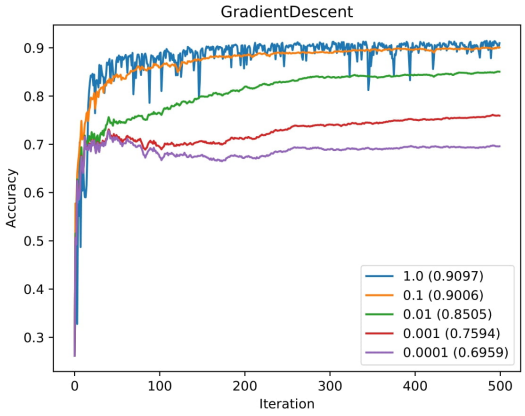
Exercise 2 - Single-layer Network on MNIST

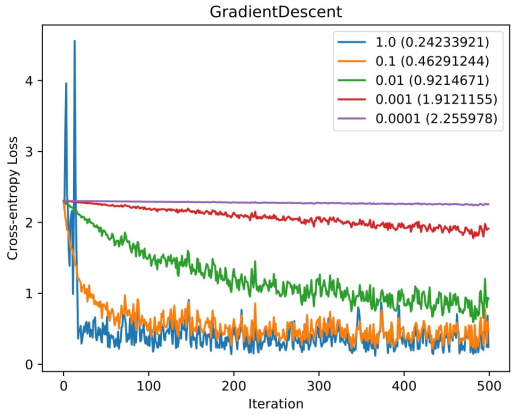
For the given single-layer model for MNIST, we tried with the above optimizers as in Ex1 along with LRs on a log scale from 1.0 to 0.0001, and we got the following results:

Note: Some of the hyperparams provided in the bracket were set differently from the default values after experimenting in Ex1 and finding the best values.

1. Gradient Descent

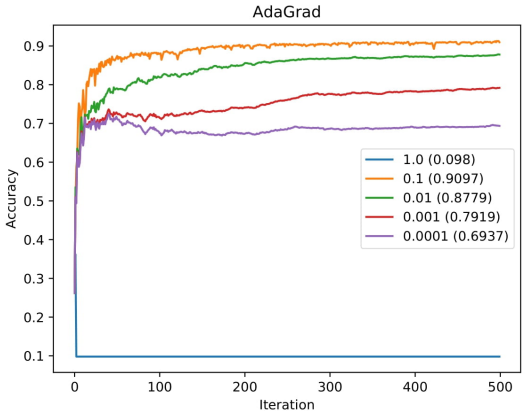
Converged in all cases as expected.

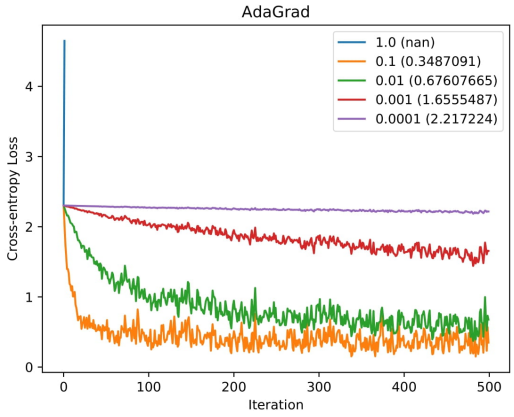




2. AdaGrad (initial_accumulator_value = 0.1)

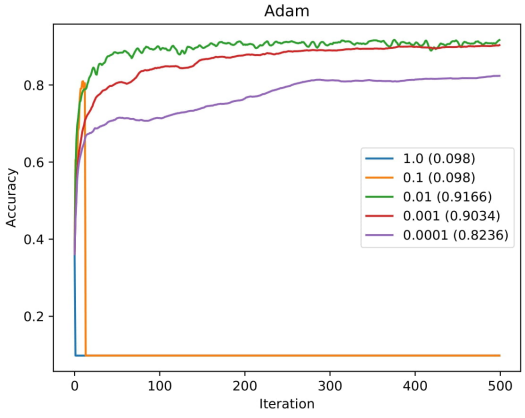
Doesn't converge with LR=1.0, best aggregated performance with LR=0.1. Best trained model at around iteration # 200.

