NLP Assignment Writeup

[Sampras M. Dsouza (sd6701)]

[Code link: https://drive.google.com/file/d/1xOPArLSi13ze0EjYHa1xC2NGz82ICpj5/view?usp=sharing]

Implementation Overview

As Part of this Homework, we have worked on Part 1: building a unigram feature extractor and then applying the changes on the Logistic regression model and testing the results. And in Part 2 we worked on Create Deep Average Neural Networks and estimated their performance.

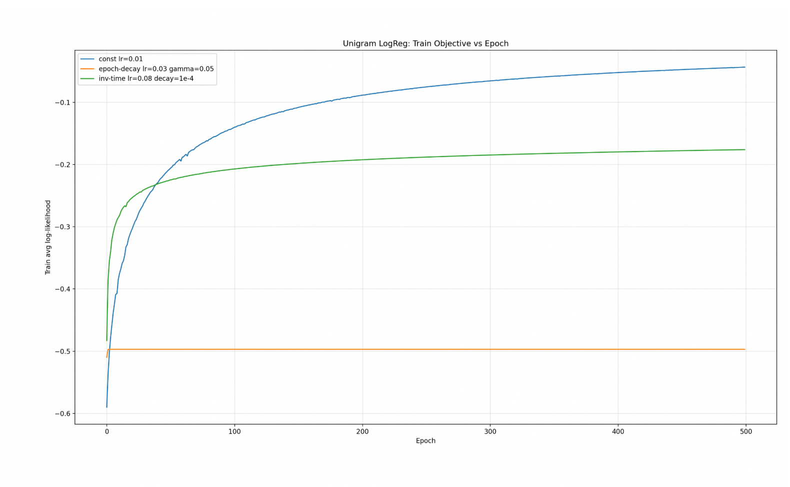
Exploration 1: Learning Rate Schedules

I experimented with multiple learning rate schedules to understand their impact on model training and convergence as I implemented several strategies like Constant Learning Rate , decrease the step size by some factor every epoch and Inverse time (decrease it by like 1/ t)

Figure 1: Unigram LR Training Objective vs Dev Accuracy for Different types of learning rate Mechanism

1. Blue Line represents the constant learning rate = 0.01
2. Orange Line represents the decay-lr = 0.03 and rate of decay =0.05
3. Green Line represents the inverse time lr = 0.08 with decay rate = 1e-4

   We can see that since the LR for Orange and Green curve is less the Constant learning rate out perform than the other learning rate



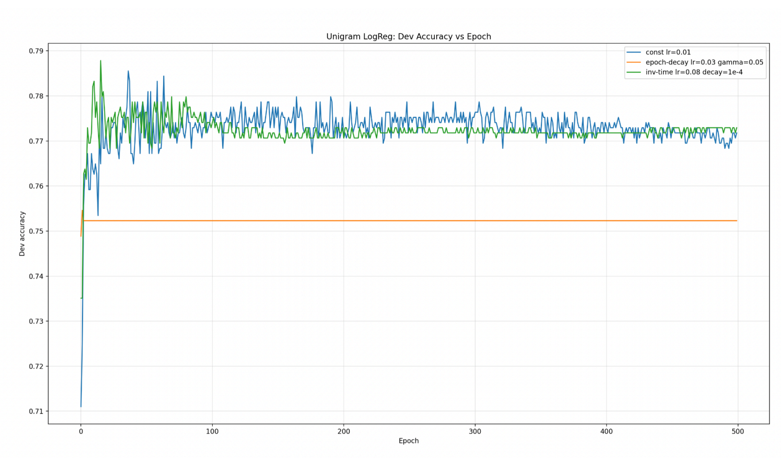
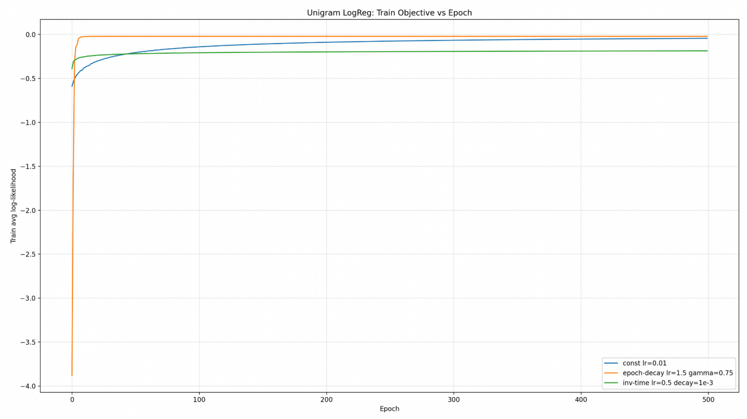
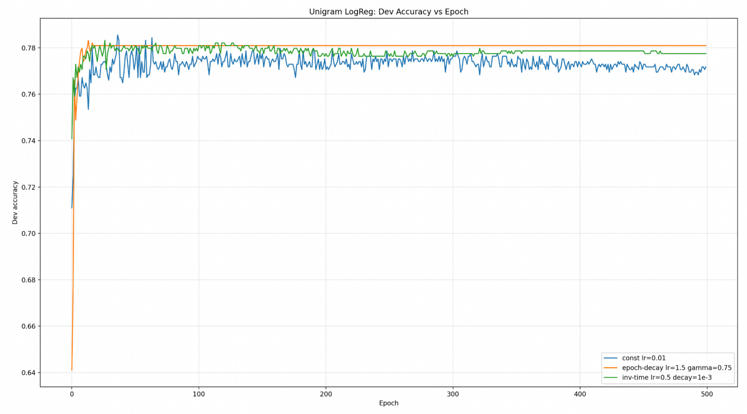


