How to Build Your Own Deep Neural Network

Kwai Wong, Stan Tomov
Daniel Nichols, Rocco Febbo, Florent Lopez
Julian Halloy, Xianfeng Ma

University of Tennessee, Knoxville

PEARC20, July 31, 2020
Acknowledgements:

• Support from NSF, UTK, JICS, NICS

• MagmaDNN is a project grown out from the RECSEM REU Summer program supported under NSF award #1659502

• Efforts and reference materials from many colleagues, collaborators, researchers, and students


• Source code: www.bitbucket.org/icl/magmadnn

• LAPENNA, www.jics.utk.edu/lapenna
How to Build Your Own Neural Network Framework

The tutorial aims to show researchers how to write a deep neural network (DNN) software program. In this tutorial, we will show how typical DNN algorithms are implemented, illustrated with programming structure of MagmaDNN and example exercises.

✓ This tutorial is intended for researchers that have previously used an open source DNN framework but would like to learn more about its programming structure
✓ The primary computing platform is a Linux computer with a Nvidia GPU card. We will not use the Jetson Nano cards for demonstrations. Illustration of computing on multiple GPUs will be done on a DGX box.
✓ This tutorial has 4 major sections, each one will last for about 45 minutes and a 10 minutes Q&A session at the end of each section.
✓ Please fill out the evaluation form

✓ Overview of Deep Neural Network : MLP, CNN, CUDNN, MKL DNN, TensorFlow, MagmaDNN, MNIST example
✓ GPU computing : Linear Algebra, MKL, CUDA, GPU programming, Magma, mixed precision, exercises
✓ Programming structure of a DNN framework : I/O, Memory Management, Tensor, layers, activation and fit functions, CFIR 100, MagmaDNN
✓ Multiple GPUs and in-depth example exercises : data and model parallelism, distributed Computing, VGG, ResNet ImageNet.
Acknowledgements and References

This portion of the tutorial contains many extracted materials from many online websites and courses. Listed below are the major sites. This portion of the materials is not intended for public distribution. Please visit the sites websites for detail contents. If you are beginner users of DNN, I suggest to read the following list of websites in its order.

1) http://neuralnetworksanddeeplearning.com
2) https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist
3) https://www.deeplearning.ai/deep-learning-specialization/
4) http://cs231n.stanford.edu/
5) More sites are listed in www.jics.utk.edu/actia
6) Others are listed on the slides
Overview of Deep Neural Network

- Introduction to machine learning and Deep Neural Network
- Activation function, optimization, cost function, SDG
- Regression and Classification
- Regression, MLP, forward and backward process
- Computational graph, example of backpropagation
- Classification, CNN,
- Arithmetic of CNN
- Number of Parameters
- AlexNet, VGG16, ResNet
- More about optimizers
- TensorFlow, MagmaDNN, MNIST example
✓ **Artificial Intelligence (AI)**: science and engineering of making intelligent machines to perform the human tasks (John McCarthy, 1956). AI applications is ubiquitous.

✓ **Machine learning (ML)**: A field of study that gives computers the ability to learn without being explicitly programmed (Arthur Samuel, 1959). A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E (Tom Mitchell, 1998).

✓ **Neural Network (NN)**: Neural Network modeling, a subfield of ML is algorithm inspired by structure and functions of biological neural nets

✓ **Deep Neural Network (DNN)**: (aka deep learning): an extension of NN composed of many layers of functional neurons, is dominating the science of modern AI applications

✓ **Supervised Learning (SL)**: A class in ML, dataset has labeled values, use to predict output values associated with new input values.
A node in the neural network is a mathematical function or activation function which maps input to output values.

Inputs represent a set of vector containing weights (w) and bias (b). They are the sets of parameters to be determined for prediction, regression.

Many nodes form a neural layer, links connect layers together, defining a NN model.

Activation function (f or σ), is generally a nonlinear data operator which facilitates identification of complex features.
Activation Function

\[ \sigma(z) \equiv \frac{1}{1 + e^{-z}}. \]

- Sigmond (logistic) function, \( \sigma \), is a smooth out preceptron operating on a set of data, \( \sigma(w \cdot x+b) \). The smoothness of \( \sigma \) means that small changes \( \Delta w \) in the weights and \( \Delta b \) in the bias will produce a small change \( \Delta \text{output} \) in the output from the neuron.

- If \( z = w \cdot x+b \) is a large positive number, then \( \exp(-z) \approx 0 \) and so \( \sigma(z) \approx 1 \).

- If \( z = w \cdot x+b \) is very negative, then \( \exp(-z) \to \infty \), and \( \sigma(z) \approx 0 \). So when \( z = w \cdot x+b \) is very negative.

- It's only when \( w \cdot x+b \) is of modest size that there's much deviation from the perceptron model.

- \( \Delta \text{output} \) is a linear function of the changes \( \Delta w_j \) and \( \Delta b \) in the weights and bias. This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output.
Minimization, Training

Define a cost function, $C$ is the quadratic cost function also referred as the mean squared error (MSE).