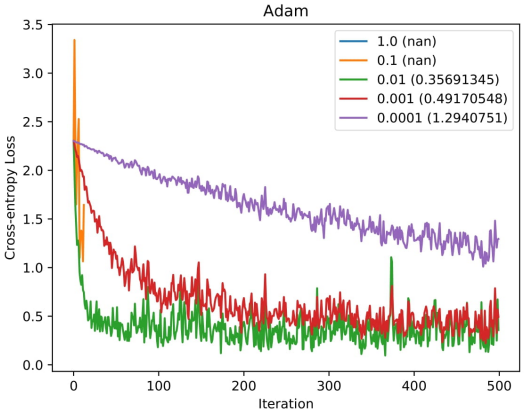




3. Adam (beta1 = 0.9, beta2 = 0.999, epsilon = 1e-7)

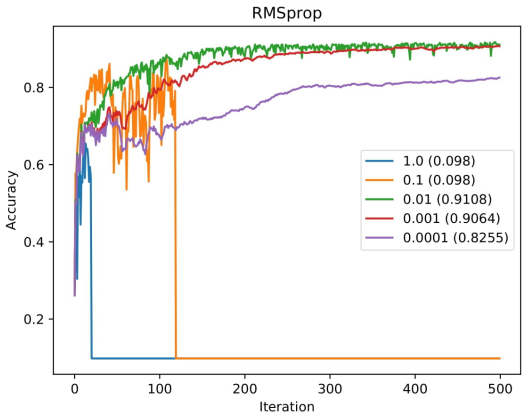
Doesn't converge at higher LRs (1.0 and 0.1). Best performance at 0.01, just before iteration # 200.

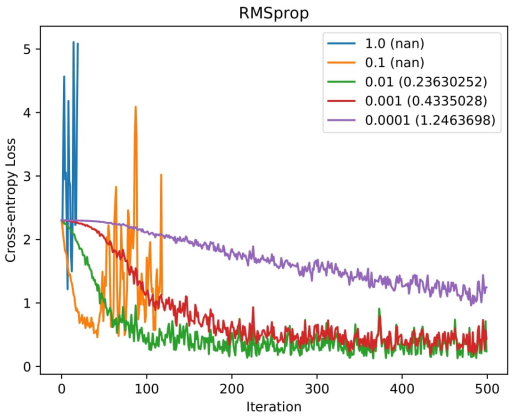




4. **RMSprop (decay = 0.9, momentum = 0.1, epsilon = 1e-7, centered = False)**

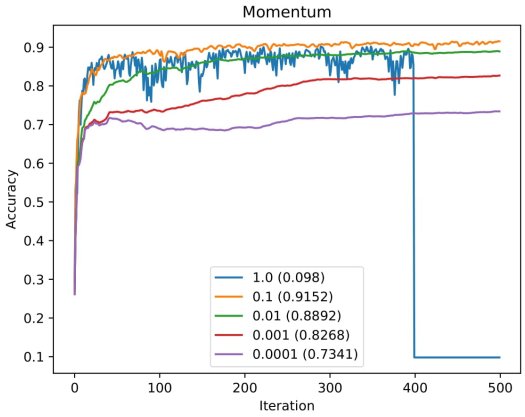
   Doesn't converge at higher LRs (1.0 and 0.1). Best performance at 0.01, around iteration # 225.