Figure 2: Development Accuracy vs. Iterations

1. Blue Line represents the constant learning rate = 0.01
2. Orange Line represents the decay-lr = 1.3 and rate of decay =0.75
3. Green Line represents the inverse time lr = 0.5 with decay rate = 1e-3

   We can see that after fine tuning the decay lr and inverse time lr it performs better than the constant lr





Learning-rate schedule observations

To study the effect of step size schedules in unigram logistic regression, I compared three update rules a **constant learning rate** (lr = 0.01), an **epoch based decay** schedule (multiplying lr by a factor each epoch), and an **inverse time** schedule (lr decays roughly as 1/*t*1). The above diagram shows both the training objective (average log likelihood) and development accuracy across epochs.

Constant learning rate (lr = 0.01).
With a fixed step size, the training objective improves steadily over time and continues making small gains even at large epoch counts. On the development set, accuracy rises quickly early in training and then fluctuates mildly around a plateau. Although the dev accuracy curve is somewhat noisy, the constant schedule is competitive and in one run achieves the highest dev accuracy among the schedules tested.

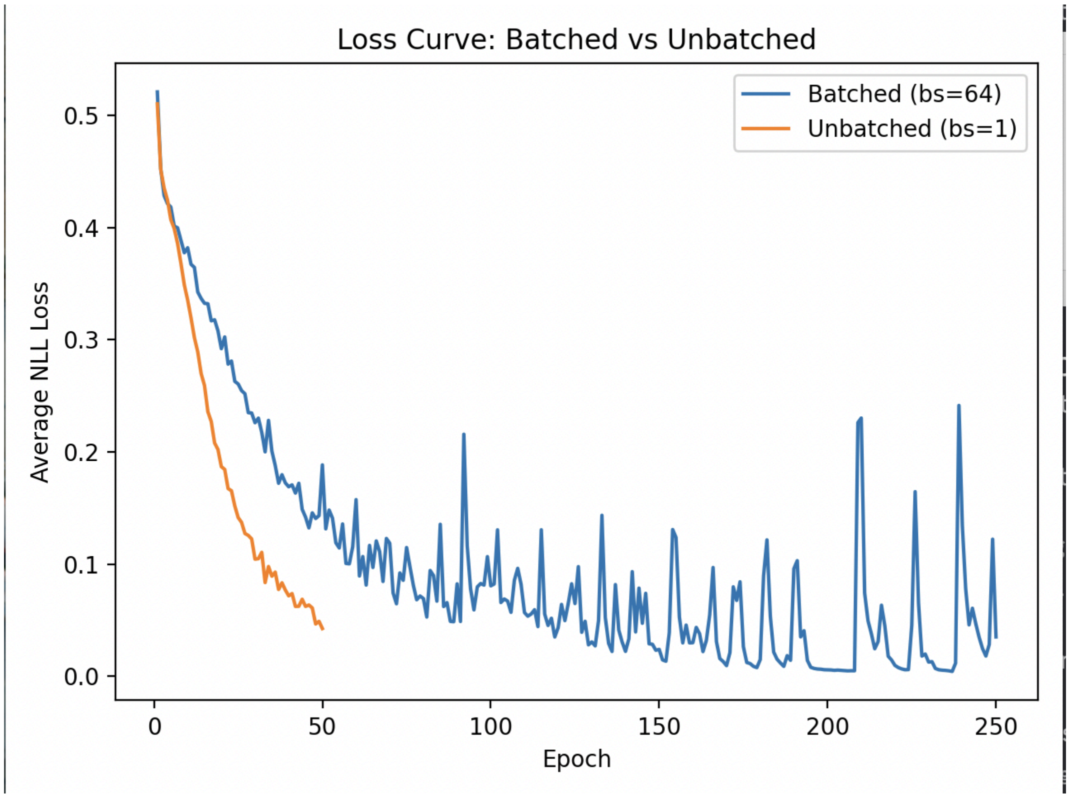
Epoch decay schedule.
The epoch decay schedule shows the fastest early improvement in the training objective, it increases sharply in the first few epochs and then quickly stabilizes once the learning rate becomes small. On the development set, this schedule produces a very stable accuracy curve (less noisy than constant lr). However, performance is sensitive to the decay hyperparameters, in one setting it matches or slightly exceeds the other schedules, while in another setting it plateaus earlier at a slightly lower accuracy, suggesting the learning rate decayed too aggressively and limited further improvement.