Define a cost function $C(w,b)$, the mean squared error (MSE), so the neural network training algorithm finds the weights and biases giving $C(w,b) \approx 0$, or minimizes the cost $C(w,b)$ as a function of the weights and biases, casting it as an optimization problem using the gradient descent algorithm.

- Need to find a way of choosing $\Delta v_1$ and $\Delta v_2$ so as to make $\Delta C$ (output) negative (move to minimum).
- To make gradient descent work correctly, choose the learning rate $\eta$ to be small but not too small such that the gradient descent algorithm stops works properly.

$\nabla C \equiv \left( \frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T.$

Choose $\Delta v = -\eta \nabla C$, $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \| \nabla C \|^2.$

$v \rightarrow v' = v - \eta \nabla C.$
Neural network is an optimization process to find a set of parameters that ensure the prediction function $f(x, W)$ to be as close as possible to the true solution $y$ for any input $x$.

Use gradient descent to find the weights $w$ and biases $b$ which minimize the cost function.

To compute the gradient $\nabla C$ we need to compute the gradients $\nabla Cx$ separately for each training input, $x$, and then average them, $\nabla C = 1/n \sum \nabla Cx$. Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly.

A way is to use stochastic gradient descent to speed up learning. The idea is to estimate the gradient $\nabla C$ by computing a small sample of randomly chosen training inputs, refer to as a mini-batch of input.

By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient $\nabla C$, and this helps speed up gradient descent, and thus learning, provided the sample size $m$ is large enough that the average value of the $\nabla CX_j$ roughly equals to the average over all $\nabla Cx$.

Another randomly chosen mini-batch are selected and trained, until all the exhausted the training inputs are used. It is said to complete an epoch (iteration) of training, then more iteration.

$$
\frac{\sum_{j=1}^{m} \nabla Cx_j}{m} \approx \frac{\sum_x \nabla Cx}{n} = \nabla C,
$$

$$
w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k},
$$

$$
b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.
$$
Forward Path Exercise

\[
\begin{align*}
net_{h1} &= w_1 * i_1 + w_2 * i_2 + b_1 * 1 \\
net_{h1} &= 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775 \\
out_{h1} &= \frac{1}{1 + e^{-net_{h1}}} = \frac{1}{1 + e^{-0.3775}} = 0.593269992 \\
out_{h2} &= 0.596884378 \\
net_{o1} &= w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1 \\
net_{o1} &= 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967 \\
E_{total} &= \sum \frac{1}{2} (target - output)^2 \\
E_{o1} &= \frac{1}{2} (target_{o1} - out_{o1})^2 = \frac{1}{2} (0.01 - 0.75136507)^2 = 0.274811083 \\
out_{o2} &= 0.772928465 \\
E_{o2} &= 0.023560026 \\
E_{total} &= E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109
\end{align*}
\]

Backpropagation Example

\[ E_{\text{total}} = \frac{1}{2} (\text{target}_{o1} - \text{out}_{o1})^2 + \frac{1}{2} (\text{target}_{o2} - \text{out}_{o2})^2 \]

\[ \frac{\partial E_{\text{total}}}{\partial w_5} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{o1}} \cdot \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} \cdot \frac{\partial \text{net}_{o1}}{\partial w_5} \]

\[ \text{out}_{o1} = \frac{1}{1 + e^{-\text{net}_{o1}}} \]

\[ \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} = \text{out}_{o1}(1 - \text{out}_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602 \]

\[ \text{net}_{o1} = w_5 \cdot \text{out}_{h1} + w_6 \cdot \text{out}_{h2} + b_2 \]

\[ \frac{\partial \text{net}_{o1}}{\partial w_5} = 1 \cdot \text{out}_{h1} \cdot w_5^{(1-1)} + 0 + 0 = \text{out}_{h1} = 0.593269992 \]

\[ \frac{\partial E_{\text{total}}}{\partial \text{out}_{o1}} = \frac{\partial E_{\text{total}}}{\partial \text{net}_{o1}} \cdot \frac{\partial \text{net}_{o1}}{\partial w_5} \]

\[ \delta_{o1} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{o1}} \cdot \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} \]

\[ \delta_{o1} = -(\text{target}_{o1} - \text{out}_{o1}) \cdot \text{out}_{o1}(1 - \text{out}_{o1}) \cdot \text{out}_{h1} \]

\[ \frac{\partial E_{\text{total}}}{\partial w_5} = \delta_{o1} \cdot \text{out}_{h1} \]

\[ \frac{\partial E_{\text{total}}}{\partial w_6} = -\delta_{o1} \cdot \text{out}_{h1} \]

\[ w_5^+ = w_5 - \eta \cdot \frac{\partial E_{\text{total}}}{\partial w_5} = 0.4 - 0.5 \times 0.082167041 = 0.35891648 \]

\[ w_6^+ = 0.408666186 \]

\[ w_5^+ = 0.511301270 \]

