DEEP LEARNING WITH COTS HPC SYSTEMS

MODELS NOW OPERATE AT UNPRECEDENTED SCALE.

- Larger and larger datasets necessitate larger models.
- As neural networks get larger, traditional distributed machines needed to run certain networks will be out of reach for many researchers.
- However, it is possible to use GPUs and high-speed communications in order to coordinate gradient computation.
PRIOR WORK

• DistBelief
  • Can train 1 billion parameters with 16000 machines.
  • Might not scale past this point

• Two types of scaling
  • Scaling out
    • Using a large amount of machines in order to increase the amount of computational power.
  • Scaling up
    • Leveraging GPUs and other advanced hardware that's capable of more efficient calculation than CPUs
CHALLENGES

• Difficulty using large clusters of GPUs due to communication bottlenecks
  • Extremely fast to compute parameters on GPU, significantly slower to transfer information
  • Parallelism requires frequent synchronization.

• Managing communication across many GPUs makes algorithms complicated.
  • Traditional message passing is cumbersome
MODEL PARALLELISM

- Making each GPU responsible for a different part of the neural network.
- Works well with a single server
- Inefficient over Ethernet
- Requires frequent synchronization
CLUSTER SETUP

• 4 NVIDIA GTX680 GPUs
  • Small number of GPUs per machine prevents host machine from being overwhelmed.

• FDR Infiniband Adapter.
  • Infiniband is significantly faster than Ethernet, allowing speed to be maintained at scale.
  • Maximum throughput of 56Gbps

• Uses C++ on top of MVAPICH2 MPI implementation

• Balances number of GPUs with CPUs
ALGORITHM

• Sparse Autoencoder
• Nine-layer network consisting of a stack of three layers repeated three times.
  • Linear Filtering Layer
  • Pooling layer
  • Contrast Normalization Layer
• Designed to extract high level features from images
ALGORITHM (CONT.)

- Trained in a greedy, layer-wise fashion
- To optimize, only filter layers need to be trained.
- Optimized using standard stochastic gradient, with momentum.
• Point-wise operations are easy to implement
• Local connectivity operations are difficult with sparse input matrices
  • Sparseness of the input means code optimized for dense matrices won’t function.
• Difficult to optimize for recent GPUs due to the level of sophistication.
• Standard methods from convolutional networks didn’t work.
CHALLENGES WITH IMPLEMENTATION

- Implementing $Y=WX$ only achieved 300 GFLOPS, which didn’t utilize the full capacity of the GPUs
  - Each GPU able to handle up to 1 TFLOPS
- Storing the filter coefficients not applicable since filters could be larger than the GPU cache.
IMPLEMENTATION

• Input of first layer is a 4D array.
  • Dimensions:
    • Mini-batch size
    • Width
    • Height
    • Number of channels

• Dataset uses a large amount of 200x200 images with 3 channels
COMPUTING LINEAR RESPONSES

• Can increase efficiency by grouping neurons into sets where each neuron has an identical receptive field.
  • For every neuron in a set, the filters have the same sparsity patterns

• Allows efficient implementation by making matrix into a large set of dense small matrices
  • Allows computation as dense array for neurons that share a single receptive field
IMPLEMENTATION

• Set of neurons with similar receptive fields used to ensure $Y = WX$ can be calculated efficiently by allowing us to use dense matrix multiplication.
  • Use $Y_F = W_F \times X_F$
  • $W$ removes the non-zero rows of $W$ and the equivalent rows for $X$

• Uses MAGMA BLAS kernels
  • Uses advanced operations in order to efficiently run matrix operations.
IMPLEMENTATION

• Use block local connectivity to group neurons into 3D blocks
  • Each 3D block has the same receptive field.
  • Blocks need to be large to fully take advantage of GPU efficiency
  • Block size can be expanded by expanding width or depth, but the step size needs to be increased.