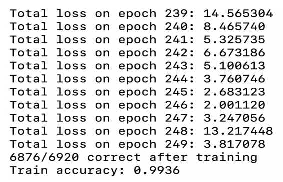
Inverse time schedule.
The inverse time schedule is highly dependent on the initial learning rate. With a sufficiently large initial lr, the method achieves competitive dev accuracy and remains stable as the step size decreases over time. When the initial lr is too small, the training objective improves more slowly and dev accuracy lags behind the constant lr baseline, indicating under updating early in training. Overall, inverse time decay provides a reasonable tradeoff between early learning and late stage stability, but requires careful choice of the initial step size.

Conclusion:
All schedules reach similar dev accuracy ranges, but they differ in convergence behavior and stability. **Constant lr** provides **strong performance** but can be **noisier** on dev accuracy, **epoch decay can converge quickly** and produce stable dev curves but is **sensitive to decay rate,** and i**nverse time decay is stable but requires an adequate initial lr to avoid slow learning**.

Exploration 2: Batching Implementation

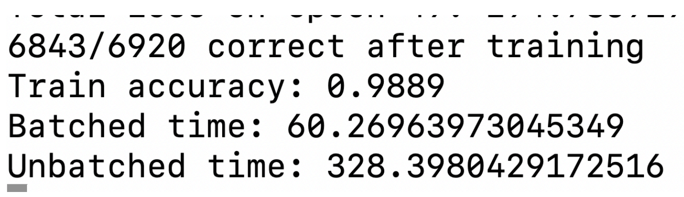
1. For Batch Size of : 64





```
Total loss on epoch 239: 14.565304
Total loss on epoch 240: 8.465740
Total loss on epoch 241: 5.325735
Total loss on epoch 242: 6.673186
Total loss on epoch 243: 5.100613
Total loss on epoch 244: 3.760746
Total loss on epoch 245: 2.683123
Total loss on epoch 246: 2.001120
Total loss on epoch 247: 3.247056
Total loss on epoch 248: 13.217448
Total loss on epoch 249: 3.817078
6876/6920 correct after training
Train accuracy: 0.9936
```