\[ w_6^+ = 0.561370121 \]
Backpropagation Example

\[ \frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} \cdot \frac{\partial out_{h1}}{\partial net_{h1}} \cdot \frac{\partial net_{h1}}{\partial w_1} \]

\[ \frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} \]

\[ \frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} \]

\[ \frac{\partial E_{o1}}{\partial net_{o1}} = 0.74136507 \cdot 0.186815602 = 0.138498562 \]

\[ net_{o1} = w_5 \cdot out_{h1} + w_6 \cdot out_{h2} + b_2 \cdot 1 \]

\[ net_{o1} = w_5 = 0.40 \]

\[ \frac{\partial E_{o2}}{\partial net_{o1}} = 0.138498562 \cdot 0.40 = 0.055399425 \]

\[ \frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} \cdot \frac{\partial out_{o1}}{\partial net_{o1}} \]

\[ out_{h1} = \frac{1}{1 + e^{-net_{h1}}} \]

\[ net_{h1} = w_1 \cdot i_1 + w_3 \cdot i_2 + b_1 \cdot 1 \]

\[ net_{h1} = i_1 = 0.05 \]

\[ w_1^+ = w_1 - \eta \cdot \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 \cdot 0.000438568 = 0.149780716 \]

\[ w_2^+ = 0.19956143 \]

\[ w_3^+ = 0.24975114 \]

\[ w_4^+ = 0.29950229 \]
ConvolusMon NN

Convolutional 3x3 filters=12
\[ W_1[3, 3, 1, 12] \]

Convolutional 6x6 filters=24
\[ W_2[6, 6, 12, 24] \text{ stride 2} \]

Convolutional 6x6 filters=32
\[ W_2[6, 6, 24, 32] \text{ stride 2} \]

Dense layer
\[ W_4[1568, 200] \]

Softmax dense layer
\[ W_5[200, 10] \]

https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/#0
Number of Parameters
AlexNet

Number of Parameters

✓ Input: Color images of size 227x227x3. The AlexNet paper mentions the input size of 224x224 but that is a typo in the paper.
✓ Conv-1: The first convolutional layer consists of 96 kernels of size 11x11 applied with a stride of 4 and padding of 0.
✓ MaxPool-1: The maxpool layer following Conv-1 consists of pooling size of 3x3 and stride 2.
✓ Conv-2: The second conv layer consists of 256 kernels of size 5x5 applied with a stride of 1 and padding of 2.
✓ MaxPool-2: The maxpool layer following Conv-2 consists of pooling size of 3x3 and a stride of 2.
✓ Conv-3: The third conv layer consists of 384 kernels of size 3x3 applied with a stride of 1 and padding of 1.
✓ Conv-4: The fourth conv layer has the same structure as the third conv layer. It consists of 384 kernels of size 3x3 applied with a stride of 1 and padding of 1.
✓ Conv-5: The fifth conv layer consists of 256 kernels of size 3x3 applied with a stride of 1 and padding of 1.
✓ MaxPool-3: The maxpool layer following Conv-5 consists of pooling size of 3x3 and a stride of 2.
✓ FC-1: The first fully connected layer has 4096 neurons.
✓ FC-2: The second fully connected layer has 4096 neurons.
✓ FC-3: The third fully connected layer has 1000 neurons.

Number of Parameters

Size of the Output Tensor (Image) of a Conv Layer

- $O = \text{Size (width) of output image}.$
- $I = \text{Size (width) of input image}.$
- $K = \text{Size (width) of kernels used in the Conv Layer}.$
- $N = \text{Number of kernels}.$
- $S = \text{Stride of the convolution operation}.$
- $P = \text{Padding}.$

The size ($O$) of the output image is given by

$$O = \frac{227 - 11 + 2 \times 0}{4} + 1 = 55$$

$$O = \frac{I - K + 2P}{S} + 1$$

Number of Parameters of a Conv Layer

- $W_c = \text{Number of weights of the Conv Layer}.$
- $B_c = \text{Number of biases of the Conv Layer}.$
- $P_c = \text{Number of parameters of the Conv Layer}.$
- $K = \text{Size (width) of kernels used in the Conv Layer}.$
- $N = \text{Number of kernels}.$
- $C = \text{Number of channels of the input image}.$

Number of Parameters of a MaxPool Layer

$$W_c = K^2 \times C \times N$$

$$B_c = N$$

$$P_c = W_c + B_c$$

Size of Output Tensor (Image) of a MaxPool Layer

The size ($O$) of the output image is given by

$$O = \frac{55 - 3}{2} + 1 = 27$$

$$O = \frac{I - P_s}{S} + 1$$

Number of Parameters

The total number of parameters in AlexNet is the sum of all parameters in the 5 Conv Layers + 3 FC Layers. It comes out to a whopping **62,378,344**! The table below provides a summary.