• Allows fast GPU kernels to exceed 1 TFLOP
COMMUNICATION WITH MPI

- GPUs are parallelized using a model parallel scheme
  - All GPUs work on each minibatch
- Distribution of arrays are partitioned spatially
- Each GPU computes responses of neurons that are assigned to it.
- Filter weights partitioned as well, such that the weights are stored on their respective neuron.
• Fetches for neurons that need values across multiple GPUs might be messy.
  • Uses a simple distributed array abstraction to hide the communication from the rest of the code.
  • Each GPU has an input and output window
    • Output: array that will be filled with results
    • Input: array of values that are needed in order to compute the output
    • On runtime, each GPU sends the intersection of its output and the other GPUs input, and receives the intersection of the other GPUs output and the
SCALING EFFICIENCY

- Recording average time to compute all layers
- Scaling tested through short optimization runs.
  - Feedforward pass to find objective function, and full backwards pass

Figure 5. Time taken to perform a mini-batch update for all weights in large neural networks of sizes ranging from 180 million parameters up to 11.2 billion parameters.
SCALING

- Little speed up when running the document at low GPU counts
- System works significantly better with larger systems.

Figure 6. Factor speedup obtained for varying sizes of network and number of GPUs, normalized for the size of the network.
HIGH LEVEL OBJECT SELECTIVE FEATURES

• Large neural network tested on large dataset of harvested Youtube thumbnails.
  • Data rescaled for consistency and contrast normalized
• Similar three-layer network as previously described.
• Each neuron tested by recording responses from 13152 labelled faces and 48000 distractors from ImageNet
  • Some neurons are able to find a face with 88% accuracy
• Data used to train with a larger network to test scalability.
• Most selective neurons in the larger network are less selective than the neurons in the smaller network.

• Nonlinearities and hyper-parameter tuning help with this but are still not quite as good.

<table>
<thead>
<tr>
<th>Object</th>
<th>Random guess</th>
<th>Best in random net</th>
<th>Best in 1.8B param. net</th>
<th>Best in 11B param. net.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human faces</td>
<td>64.7%</td>
<td>64.8%</td>
<td>88.2%</td>
<td>86.5%</td>
</tr>
<tr>
<td>Upper body</td>
<td>64.7%</td>
<td>64.8%</td>
<td>80.2%</td>
<td>74.5%</td>
</tr>
<tr>
<td>Cats</td>
<td>64.7%</td>
<td>64.8%</td>
<td>73.0%</td>
<td>69.4%</td>
</tr>
</tbody>
</table>
LARGE BATCH OPTIMIZATION FOR DEEP LEARNING: TRAINING BERT IN 76 MINUTES

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, Cho-Jui Hsieh
PROBLEMS

• Large datasets are extremely difficult to work with and require a high level of optimization
• SGD’s sequential nature makes it extremely hard to scale.
• Asynchronous set-up can lead to degraded performance.
RECENT ADVANCES

• Recent advances involve using Synchronous SGD with large minibatches calculating gradient in parallel
  • Increasing the batch size naively also can cause degraded performance.
  • Able to function as an efficient alternative to asynchronous SGD
• Linear scaling of the learning rate can speed up training
  • Doesn't work past a certain batch size
  • Harmful during the early phase, needs hand-tuning
EARLIER WORKS

- Using larger minibatches improves convergence at the cost of computation.
- Linearly improving the learning rate works up to a certain point
  - **LR Warmup** can be used for the first few epochs before linear scaling to prevent loss in generalization performance.
    - Warmup strategy involves using lower learning rates at the start of training
- Adaptive learning rates can reduce the hand-tuning of hyperparameters.
  - Can be used at large scales without hurting performance
LAMB

• Specifically designed for large batch learning.
• Able to rapidly train on BERT without degrading.
• Extremely efficient on image classification models.
  • The first adaptive solver with high accuracy for image classification models.
• Supports adaptive elementwise updating and layer-wise learning rates
ISSUES WITH STOCHASTIC GRADIENT DESCENT

• Goal is to solve non-convex stochastic optimization problems like that of the first equation

• Third equation shows the iterates of the SGD algorithm, for a tuned learning rate.
  • Tuning the learning rate isn’t easy
  • Depending on the max smoothness (the maximum Lipschitz constant) can cause slow convergence.
GENERAL STRATEGY

• Using a standard update doesn’t scale well.
  • Normalize the update to the unit $l_2$ norm.
  • Scale the learning rate to ensure the norm of the update is the same as that of the parameter.
  • Change in learning rate is approximately equal to the inverse of the Lipschitz constant, or $\frac{\|g'(t)\|}{\|g'(0)\|}$.
TESTING DIFFERENT NORMS

- Multiple matrix and tensor norms tested for updating parameters.
- No significant difference in terms of validation accuracy.