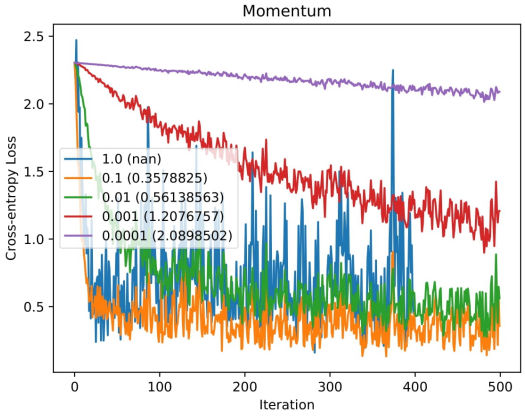




5. **Momentum (momentum = 0.95)**

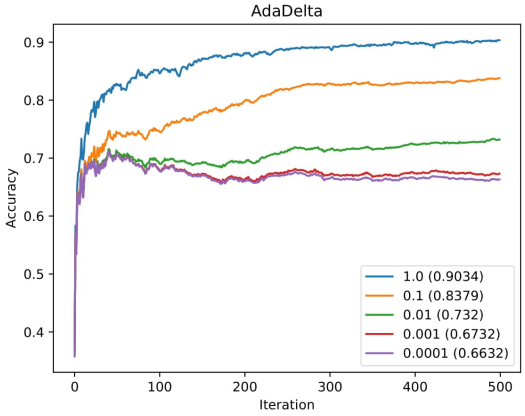
   Doesn't converge at a higher LR of 1.0, unstable training for lower LRs. Best performance at 0.1, just around iteration # 200.

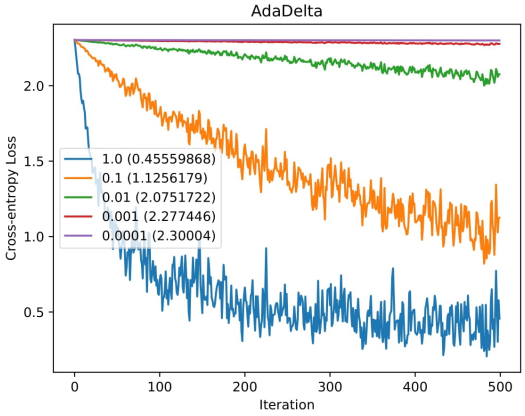




6. **AdaDelta (rho = 0.95, epsilon = 1e-7)**

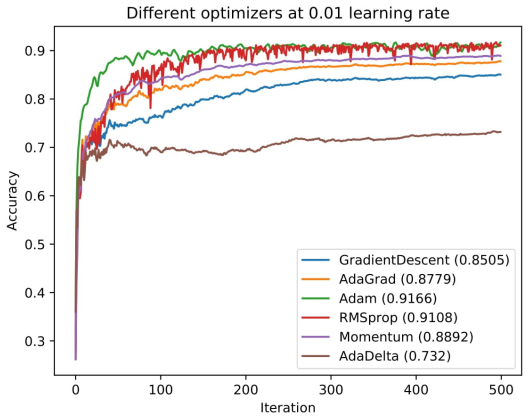
   Converges in all cases. Best performance at 1.0 but unstable training. Model underfitting even after 500 iterations.

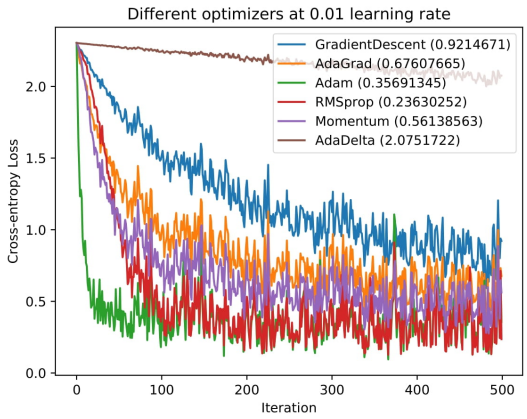




In almost all cases, we see stable but very slow convergence for lower LRs (0.001 & 0.0001).

For the same **LR of 0.01**, the results are:





Both Adam and RMSprop were neck-to-neck wrt performance - accuracy, losses as well as wrt training time, however, RMSprop was marginally better by the end of 500 iterations. Although using Early Stopping, we could've gotten a better model around iteration # 200 itself.

Note: Experimental optimizers like FTRL, Proximal AdaGrad, and Proximal Gradient Descent from Ex1 are removed in this exercise due to device assignment errors during Variable/read.

Exercise 3 - Multi-layer Network on MNIST

We started by listing all the available optimizers against varying learning rates and got the accuracy for every combination, running training for 1 epoch for now.