<table>
<thead>
<tr>
<th>Layer Name</th>
<th>Tensor Size</th>
<th>Weights</th>
<th>Biases</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Image</td>
<td>227x227x3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Conv-1</td>
<td>55x55x96</td>
<td>34,848</td>
<td>96</td>
<td>34,944</td>
</tr>
<tr>
<td>MaxPool-1</td>
<td>27x27x96</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Conv-2</td>
<td>27x27x256</td>
<td>614,400</td>
<td>256</td>
<td>614,656</td>
</tr>
<tr>
<td>MaxPool-2</td>
<td>13x13x256</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Conv-3</td>
<td>13x13x384</td>
<td>884,736</td>
<td>384</td>
<td>885,120</td>
</tr>
<tr>
<td>Conv-4</td>
<td>13x13x384</td>
<td>1,327,104</td>
<td>384</td>
<td>1,327,488</td>
</tr>
<tr>
<td>Conv-5</td>
<td>13x13x256</td>
<td>884,736</td>
<td>256</td>
<td>884,992</td>
</tr>
<tr>
<td>MaxPool-3</td>
<td>6x6x256</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FC-1</td>
<td>4096x1</td>
<td>37,748,736</td>
<td>4,096</td>
<td>37,752,832</td>
</tr>
<tr>
<td>FC-2</td>
<td>4096x1</td>
<td>16,777,216</td>
<td>4,096</td>
<td>16,781,312</td>
</tr>
<tr>
<td>FC-3</td>
<td>1000x1</td>
<td>4,096,000</td>
<td>1,000</td>
<td>4,097,000</td>
</tr>
<tr>
<td>Output</td>
<td>1000x1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>62,378,344</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ W_{ff} = \text{Number of weights of a FC Layer which is connected to an FC Layer} \]
\[ B_{ff} = \text{Number of biases of a FC Layer which is connected to an FC Layer} \]
\[ P_{ff} = \text{Number of parameters of a FC Layer which is connected to an FC Layer} \]
\[ F = \text{Number of neurons in the FC Layer} \]
\[ F_{-1} = \text{Number of neurons in the previous FC Layer} \]

\[ W_{ff} = F_{-1} \times F \]
\[ B_{ff} = F \]
\[ P_{ff} = W_{ff} + B_{ff} \]

In the above equation, \( F_{-1} \times F \) is the total number of connection weights from neurons of the previous FC Layer the neurons of the current FC Layer. The total number of biases is the same as the number of neurons (\( F \)).

**Example:** The last fully connected layer of AlexNet is connected to an FC Layer. For this layer, \( F_{-1} = 4096 \) and \( F = 1000 \). Therefore,

\[ W_{ff} = 4096 \times 1000 = 4,096,000 \]
\[ B_{ff} = 1,000 \]
\[ P_{ff} = W_{ff} + B_{ff} = 4,097,000 \]

\[ W_{cf} = \text{Number of weights of a FC Layer which is connected to a Conv Layer} \]
\[ B_{cf} = \text{Number of biases of a FC Layer which is connected to a Conv Layer} \]
\[ O = \text{Size (width) of the output image of the previous Conv Layer} \]
\[ N = \text{Number of kernels in the previous Conv Layer} \]
\[ F = \text{Number of neurons in the FC Layer} \]

\[ W_{cf} = O^2 \times N \times F \]
\[ B_{cf} = F \]
\[ P_{cf} = W_{cf} + B_{cf} \]

**Example:** The first fully connected layer of AlexNet is connected to a Conv Layer. For this layer, \( O = 6 \), \( N = 256 \) and \( F = 4096 \). Therefore,

\[ W_{cf} = 6^2 \times 256 \times 4096 = 37,748,736 \]
\[ B_{cf} = 4096 \]
\[ P_{cf} = W_{cf} + B_{cf} = 37,752,832 \]

GD Optimization Algorithms

Batch Gradient Decent
\[ \theta = \theta - \eta \cdot \nabla_{\theta} J(\theta). \]

Stochastic Gradient Decent
\[ \theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)}). \]

Mini Stochastic Gradient Decent
\[ \theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i:i+n)}; y^{(i:i+n)}). \]

- Choosing a proper learning rate can be difficult, slow convergence or diverge!!
- Learning rate have to be defined in advance and are thus unable to adapt.
- If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima.


https://sites.google.com/view/mlss-2019/lectures-and-tutorials
Momentum \[^{[5]}\] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Image 3. It does this by adding a fraction \(\gamma\) of the update vector of the past time step to the current update vector:

\[
v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta)
\]

\[
\theta = \theta - v_t
\]

Momentum

Nesterov accelerated gradient

\[
v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1})
\]

\[
\theta = \theta - v_t
\]

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We’d like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

Now that we are able to adapt our updates to the slope of our error function and speed up SGD in turn, we would also like to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.