![ImageNet/ResNet-50 by LAMB (90 epochs, Batch Size=32K)](image)
**LARS ALGORITHM**

- Uses heavy-ball momentum to reduce the variance in stochastic gradients at the cost of little bias.
- Converges better than SGD when the gradient is denser than the curvature and stochasticity.
LAMB ALGORITHM

• Per dimension normalization per the square root of the second moment used in ADAM

• Layerwise normalization obtained due to layerwise adaptivity.

• Convergence rates of LARS and LAMB depend on average of Lipschitz constants rather than the maximum one.

*Theorem 3.* Let $\eta_t = \eta - \sqrt{\beta_1 (z_t) / \alpha_2}$ for all $t \in [T]$, $b = T$, $d_i = d / h$ for all $i \in [h]$, and $\alpha_1 \leq \phi(v) \leq \alpha_2$ for all $v > 0$ where $\alpha_1, \alpha_2 > 0$. Then for $x_t$ generated using LAMB (Algorithm 2), we have the following bounds:

1. When $\beta_2 = 0$, we have
   \[
   \left( \mathbb{E} \left[ \frac{1}{\sqrt{d}} \| \nabla f(x_a) \|_1 \right] \right)^2 \leq O \left( \frac{\| f(x_t) - f(x^*) \|_{L_{\text{avg}}}}{T} + \frac{\| \delta \alpha \|}{T h} \right),
   \]

2. When $\beta_2 > 0$, we have
   \[
   \mathbb{E} \| \nabla f(x_a) \|^2 \leq O \left( \sqrt{\frac{G^2 d}{h (1 - \beta_2)}} \times \left[ \frac{2 (f(x_t) - f(x^*)) \| L \|_1}{T} + \frac{\| \delta \alpha \|}{\sqrt{T}} \right] \right),
   \]
where $x^*$ is an optimal solution to the problem in equation 1 and $x_a$ is an iterate uniformly randomly chosen from $\{x_1, \ldots, x_T\}$.
C COMPARISON OF CONVERGENCE RATES OF LARS AND SGD

Inspired by the comparison used by (Bernstein et al., 2018) for comparing SIGN SGD with SGD, we define the following quantities:

\[
\left( \frac{1}{h} \sum_{i=1}^{h} \| \nabla f(x_i) \| \right)^2 = \frac{\psi_i \| \nabla f(x_i) \|^2}{h} \geq \frac{\psi_i d^2 \| \nabla f(x_i) \|^2}{h}
\]

\[
\frac{\| L \|_L^2}{h^2} \leq \frac{\psi_i d \| L \|_L^2}{h^2}
\]

\[
\frac{\| \sigma \|_L^2}{h^2} = \frac{\psi_i d \| \sigma \|^2}{h^2}
\]

Then LARS convergence rate can be written in the following manner:

\[
\mathbb{E} \left[ \| \nabla f(x_0) \|^2 \right] \leq O \left( \frac{(f(x_1) - f(x^*)) L_{\infty}}{T} \psi_i \frac{d^2}{e_2^2} + \frac{\| \sigma \|^2 \| \sigma \|^2}{T^2} \psi^2 \right).
\]

If \( \psi_i \ll \psi^2 \) and \( \psi \ll \psi^2 \) then LARS (i.e., gradient is more denser than curvature or stochasticity), we gain over SGD. Otherwise, SGD’s upper bound on convergence rate is better.

CONVERGENCE RATES

• LARS converges faster than SGD when the gradient is denser than the stochasticity
• LARS and LAMB are generally faster than SGD, since they use the average Lipschitz constant rather than the maximum one.
• ADAMW has a term that corrects the learning rate.
• Since this is similar to the learning rate warm-up, it can be removed.
• This was tested on both BERT and ImageNet.
EXPERIMENTS

• $\beta_1$ and $\beta_2$ are set to 0.9 and 0.999

• Uses the BERT baseline learning rate of $\eta_t = \eta_0 \times (1 - t/T)$

• Uses minimal hyper parameter tuning for LAMB in order to demonstrate LAMB’s robustness.

