Objective

Optimize the implementation of naive algorithms by adding extensions or substituting algorithms for optimal counterpart extensions.

Implementation

We chose to extend our naive implementation of Neural Network (using Batch Gradient Descent) done in our homework assignment by adding Momentum, RMSProp, and ADAM algorithms to optimize our implementation and make our results robust (over an average of 5 runs)

Implementations using Numpy

We adapted our naive implementation from the homework assignment by modularizing it (for ease of addition of new algorithms). The structure of the network (64, 30, 10) was kept the same for the dataset collected from Scikit-Learn (MNIST digits small dataset), while the structure (784, 30, 10) was adapted to suit the large dataset collected from TensorFlow Keras API (MNIST digits full dataset). We kept the hidden-layer consistent to map out our implementation with as few variables as possible that differed between datasets.

We found that the ADAM optimizer outperformed on all possible combinations between datasets and algorithms with having a 5 run average of approx. 89% (with ⅘ runs > 90%). Furthermore, we also observed that training on our naive implementation had to be tuned differently in terms of the learning rate as compared to the one used in TensorFlow implementations. We attributed this behavior to the possibility of time step scaling of learning rate in the case of ADAM optimizer and the lack of scaling factor of (1/batch_size)[1] in TensorFlow implementations as compared to our implementation using NumPy.

Implementations using TensorFlow

We emulated our homework implementation as closely as we could (except for the scaling of learning rate) to test the possible extensions that would further optimize our algorithm. We tried a bunch of combinations of possible extensions - Momentum, RMSProp, ADAM, ADAM w. Dropout, Batch Norm, and settled on the first three - as they have a tight interlinkage within themselves and moreover we achieved satisfactory results with them.

[1] - TensorFlow Keras implementation documentation for SGD states the usage of learning rate variable in which the learning rate is not scaled by a factor of 1/no_examples. While, in our implementation adapted from homework, we did scale the learning rate by a factor of 1/N (we use mini-batches and hence, 1/batch_size).

Experiments

We list out our experiments, provide a possible explanation for the observed behavior, and present accuracy tables for comparison between implementations - TensorFlow vs NumPy for each of our chosen extensions - Momentum, RMSProp, and ADAM.

We chose the following combination for the hyperparameters from our naive implementation of batch gradient descent in homework assignment -

Activation function - **ReLU**, Weight Distribution - **Uniform**, Regularization - **L2 (0.01)**

Furthermore, we adapted our implementation of batch gradient descent and optimized it to include mini-batches, and the experiments performed and reported in this report are all done using the mini-batch implementation of gradient descent. This was done so as to, optimize our code in terms of running time for large datasets and robustness, and second, to match the implementation of how TensorFlow (and other libraries) implements gradient descent. And hence, we added one more hyperparameter to our list to tune - batch_size. Luckily, we found that a combination of batch_size = 32 for smaller datasets and a batch_size = 512 for large datasets worked best, and to maintain consistency, we kept the same across all our implementations.

Adding Momentum

We chose to add momentum because of its simplistic nature and yet the ability to produce fast results in comparison to MBGD (Mini-Batch Gradient Descent). Momentum is able to speed up the process of locating the minima by damping the oscillations produced in directions perceived to be vertical in comparison to the direction of minima. It achieves this behavior by taking an exponentially weighted average of the gradients over a fixed time frame and using this average to update the weights and biases instead of the traditional dW and db.
The number of gradients that it keeps an average (a moving average) over depends on the hyperparameter - beta_1[2]. Industry standards guide us to keep this value fixed at 0.9 - which has an equivalent effect of having a moving average over the past 10 examples or so and hence damping the oscillations in directions where gradient would've bounced from +ve to -ve and amplifying the gradient in the direction towards the minima.

We found that the effect of adding momentum was noticeable in terms of the improved accuracy on both, the TensorFlow implementation and our extension using NumPy. However, we did also notice the rather flaky behavior of SGD still being present and over many runs, we saw 1-shot accuracies having varying values all over the spectrum. This behavior was somewhat expected since what momentum does is it essentially speeds up the process of locating the minima but doesn't help with not hovering around the minima. We present our best selection of accuracies for both implementations in the table below.

[2] - Since all the three extensions use the same hyperparameter name - beta, we modify beta for momentum = beta_1, beta for RMSprop = beta_2, and ADAM has a combination of both.

Extension 1 - Adding **Momentum** to MBGD

Package	Dataset	Implementation	Accuracy (1-Shot)	Accuracy (5 runs average)
TensorFlow	Scikit-Learn MNIST	Naive	92.63 %	71.46 %
		Extended	85.81 %	73.71 %
	Keras MNIST	Naive	89.32 %	79.01 %
		Extended	89.45 %	85.85 %
Numpy	Scikit-Learn MNIST	Naive	89.29 %	86.20 %
		Extended	97.36 %	87.09 %
	Keras MNIST	Naive	93.24 %	85.50 %
		Extended	93.17 %	89.81 %

As we can see from the table above, we're able to achieve higher accuracies when compared to the naive implementation in both, TensorFlow and NumPy. However, as it evident from the table, again, the accuracy bump is significant in terms of the dataset from Keras but not so much in the dataset from Scikit-Learn. We were able to attribute this behavior to the amount of data that the Keras dataset has in comparison to the Scikit-Learn - around 35 times more. And as we know, with more data, comes more accuracy (usually).