Choosing the highest accuracy (similar to Ex2), we got RMSprop at LR=0.001 to be the best performer in terms of accuracy. Taking this, we tweaked the following:

1. **Increase epochs for training**
2. **Batch selection for every epoch:** Choosing the batches across epochs from the same subset of data, size of the said subset & hence the number of batches used for training per epoch, and whether batches could be overlapping or not.
3. **Batch size for every epoch:** With 50 being the baseline in the given code, we increased and decreased the batch size, and also made it dynamic wrt the epochs.
4. **Loss functions:** For the loss functions, we primarily used various cross-entropy-based loss functions, since this is a single-label multiclass classification problem. Most of these used one-hot encoded values as the labels with the exception of sparse softmax cross-entropy (v10 and v20).
    a. **Softmax Cross Entropy with Logits -** It measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each instance is exactly one class).
    b. **Sparse Softmax Cross Entropy with Logits -** Similar to above but when the activations are sparse.
    c. **Weighted Cross Entropy with Logits -** Similar to Sigmoid CE with Logits except that *pos_weight* allows a trade-off between recall and precision by up- or down-weighting the cost of a positive error relative to a negative error.
        i. *pos_weight* > 1 decreases false negative count, increasing recall.
        ii. *pos_weight* < 1 decreases false positive count, increasing precision. In our case with MNIST, increasing precision gave better results.

d. **Sigmoid Focal Cross Entropy -** It down-weights well-classified examples and focuses on hard examples, and hence is extremely useful for classification when you have highly imbalanced classes as the loss value is much higher for a sample that is misclassified. The TF version also has the option to use it with logits or not - with the *logits=True* performing far better in the case of MNIST. Alpha is the balancing factor (defaults to 0.25).
e. **Sigmoid Cross Entropy with Logits -** It measures the probability error in tasks with two outcomes in which each outcome is independent.
f. **Weighted Kappa Loss** - It is widely used in multi-class classification problems using ordinal data, however unfortunately it ended up performing the worst, proving the hypothesis that it is not right for the current use case - where any misclassification cannot be "more wrong" than another misclassification (i.e. misclassifying "1" as "2" is *as wrong as* misclassifying "1" as "7")

Below are the versions of Ex3 code we ran with the above modifications while keeping the given architecture intact:

exp_ no	batch size	batch selection from (shuffled)	loss function	loss function hyperparams
1	50, fixed, no overlap	subset of 1k	softmax cross entropy with logits	-
2	100, fixed, with overlap	subset of 1k	softmax cross entropy with logits	-
3	100, fixed, no overlap	subset of 1k	softmax cross entropy with logits	-
4	25, fixed, no overlap	subset of 1k	softmax cross entropy with logits	-
5	50, fixed, no overlap	subset of 5k	softmax cross entropy with logits	-
6	50, fixed, no overlap	subset of 500	softmax cross entropy with logits	-
7	50, fixed, no overlap	subset of 2k	softmax cross entropy with logits	-
9	32, fixed, no overlap	subset of 1k	softmax cross entropy with logits	-
10	50, fixed, no overlap	subset of 1k	sparse softmax cross entropy with logits	-
11	50, fixed, no overlap	subset of 1k	weighted cross entropy with logits	pos_weight=1.5
12	50, fixed, no overlap	subset of 1k	weighted cross entropy with logits	pos_weight=0.5
14	50, fixed, no overlap	subset of 1k	weighted Kappa loss	-
15	50, fixed, no overlap	subset of 1k	sigmoid focal cross entropy	logits=True, alpha=0.5
16	50, fixed, no overlap	subset of 1k	sigmoid cross entropy with logits	-
17	50, variable on epoch #, with overlap	subset of 1k	sigmoid cross entropy with logits	-
18	variable on epoch, no overlap	subset of 1k	sigmoid cross entropy with logits	-
19	variable on epoch, no overlap	subset of 1k	softmax cross entropy with logits	-
20	variable on epoch, no overlap	subset of 1k	sparse softmax cross entropy with logits	-

Furthermore, we also plotted accuracy and loss graphs for every version. Notably, the highest accuracy was obtained with v18 (sigmoid cross-entropy, batch size variable on epoch, and no overlap between batches) at 99.4% at epoch #14, however, the lowest loss

was in the case of v12 (weighted cross-entropy with logits, batch size fixed at 50 with no overlap between batches) at 0.003749 at epoch #3.