Adagrad uses a different learning rate for every parameter \(\theta_i\) at every time step \(t\), we first show Adagrad’s per-parameter update, which we then vectorize. For brevity, we use \(g_t\) to denote the gradient at time step \(t\). \(g_{t,i}\) is then the partial derivative of the objective function w.r.t. to the parameter \(\theta_i\) at time step \(t\):

\[
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t} + \epsilon} \odot g_t.
\]

\[
\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.
\]

\[
g_{t,i} = \nabla_{\theta} J(\theta_{t,i}).
\]

\[
\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.
\]
RMSProp also tries to dampen the oscillations, but in a different way than momentum. RMSprop also takes away the need to adjust learning rate, and does it automatically. More so, RMSProp chooses a different learning rate for each parameter.

\[
\begin{align*}
\nu_1 &= -G_1 \\
\nu_2 &= -0.9 \times G_1 - G_2 \\
\nu_3 &= -0.9 \times (0.9 \times G_1 - G_2) - G3 = -0.81 \times (G_1) - (0.9) \times G_2 - G_3
\end{align*}
\]

For each Parameter \(w^j\) 

\[
\begin{align*}
\nu_t &= \rho \nu_{t-1} + (1 - \rho) \times g_t^2 \\
\Delta \omega_t &= -\frac{\eta}{\sqrt{\nu_t + \epsilon}} \times g_t \\
\omega_{t+1} &= \omega_t + \Delta \omega_t
\end{align*}
\]

\(\eta\): Initial Learning Rate 
\(\nu_t\): Exponential Average of squares of gradients 
\(g_t\): Gradient at time \(t\) along \(w^j\)
So far, we’ve seen RMSProp and Momentum take contrasting approaches. While momentum accelerates our search in direction of minima, RMSProp impedes our search in direction of oscillations.

**Adam or Adaptive Moment Optimization** algorithms combines the heuristics of both Momentum and RMSProp. Here are the update equations.

For each Parameter $w^j$

\[
\nu_t = \beta_1 \times \nu_{t-1} - (1 - \beta_1) \times g_t \\
s_t = \beta_2 \times s_{t-1} - (1 - \beta_2) \times g_t^2 \\
\Delta \omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} \times g_t \\
\omega_{t+1} = \omega_t + \Delta \omega_t
\]

- $\eta$: Initial Learning rate
- $g_t$: Gradient at time $t$ along $w^j$
- $\nu_t$: Exponential Average of gradients along $w^j$
- $s_t$: Exponential Average of squares of gradients along $w^j$
- $\beta_1, \beta_2$: Hyperparameters
- $\epsilon$: Epsilon

The hyperparameter $beta1$ is generally kept around 0.9 while $beta_2$ is kept at 0.99. Epsilon is chosen to be 1e-10 generally.

https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/
MNIST Example (28x28 pixels)

Innovative Computing Laboratory

neuralnetworksanddeeplearning.com

Training digits and their labels:
5 9 0 1 5 0 4 2 4 5 7 5 0 6 7 5 0 0 8 8 0 6 9
5 9 0 1 5 0 4 2 4 5 7 5 0 6 7 5 0 0 8 8 0 6 9

Validation digits and their labels:
7 2 1 0 4 1 9 5 9 0 6 9 0 1 5 9 7 8 4 9 6 6 5
7 2 1 0 4 1 9 5 9 0 6 9 0 1 5 9 7 8 4 9 6 6 5

Input layer (784 neurons)

Hidden layer (n = 15 neurons)

Output layer

28x28 pixels → 784 pixels

"neurons"

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9
import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential(
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
)

predictions = model(x_train[:1]).numpy()
predictions

tf.nn.softmax(predictions).numpy()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

loss_fn(y_train[:1], predictions).numpy()

model.compile(optimizer='adam', loss=loss_fn, metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

model.evaluate(x_test, y_test, verbose=2)
The model is trained in about 10 seconds completing 5 epochs with a Nvidia GTX 1080 GPU and has an accuracy of around 97-98%
```cpp
#include <cstddef>
#include <cstdio>
#include <vector>
#include <iostream>

/*
  we must include magmadnn as always */
#include "magmadnn.h"

/* tell the compiler we're using functions from the magmadnn namespace */
using namespace magmadnn;

// void print_image(uint32_t image_idx, Tensor<float> *images, Tensor<float> *labels, uint32_t n_rows, uint32_t n_cols);

int main(int argc, char **argv) {
    using T = float;

    // Every magmadnn program must begin with magmadnn_init. This allows magmadnn to test the environment and initialize some GPU data.
    magmadnn_init();

    // Location of the MNIST dataset
    std::string const mnist_dir = ".";
    // Load MNIST training dataset
    magmadnn::data::MNIST<T> train_set(mnist_dir, magmadnn::data::Train);
    magmadnn::data::MNIST<T> test_set(mnist_dir, magmadnn::data::Test);

    // Number of features
    uint32_t n_features;
    // Memory used for training (CPU or GPU)
    memory_t training_memory_type;

    n_features = train_set.nrows() * train_set.ncols();

    /* if you run this program with a number argument, it will print out that number sample on the command line */
    if (argc == 2) {
        train_set.print_image(
            std::atoi(argv[1]));
    }

    // Initialize our model parameters
    model::nn_params_t params;
    params.batch_size = 128; /* batch size: the number of samples to process in each mini-batch */
    params.n_epochs = 10; /* # of epochs: the number of passes over the entire training set */
    params.learning_rate = 0.05;

    // This is only necessary for a general example which can handle all install types. Typically, When you write your own MagmaDNN code you will not need macros as you can simply pass DEVICE to the x_batch constructor.
    #if defined(MAGMADNN_HAVE_CUDA)
        // training_memory_type = DEVICE;
        // std::cout << "Training on GPUs" << std::endl;
        training_memory_type = HOST;
        std::cout << "Training on CPUs" << std::endl;
    #else
        training_memory_type = HOST;
        std::cout << "Training on CPUs" << std::endl;
    #endif
```
Creating the network
// Create a variable (of type T=float) with size (batch_size x n_features) This will serve as the input to our network.
auto x_batch = op::var<T>(
    "x_batch",
    {params.batch_size, train_set.channels(), train_set.rows(), train_set.cols()},
    {NONE, {}}, training_memory_type);