• Hyperparameters tuned for ADAM, ADAGRAD, and ADAMW using grid search, and ADAMW used tune weight decay.

• Uses TPUv3
  • Single pod contains 1024 chips and can reach 100 petaflops performance
EXPERIMENTS

• Experiments run using BERT training
  • Used a large dataset containing a combination of Wikipedia and BooksCorpus
  • Tested model using SQuAD-v1, a language comprehension dataset.
  • Results judged using F1 score
  • Used a similar set-up to prior work for testing purposes.

• TPUv3 Pod necessitated a maximum batch size of 32K
TRAINING PROCEDURE

• BERT training contains two stages – pre-training and fine-tuning

• Second stage can have a maximum batch size of 32768 due to the memory limits of the TPUv3 pod

• First stage batch size can be increased to 131072, due to shorter sequences

• First stage batch size was run at batch size of 65536, stabilizing the first stage
  • Decreasing the batch size can result in chaotic and poor optimization
  • To stabilize the second stage, the learning rate was warmed up from 0.

• Process allowed BERT to be trained in 8599 iterations, or 76 minutes.
BERT TRAINING (CONT.)

• Able to get a massive 49.1 speedup over previous methods
  • This is due to the use of synchronous data parallelism
  • Requires communication overhead due to transferring gradients
• Less accurate than ResNet-50 due to BERT’s large size.
## EXPERIMENTS

Trained using the BERT model

<table>
<thead>
<tr>
<th>Solver</th>
<th>batch size</th>
<th>steps</th>
<th>F1 score on dev set</th>
<th>TPU</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>512</td>
<td>1000k</td>
<td>90.395</td>
<td>16</td>
<td>81.4h</td>
</tr>
<tr>
<td>LAMB</td>
<td>512</td>
<td>1000k</td>
<td>91.752</td>
<td>16</td>
<td>82.8h</td>
</tr>
<tr>
<td>LAMB</td>
<td>1k</td>
<td>500k</td>
<td>91.761</td>
<td>32</td>
<td>43.2h</td>
</tr>
<tr>
<td>LAMB</td>
<td>2k</td>
<td>250k</td>
<td>91.946</td>
<td>64</td>
<td>21.4h</td>
</tr>
<tr>
<td>LAMB</td>
<td>4k</td>
<td>125k</td>
<td>91.137</td>
<td>128</td>
<td>693.6m</td>
</tr>
<tr>
<td>LAMB</td>
<td>8k</td>
<td>62500</td>
<td>91.263</td>
<td>256</td>
<td>390.5m</td>
</tr>
<tr>
<td>LAMB</td>
<td>16k</td>
<td>31250</td>
<td>91.345</td>
<td>512</td>
<td>200.0m</td>
</tr>
<tr>
<td>LAMB</td>
<td>32k</td>
<td>15625</td>
<td>91.475</td>
<td>1024</td>
<td>101.2m</td>
</tr>
<tr>
<td>LAMB</td>
<td>64k/32k</td>
<td>8599</td>
<td>90.584</td>
<td>1024</td>
<td>76.19m</td>
</tr>
</tbody>
</table>
BERT TRAINING RESULTS

- LAMB is significantly better with large scale BERT training than other optimizers
- ADAMW failed to achieve the target score after a batch size of 16K
- LAMB consistently scored better than LARS in terms of F1 score.
- LAMB trained BERT in 76 minutes.

Table 2: LAMB achieves a higher performance (F1 score) than LARS for all the batch sizes. The baseline achieves a F1 score of 90.390. Thus, LARS stops scaling at the batch size of 16K.

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>512</th>
<th>1K</th>
<th>2K</th>
<th>4K</th>
<th>8K</th>
<th>16K</th>
<th>32K</th>
</tr>
</thead>
<tbody>
<tr>
<td>LARS</td>
<td>90.717</td>
<td>90.369</td>
<td>90.748</td>
<td>90.537</td>
<td>90.548</td>
<td>89.589</td>
<td>diverge</td>
</tr>
<tr>
<td>LAMB</td>
<td>91.752</td>
<td>91.761</td>
<td>91.946</td>
<td>91.137</td>
<td>91.263</td>
<td>91.345</td>
<td>91.475</td>
</tr>
</tbody>
</table>
Table 8: **ADAMW** stops scaling at the batch size of 16K. The target F1 score is 90.5. **LAMB** achieves a F1 score of 91.345. The table shows the tuning information of **ADAMW**. In this table, we report the best F1 score we observed from our experiments.