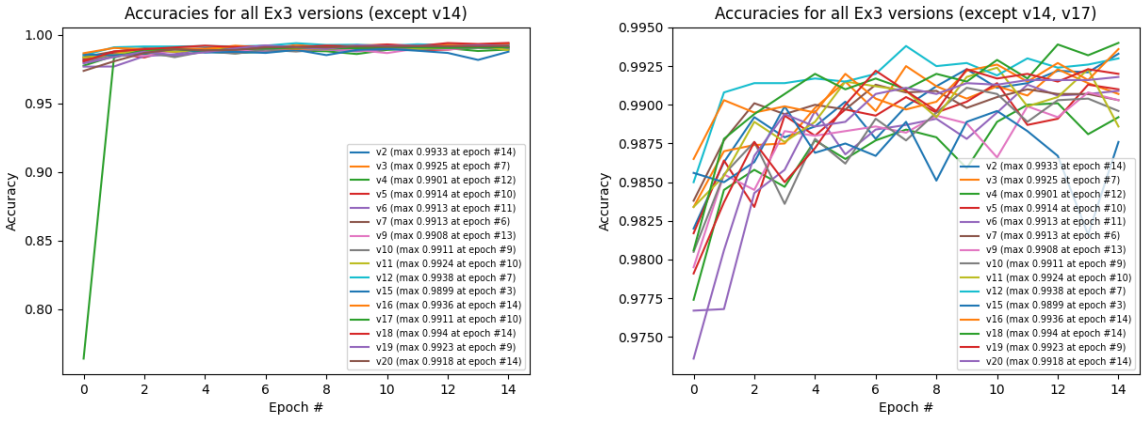


Fig. Second subplot showing accuracies excluding v17 for better distinguishable plots

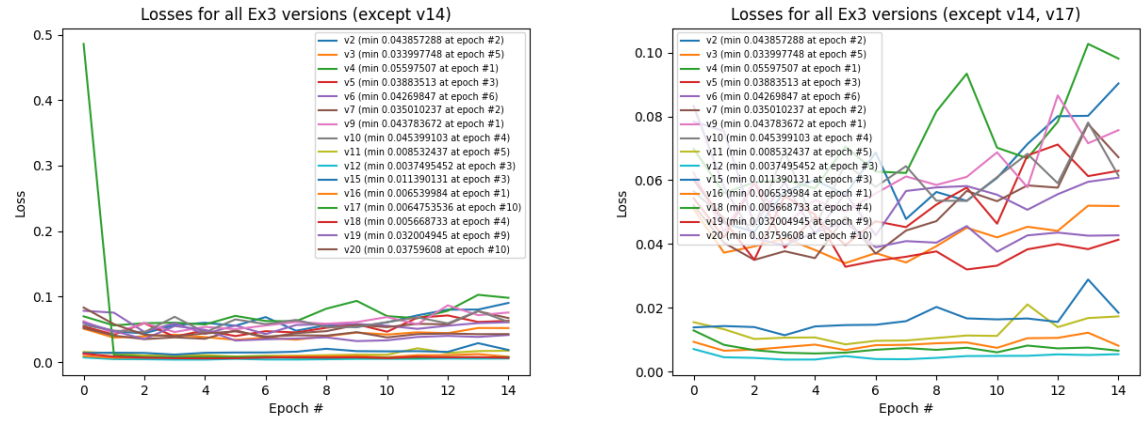


Fig. Second subplot showing losses excluding v17 for better distinguishable plots

Exercise 4 - Multi-layer Network on MNIST with multiple GPUs

We chose the version of Ex3 with the highest accuracy as the base model for Ex4 and ran it with 2 & 4 GPUs. However, the multi-GPU training did not give any significant improvement in training times. This could be attributed to the following:

1. The model is fairly small with few parameters. The batch size is also far too small to be requiring more GPUs. For a small-scale model like this, it barely can saturate the 40 CPUs per GPU in the single GPU setting - much less take advantage of any additional computing power we provide.
2. The overhead of maintaining (copying & updating) parameters of the model across multiple GPUs does not outweigh the improvements obtained (which, in fact, are negligible). Hence the degradation in training time.
    a. Especially the dense layer towards the end has a lot of non-shared parameters, which could be the result of the bulk of this bottleneck.