// Initialize the layers in our network
auto input = layer::input(x_batch);
auto pool = layer::pooling(input->out(), {2, 2}, {0, 0}, {2, 2}, AVERAGE_POOL);
auto pool = layer::pooling(input->out(), {2, 2}, {0, 0}, {2, 2}, MAX_POOL);
auto flatten = layer::flatten(pool->out());
auto flatten = layer::flatten(input->out());

auto fc1 = layer::fullyconnected(flatten->out(), 784, false);
auto fc1 = layer::fullyconnected(input->out(), 784, false);
auto act1 = layer::activation(fc1->out(), layer::RELU);
auto act1 = layer::activation(fc1->out(), layer::RELU);

auto fc3 = layer::fullyconnected(act2->out(), train_set.n_classes(), false);
auto act3 = layer::activation(fc3->out(), layer::SOFTMAX);
auto output = layer::output(act3->out());

// Wrap each layer in a vector of layers to pass to the model
std::vector<layer::Layer<float> *> layers =
    {input,
     pool,
     flatten,
     fc1, act1,
     fc2, act2,
     fc3, act3, output};

// This creates a Model for us. The model can train on our data
// and perform other typical operations that a ML model can
// - layers: the previously created vector of layers containing our
// network
model::NeuralNetwork<float> model(layers, optimizer::CROSS_ENTROPY, optimizer::SGD, params);

// model::metric_t records the model metrics such as accuracy, loss, and
// training time
model::metric_t metrics;

/* fit will train our network using the given settings.
X: independent data
y: ground truth
metrics: metric struct to store run time metrics
verbose: whether to print training info during training or not */
model.fit(&train_set.images(), &train_set.labels(), metrics, true);

// Clean up memory after training
delete output;

// Every magmadnn program should call magmadnn_finalize before
// exiting
magmadnn_finalize();
return 0;
Lecture 1 : Overview of Deep Neural Network : MLP, CNN, CUDNN, MKL DNN, TensorFlow, MagmaDNN, MNIST example

Lecture 2 : GPU computing : Linear Algebra, MKL, CUDA, GPU programming, Magma, mixed precision, exercises

Lecture 3 : Programming structure of a DNN framework : I/O, Memory Management, Tensor, layers, activation and fit functions, CFIR 100, MagmaDNN

Lecture 4 : Multiple GPUs and in-depth example exercises : data and model parallelism, distributed Computing, VGG16, ResNet 18, 50, ImageNet.
Linear Algebra Overview
Linear Algebra Libraries
GPU architectures and programming
MAGMA libraries
Performance and benchmarks
Mixed precision
Exercises
• Dense Linear Algebra (DLA) is needed in a wide variety of science and engineering applications, including ML and data analytics problems:

• Linear systems: Solve $Ax = b$
  • Computational electromagnetics, material science, applications using boundary integral equations, airflow past wings, fluid flow around ship and other offshore constructions, and many more

• Least squares: Find $x$ to minimize $||Ax - b||$
  • Computational statistics (e.g., linear least squares or ordinary least squares), econometrics, control theory, signal processing, curve fitting, and many more

• Eigenproblems: Solve $Ax = \lambda x$
  • Computational chemistry, quantum mechanics, material science, face recognition, PCA, data-mining, marketing, Google Page Rank, spectral clustering, vibrational analysis, compression, and many more

• SVD: $A = U \Sigma V^*$ ($Au = \sigma v$ and $A^*v = \sigma u$)
  • Information retrieval, web search, signal processing, big data analytics, low rank matrix approximation, total least squares minimization, pseudo-inverse, and many more

• Many variations depending on structure of $A$
  • $A$ can be symmetric, positive definite, tridiagonal, Hessenberg, banded, sparse with dense blocks, etc.