<table>
<thead>
<tr>
<th>Solver</th>
<th>batch size</th>
<th>warmup steps</th>
<th>LR</th>
<th>last step information</th>
<th>F1 score on dev set</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAMW</td>
<td>16K</td>
<td>0.05×31250</td>
<td>0.0001</td>
<td>loss=8.04471, step=28126</td>
<td>diverged</td>
</tr>
<tr>
<td>ADAMW</td>
<td>16K</td>
<td>0.05×31250</td>
<td>0.0002</td>
<td>loss=7.89673, step=28126</td>
<td>diverged</td>
</tr>
<tr>
<td>ADAMW</td>
<td>16K</td>
<td>0.05×31250</td>
<td>0.0003</td>
<td>loss=8.35102, step=28126</td>
<td>diverged</td>
</tr>
<tr>
<td>ADAMW</td>
<td>16K</td>
<td>0.10×31250</td>
<td>0.0001</td>
<td>loss=2.01419, step=31250</td>
<td>86.034</td>
</tr>
<tr>
<td>ADAMW</td>
<td>16K</td>
<td>0.10×31250</td>
<td>0.0002</td>
<td>loss=1.04689, step=31250</td>
<td>88.540</td>
</tr>
<tr>
<td>ADAMW</td>
<td>16K</td>
<td>0.10×31250</td>
<td>0.0003</td>
<td>loss=8.05845, step=20000</td>
<td>diverged</td>
</tr>
<tr>
<td>ADAMW</td>
<td>16K</td>
<td>0.20×31250</td>
<td>0.0001</td>
<td>loss=1.53706, step=31250</td>
<td>85.231</td>
</tr>
<tr>
<td>ADAMW</td>
<td>16K</td>
<td>0.20×31250</td>
<td>0.0002</td>
<td>loss=1.15500, step=31250</td>
<td>88.110</td>
</tr>
<tr>
<td>ADAMW</td>
<td>16K</td>
<td>0.20×31250</td>
<td>0.0003</td>
<td>loss=1.48798, step=31250</td>
<td>85.653</td>
</tr>
</tbody>
</table>
TRAINING LOSS

- LAMB is capable of making the program converge smoothly even at extremely a batch size of 64K.
- LAMB is able to get 76.8% scaling accuracy with a batch size of 64K.
RESNET-50 EXPERIMENTS

- ResNet-50 is an industry standard metric.
- Prior best results use momentum-based SGD or the LARS optimizer.
- ADAMW optimizer is incapable of high accuracy with regards to ResNet-50.
  - Comprehensive hyperparameter tuning only brings ADAMW up to 73% accuracy.
- LAMB is comparable to LARS, but has greater accuracy at higher scales.

<table>
<thead>
<tr>
<th>optimizer</th>
<th>adagrad/adagrad+</th>
<th>adam/adam+</th>
<th>adamw/adamw+</th>
<th>momentum</th>
<th>lamb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.5538/0.7201</td>
<td>0.6604/0.7348</td>
<td>0.6727/0.7307</td>
<td>0.7520</td>
<td>0.7666</td>
</tr>
</tbody>
</table>
### RESULTS

#### Table 4: Untuned LAMB for BERT training across different batch sizes (fixed #epochs). We use square root LR scaling and linear-epoch warmup. For example, batch size 32K needs to finish 15625 iterations. It uses $0.2 \times 15625 = 3125$ iterations for learning rate warmup. BERT’s baseline achieved a F1 score of 90.395. We can achieve an even higher F1 score if we manually tune the hyperparameters.