Plotting a graph for training time per epoch, across epochs, for the corresponding batch sizes, we see below that the difference becomes comparable (especially for the 1 and 2 GPUs) with an increase in batch size - which in this case, is essentially the only parameter we can tweak to increase/saturate the computational complexity since the model architecture is fixed.

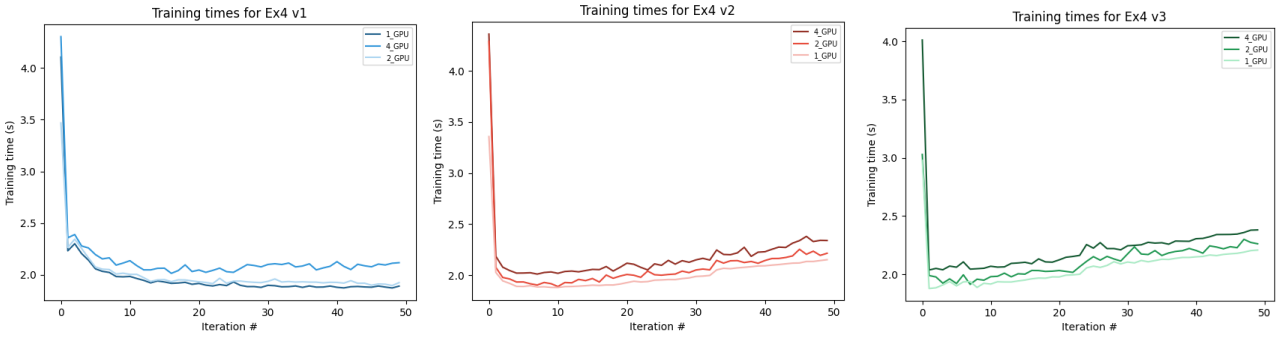


Fig. Training time plots for each version, in a single GPU, 2 GPU, and 4 GPU setting

Therefore, to actually see the difference and upgrade a multi-GPU setting brings, we tried to increase the batch size. The difference was still not evident as the batch size was still not large enough to require such computing power, but at least it became comparable to the single GPU versions.

Model	Batch size
v1	(30 + (epoch * 10))
v2	(50 + (epoch * 50))
v3	(500 + (epoch * 50))

Using 4 GPUs still gave worse training times in all cases due to the overhead required to update parameters across GPUs. Furthermore, v1 (least batch size) has the most evident training time difference, especially for a single vs. double GPU setting, whereas the difference between 2 & 4 GPUs is fairly small and more or less the same for all versions of training:

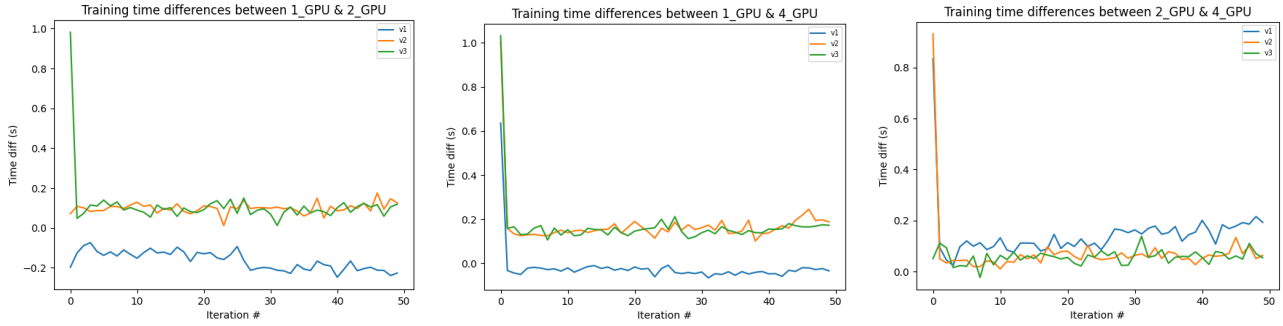


Fig. Training time difference plots for each pair of difference, for model versions v1, v2, v3 (in increasing batch size)