Training Accuracy of Batched Process



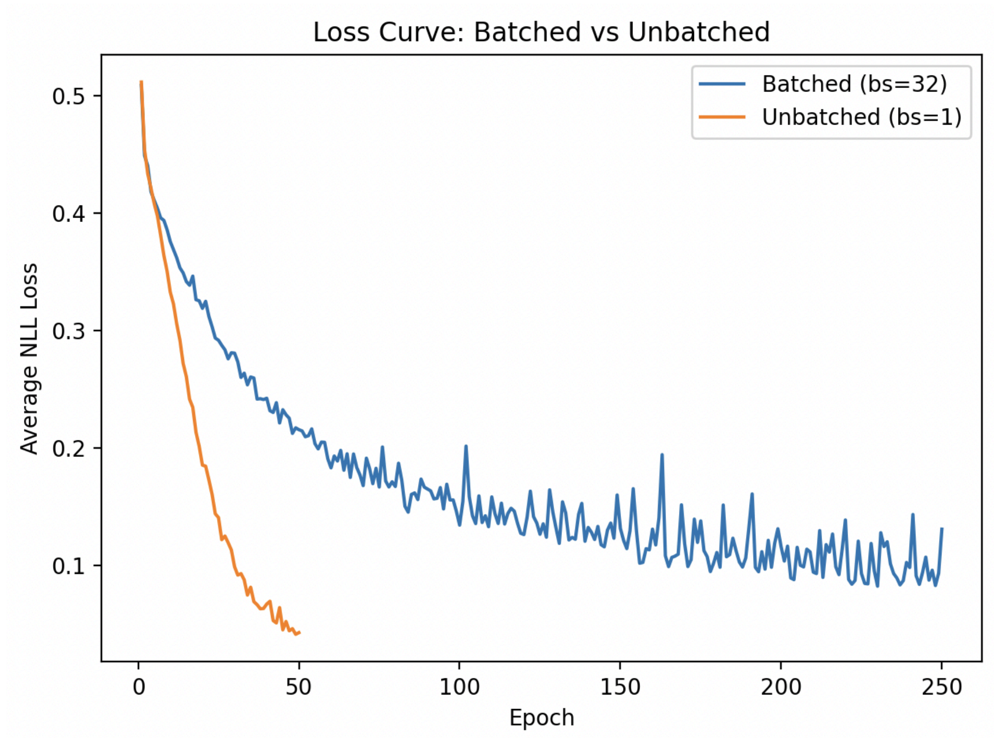
```
6843/6920 correct after training
Train accuracy: 0.9889
Batched time: 60.26963973045349
Unbatched time: 328.3980429172516
```

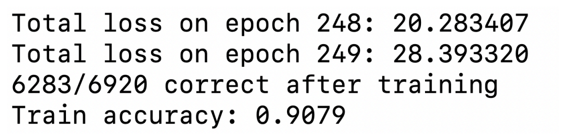
Training Accuracy of Unbatched process and Time required for both the process.

Summary:
The plot compares average NLL loss for the same FFNN trained with batching (bs=64, Adam lr=0.01, 250 epochs) versus unbatched updates (bs=1, Adam lr=0.001, 50 epochs). In both settings the **loss drops quickly at the start,** showing the model learns useful parameters early. The unbatched run decreases smoothly and steadily, reaching a low loss by epoch 50, this **stable behavior** is expected with a smaller learning rate and single example updates. The batched run continues improving over many more epochs and eventually reaches very low loss, but it is noticeably **noisier** and shows **occasional sharp spikes**. This spiky behavior can be expected when using a larger learning rate with mini-batch optimization (and can also be amplified if the plotted values are per-batch or not perfectly averaged per epoch). Overall, **batching provides a faster training setup that can reach lower loss given enough epochs, while the unbatched configuration converges more smoothly but was run for fewer epochs with a smaller learning rate.**

2. Batch Size of 32

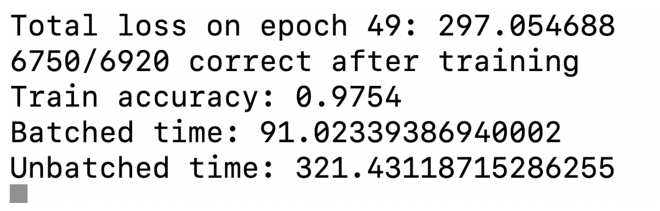
Training Accuracy for batch size of 32





```
Total loss on epoch 248: 20.283407
Total loss on epoch 249: 28.393320
6283/6920 correct after training
Train accuracy: 0.9079
```

Training Accuracy of Batched Process



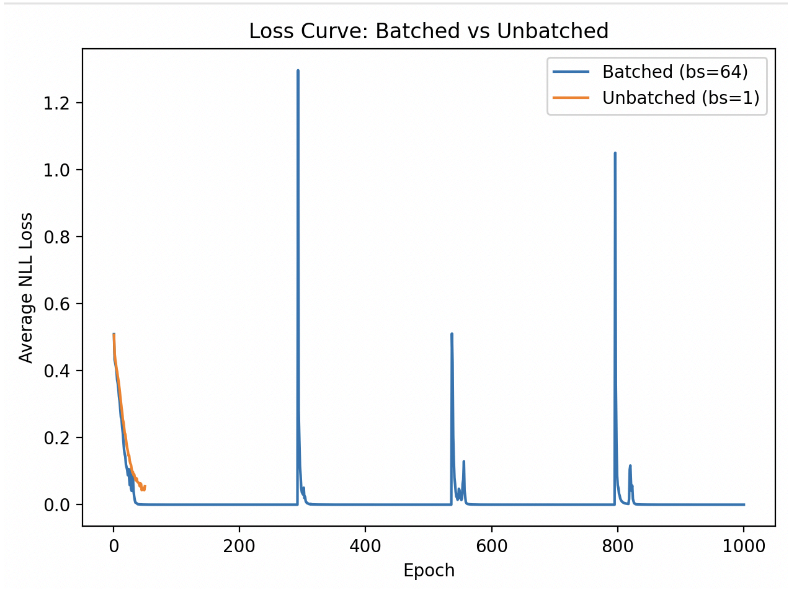
```
Total loss on epoch 49: 297.054688
6750/6920 correct after training
Train accuracy: 0.9754
Batched time: 91.02339386940002
Unbatched time: 321.43118715286255
```