• DLA is crucial to the development of sparse solvers
LA for modern architectures

- **Leverage latest numerical algorithms and building blocks**
  MAGMA, PLASMA, SLATE (DOE funded), MAGMA Sparse, POMPEI project*

- **Polymorphic approach**
  Use MAGMA sub-packages for various architectures;
  Provide portability through single templated sources using C++

- **Programming model**
  BLAS tasking + scheduling

- **Open standards**
  OpenMP4 tasking + MPI

---

**Use of BLAS for portability**

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
<th>LINPACK (70’s) (Vector operations)</th>
<th>Level 1 BLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LAPACK (80’s) (Blocking, cache friendly)</td>
<td>Level 3 BLAS</td>
</tr>
<tr>
<td></td>
<td>ScaLAPACK (90’s) (Distributed Memory)</td>
<td>PBLAS</td>
</tr>
<tr>
<td></td>
<td>PLASMA (00’s) New Algorithms (many-core friendly)</td>
<td>BLAS on tiles + DAG scheduling</td>
</tr>
</tbody>
</table>

**MAGMA**

Hybrid Algorithms (heterogeneity friendly)
BLAS: Basic Linear Algebra Subroutines

• Level 1 BLAS — vector operations
  – $O(n)$ data and flops (floating point operations)
  – Memory bound: $O(1)$ flops per memory access

• Level 2 BLAS — matrix-vector operations
  – $O(n^2)$ data and flops
  – Memory bound: $O(1)$ flops per memory access

• Level 3 BLAS — matrix-matrix operations
  – $O(n^2)$ data, $O(n^3)$ flops
  – Surface-to-volume effect
  – Compute bound: $O(n)$ flops per memory access
Why Higher Level BLAS?

- By taking advantage of the principle of locality:
- Present the user with as much memory as is available in the cheapest technology.
- Provide access at the speed offered by the fastest technology.
- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this

<table>
<thead>
<tr>
<th>BLAS</th>
<th>Memory Refs</th>
<th>Flops</th>
<th>Flops/ Memory Refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>3n</td>
<td>2n</td>
<td>2/3</td>
</tr>
<tr>
<td>(y = y + \alpha x)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 2</td>
<td>(n^2)</td>
<td>(2n^2)</td>
<td>2</td>
</tr>
<tr>
<td>(y = y + A\mathbf{x})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 3</td>
<td>(4n^2)</td>
<td>(2n^3)</td>
<td>(n/2)</td>
</tr>
<tr>
<td>(C = C + A\mathbf{B})</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Level 1, 2 and 3 BLAS**

Nvidia **P100**, 1.19 GHz, Peak DP = 4700 Gflop/s

Matrix size (N), vector size (NxN)

- C = C + A*B
- y = y + A*x
- y = α *x + y

**Nvidia P100**
Theoretical peak double precision is 4700 Gflop/s
CUDA version 8.0
What about accelerated LA for Data Analytics?

• Traditional libraries like MAGMA can be used as backend to accelerate the LA computations in data analytics applications

• Need support for
  1) New data layouts,  2) Acceleration for small matrix computations,  3) Data analytics tools

Need data processing and analysis support for
Data that is multidimensional / relational

Small matrices, tensors, and batched computations
- Fixed-size batches
- Variable-size batches
- Dynamic batches
- Tensors
MagmaDNN is HP Data Analytics and ML framework built around the MAGMA libraries aimed at providing a modularized and efficient tool for training DNNs.

MagmaDNN makes use of the highly optimized MAGMA libraries giving significant speed boosts over other modern frameworks.
MagmaDNN software stack

**Applications**
High-performance data analytics and machine learning for many-core CPUs and GPU accelerators

**MagmaDNN**
Scalable LA on new architectures
Data abstractions and APIs
Heterogeneous systems portability

**MAGMA Templates**
scalable LA on new architectures
Data abstractions and APIs
Heterogeneous systems portability

**SLATE**
Tile algorithms
LAPACK++
BLAS++

**ScalAPACK API**

**MPI**

**MAGMA (dense)**

**MAGMA Batched**

**MAGMA Sparse**

**Single Heterogeneous Node**

**BLAS API**

**LAPACK API**

**Batched BLAS API**

**OpenMP**

**MKL**

**ESSL**

**cuBLAS**

**ACML**

**Shared memory**

**SVD performance speedup**

**MAGMA 2.3 LU factorization in double precision arithmetic**

**Matrix size N x N**

**Speedup**

- MAGMA-2
- MAGMA
- MKL
- EIGEN

**Matrix sizes**

- 5000
- 10000
- 15000
Outline

Availability

Routines

Code

Testers

Methodology
• Availability
  – http://icl.utk.edu/magma/ download (latest MAGMA 2.5.3), documentation, forum
  – https://bitbucket.org/icl/magma Git repo

• Support
  – Linux, macOS, Windows
  – CUDA >= 7.0; recommend latest CUDA
  – CUDA architecture >= 2.0 (Fermi, Kepler, Maxwell, Pascal, Volta, Turing)
  – BLAS & LAPACK: Intel MKL, OpenBLAS, macOS Accelerate, ...