Using RMSprop

We chose to replace our vanilla gradient descent with RMSprop owing to reasons similar to those of choosing momentum - because of its ability to produce fast results in comparison to MBGD (Mini-Batch Gradient Descent). But RMSprop is a more powerful tool in comparison to momentum because of its characteristics to normalize the gradients by an average gradient over a time frame. This averaging property is particularly helpful in cases where there are outliers present, which while using momentum would have an impact on the effective update, but in RMSprop, their value gets normalized and hence the effect is small. It achieves this behavior by taking an exponentially weighted average of the square of the gradients over a fixed time frame and using this average to normalize the weights and biases and use them instead of the traditional dW and db.
The number of square of gradients that it keeps an average (a moving average) over depends on the hyperparameter - beta_2[2]. The proposer of this method - Geoffery Hinton and industry standards guides us to keep this value fixed at 0.9 - which has an equivalent effect of having a moving average over the past 10 examples or so. The effect of RMSprop stands out truly in the case of outliers, where if 9 examples had a gradient of +0.001 and the 10 one had a gradient value of -0.009 then in the case of momentum and the weighted average, the weights wouldn't

be updated at all, however, in case of averaging the squares of gradients and normalizing each gradient using this average has the effect of impeding any step in the direction of the outlier.

We found that the effect of using RMSprop was noticeable in terms of the improved accuracy on the TensorFlow implementation, however, we're not able to reproduce stark differences in the accuracy using our implementation on NumPy. This behavior was troubling to debug and after a lot of hyperparameter combinations, we concluded that the learning rate conundrum had a role to play in this - having a value of 0.001 with TF worked well, but it did not do so well on our implementation (because of the scale factor of 1/batch_size[1]). We present our best selection of accuracies for both implementations in the table below.

Extension 2 - Using **RMSProp** optimizer instead of SGD

Package	Dataset	Implementation	Accuracy (1-Shot)	Accuracy (5 runs average)
TensorFlow	Scikit-Learn MNIST	Naive	86.79 %	77.11 %
		Extended	94.16 %	84.01 %
	Keras MNIST	Naive	89.21 %	82.44 %
		Extended	87.37 %	84.01 %
Numpy	Scikit-Learn MNIST	Naive	98.33 %	88.21 %
		Extended	94.99 %	87.23 %
	Keras MNIST	Naive	93.27 %	86.87 %
		Extended	91.07 %	87.16 %

As we mentioned above, the accuracy bump was not so stark as compared to that using momentum, but we did achieve more robust results in the sense of having less random accuracy values as we did while using momentum. The accuracy for our implementation using NumPy on the Scikit dataset did not improve in comparison to the naive implementation using NumPy (88.21% to 87.23%), however, we already did achieve better results in comparison to the TensorFlow implementation. We were able to attribute this behavior to the same scaling factor discussed previously in the report. However, for the case of a decrease in accuracy using NumPy on the Scikit dataset, we attributed this to perhaps the lack of outliers in the small dataset, and by definition both the implementations had almost the same gradient updates.

Using ADAM

We chose ADAM as our final extension to replicate the intentions of the authors of ADAM paper - to combine the effects of using momentum and RMSprop to achieve, both speed and robustness using gradient descent. ADAM combines the two concepts of speeding the search for minima and impeding random oscillations in directions other than towards the minima by normalizing the weighted average of gradients (calculated using momentum) over a time frame. It further applies a bias correction on the weighted averages to assert the gain of information from the gradients in the initial time steps. This correction is a decay function that has little effect once the gradients have been warmed up.
The number of square of gradients that it keeps an average (a moving average) over depends on the hyperparameter - beta_2[2] and the number of gradients that it keeps an average over depends on the hyperparameter - beta_1. The authors of the ADAM paper and other sources online suggest the use of beta_1=0.9 and beta_2=0.999 and having these values did prove to be robust and efficient enough for us as well. We could conclude that the reason ADAM outperforms the other two extensions is simply that it takes the good from both the algorithms and combines them in a way to achieve the best results - momentum suffers from outliers causing the gradient to be updated erratically, while RMSprop suffers from the slowness in comparison to momentum but solves the erratic behavior.

We found that the effect of using ADAM was noticeable across the board and we're able to achieve the most robust set of accuracies in comparison to the other algorithms that we introduced and experimented with earlier.

Extension 3 - Using **ADAM** optimizer instead of SGD

Package	Dataset	Implementation	Accuracy (1-Shot)	Accuracy (5 runs average)
TensorFlow	Scikit-Learn MNIST	Naive	93.46 %	82.64 %
		Extended	94.30 %	84.65 %
	Keras MNIST	Naive	88.99 %	86.56 %
		Extended	90.12 %	89.69 %
Numpy	Scikit-Learn MNIST	Naive	97.08 %	78.41 %
		Extended	96.11 %	85.12 %
	Keras MNIST	Naive	93.25 %	83.79 %
		Extended	91.31 %	88.84 %

Conclusions

We can say objectively that using ADAM optimizer produced the most robust results in terms of an average accuracy over 5 runs in comparison to using RMSprop and Momentum. The accuracies listed in the tables are the set of final accuracies that we wanted to report and for the purpose of being consistent, we did not alter the python notebooks attached to this report. However, while experimenting with the algorithms, we found that although momentum took a lot less time in comparison to RMS, it had a lot of variations in terms of its accuracies. RMS helped curb that to some extent but it suffered from running for a longer period than momentum. Additionally, RMS could not improve the accuracy of the Scikit dataset considerably as the other 2 algorithms could. And lastly, ADAM had a much narrower range of reporting accuracies and ran in considerable time as compared to RMS, with accuracies improved across the board.

With our experiments on adding extensions to our naive implementation from the homework assignment, we can conclude that although we achieved higher accuracies for almost all the experiments (with the exception of 1), the use of extensions should really be guided by the dataset and the underlying distribution of the data. Additionally, we tried manipulating the structure of the neural network to accommodate the large dataset and found that the accuracies got a bump, but we decided to leave it out of our final report to restore consistency and ease while drawing comparisons across the algorithms.