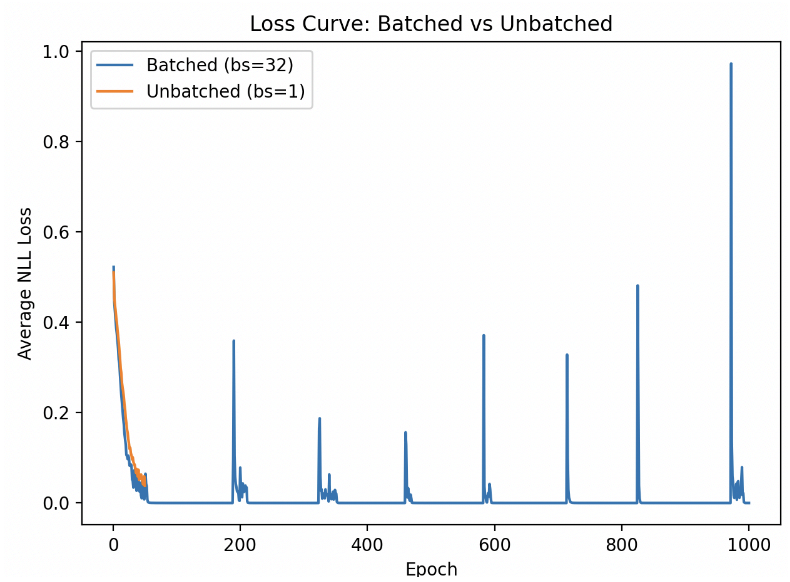
Training Accuracy of Unbatched process and Time required for both the process.

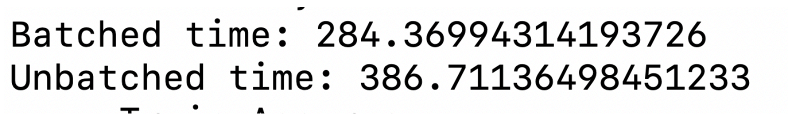
Summary :

This figure compares training loss for mini-batch training (bs=32) vs unbatched training (bs=1). Both runs start around the same NLL (~0.5) and drop quickly in the first few epochs, showing the model learns useful parameters early. The unbatched model (orange) converges much faster in terms of loss over the first 50 epochs, reaching a very low average NLL and a high training accuracy (0.9754), but it is significantly slower wall-clock (~321s) because it performs one optimizer update per example. The batched model (blue) is much faster (~91s) since it performs far fewer updates per epoch, but its loss decreases more gradually and remains higher even after many epochs, ending with lower training accuracy (0.9079). The small oscillations/spikes in the batched curve are expected with mini-batch optimization (stochastic batches), and the overall results highlight the typical tradeoff**: batching improves speed, but you often need more epochs and/or learning-rate tuning to match the convergence (and training accuracy) of unbatched training.**

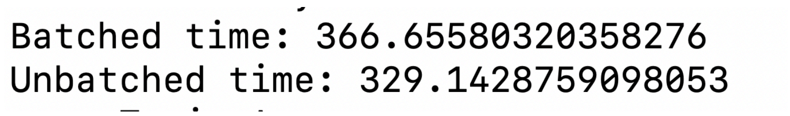
Running Batch size of 64 and batch size of 32 for 1000 epochs at lr = 0.01







```
Batched time: 284.36994314193726
Unbatched time: 386.711364984512 33
```



```
Batched time: 366.65580320358276
Unbatched time: 329.1428759098053
```

Based on the above observations, we can conclude that over an increased epoch the batch is better and faster than the unbatched process.