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>512</th>
<th>1K</th>
<th>2K</th>
<th>4K</th>
<th>8K</th>
<th>16K</th>
<th>32K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning Rate</td>
<td>$5 \times 2^{3.0 \times 10^3}$</td>
<td>$5 \times 2^{3.5 \times 10^3}$</td>
<td>$5 \times 2^{3.0 \times 10^3}$</td>
<td>$5 \times 2^{3.5 \times 10^3}$</td>
<td>$5 \times 2^{3.0 \times 10^3}$</td>
<td>$5 \times 2^{3.5 \times 10^3}$</td>
<td>$5 \times 2^{3.0 \times 10^3}$</td>
</tr>
<tr>
<td>Warmup Ratio</td>
<td>$\frac{1}{320}$</td>
<td>$\frac{1}{160}$</td>
<td>$\frac{1}{80}$</td>
<td>$\frac{1}{40}$</td>
<td>$\frac{1}{20}$</td>
<td>$\frac{1}{10}$</td>
<td>$\frac{1}{5}$</td>
</tr>
<tr>
<td>F1 score</td>
<td>91.752</td>
<td>91.761</td>
<td>91.946</td>
<td>91.137</td>
<td>91.263</td>
<td>91.345</td>
<td>91.475</td>
</tr>
<tr>
<td>Exact Match</td>
<td>85.090</td>
<td>85.260</td>
<td>85.355</td>
<td>84.172</td>
<td>84.901</td>
<td>84.816</td>
<td>84.939</td>
</tr>
</tbody>
</table>

#### Table 5: Untuned LAMB for ImageNet training with ResNet-50 for different batch sizes (90 epochs). We use square root LR scaling and linear-epoch warmup. The baseline Goyal et al. (2017) gets 76.3% top-1 accuracy in 90 epochs. Stanford DAWN Bench (Coleman et al., 2017) baseline achieves 93% top-5 accuracy. LAMB achieves both of them. LAMB can achieve an even higher accuracy if we manually tune the hyperparameters.

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>512</th>
<th>1K</th>
<th>2K</th>
<th>4K</th>
<th>8K</th>
<th>16K</th>
<th>32K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning Rate</td>
<td>$\frac{4}{2^{3.0 \times 100}}$</td>
<td>$\frac{4}{2^{3.5 \times 100}}$</td>
<td>$\frac{4}{2^{3.0 \times 100}}$</td>
<td>$\frac{4}{2^{3.5 \times 100}}$</td>
<td>$\frac{4}{2^{3.0 \times 100}}$</td>
<td>$\frac{4}{2^{3.5 \times 100}}$</td>
<td>$\frac{4}{2^{3.0 \times 100}}$</td>
</tr>
<tr>
<td>Warmup Epochs</td>
<td>0.3125</td>
<td>0.625</td>
<td>1.25</td>
<td>2.5</td>
<td>5</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Top-5 Accuracy</td>
<td>0.9335</td>
<td>0.9349</td>
<td>0.9353</td>
<td>0.9332</td>
<td>0.9331</td>
<td>0.9322</td>
<td>0.9308</td>
</tr>
<tr>
<td>Top-1 Accuracy</td>
<td>0.7696</td>
<td>0.7706</td>
<td>0.7711</td>
<td>0.7692</td>
<td>0.7689</td>
<td>0.7666</td>
<td>0.7642</td>
</tr>
</tbody>
</table>
TUNING PROCESS FOR ADAM

• For testing ADAMW/ADAM/ADAGRAD, a warm-up and decay scheme was added in order to improve accuracy
  • 5-epoch warm-up stabilized the initial stage
  • The learning rate was multiplied by 0.1 at the 30th, 60th, and 80th epochs.
• Multiple tuning sets used, since both L2 normalization and weight decay can affect performance
  • Tuning sets with L2 normalization enabled and disabled
  • Tuning sets with AdamW+
• Still performed worse than the LAMB optimizer
EXPERIMENTS WITH SMALLER DATASETS

- DavidNet
  - Residual ConvNet that is the fastest method for the CIFAR-10 dataset
    - Image classification with 10 classes
    - Able to achieve near human level accuracy
  - Fastest optimizer for this network was a momentum SGD.
- LAMB can outperform this.
NESTEROV MOMENTUM FOR LAMB

- Different form of momentum step that has been shown to work better than standard gradients.
- Using Nesterov’s accelerated gradient is roughly comparable with a standard gradient when compared to LAMB.
THANK YOU FOR LISTENING