• May be pre-installed on supercomputers

titan-ext1> module avail magma
---------- /sw/xk6/modulefiles ----------
magma/1.3            magma/1.6.2(default)
Installation options

1. **Makefile**
   - Edit make.inc for compilers and flags (see make.inc examples)
   - `magma> make && make install`

2. **CMake**
   - `magma> mkdir build && cd build`
   - `magma/build> cmake ..` or `ccmake ..`
   - Adjust settings, esp. LAPACK_LIBRARIES and GPU_TARGET
   - `magma/build> make && make install`

3. **Spack**
   - Part of xSDK
   - `spack install magma`
   - Caveat: nvcc is picky about its host compiler (gcc, ...)
Outline
Availability
Routines
Code
Testers
Methodology
MAGMA Overview

• Hybrid LAPACK-style functions
  – Matrix factorizations: LU, Cholesky, QR, eigenvalue, SVD, ...
  – Solve linear systems and linear least squares, ...
  – Nearly all are synchronous:
    return on CPU when computation is finished

• GPU BLAS and auxiliary functions
  – Matrix-vector multiply, matrix norms, transpose (in-place and out-of-place), ...
  – Most are asynchronous:
    return immediately on CPU; computation proceeds on GPU

• Wrappers around CUDA and cuBLAS
  – BLAS routines (gemm, symm, symv, ...)
  – Copy host ↔ device, queue (stream) support, GPU malloc & free, ...

• Batched BLAS and LAPACK

• Sparse LA component
Naming example

- **magma_** or **magmablas_** prefix
- **Precision** (1–2 characters)
  - Single, Double, single Complex, “Z” double complex, Integer
    Mixed precision (DS and ZC)
- **Matrix type** (2 characters)
  - GEneral
  - SYmmetric
  - HErmetian
  - ORthogonal
  - UNitary
  - TRIangular
  - Positive definite
- **Operation** (2–3+ characters)
  - SV: solve
  - TRF: triangular factorization
  - EV: eigenvalue problem
  - GV: generalized eigenvalue problem
  - etc.
- **_gpu** suffix for interface

Example: **magma_zgesv_gpu**
# Linear solvers

- **Solve linear system:** $AX = B$
- **Solve linear least squares:** minimize $\|AX - B\|_2$

<table>
<thead>
<tr>
<th>Type</th>
<th>Routine</th>
<th>Mixed precision</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>dgesv</td>
<td>dsgesv</td>
<td>✓</td>
</tr>
<tr>
<td>Positive definite</td>
<td>dposv</td>
<td>dsposv</td>
<td>✓</td>
</tr>
<tr>
<td>Symmetric</td>
<td>dsyev</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Hermitian</td>
<td>zhesv</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Least squares</td>
<td>dgels</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Selected routines; complete documentation at [http://icl.utk.edu/magma/](http://icl.utk.edu/magma/)
Eigenvalue / singular value problems

- **Eigenvalue problem**: \( Ax = \lambda x \)
- **Generalized eigenvalue problem**: \( Ax = \lambda Bx \) (and variants)
- **Singular value decomposition**: \( A = U \Sigma V^H \)

<table>
<thead>
<tr>
<th>Matrix type</th>
<th>Operation</th>
<th>Routine</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>SVD</td>
<td>dgesvd, dgesdd</td>
<td>✓</td>
</tr>
<tr>
<td>General non-symmetric</td>
<td>Eigenvalue</td>
<td>dgeev</td>
<td>✓</td>
</tr>
<tr>
<td>Symmetric</td>
<td>Eigenvalue</td>
<td>dsysyevd / zheevd</td>
<td>✓</td>
</tr>
<tr>
<td>Symmetric</td>
<td>Generalized</td>
<td>dsysygvd / zhegvd</td>
<td>✓</td>
</tr>
</tbody>
</table>

Additional variants; complete documentation at [http://icl.utk.edu/magma/](http://icl.utk.edu/magma/)
Fastest are divide-and-conquer (gesdd, syevd) and 2-stage versions.
Computational routines

• Computational routines solve one part of problem

<table>
<thead>
<tr>
<th>Matrix type</th>
<th>Operation</th>
<th>Routine</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>LU</td>
<td>dgetrf</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Solve (given LU)</td>
<td>dgetrs</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Inverse</td>
<td>dgetri</td>
<td>✓</td>
</tr>
<tr>
<td>SPD</td>
<td>Cholesky</td>
<td>dpotrf</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Solve (given LLᵀ)</td>
<td>dpotrs</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Inverse</td>
<td>dpotri</td>
<td>✓</td>
</tr>
<tr>
<td>General</td>
<td>QR</td>
<td>dgeqrf</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Generate Q</td>
<td>dorgqr /</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Multiply by Q</td>
<td>dormqr /</td>
<td>✓</td>
</tr>
</tbody>
</table>

Selected routines; complete documentation at [http://icl.utk.edu/magma/](http://icl.utk.edu/magma/)
## BLAS and auxiliary routines

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation</th>
<th>Routine (all GPU interface)</th>
</tr>
</thead>
</table>
| Level 1 BLAS | $y = \alpha x + y$  
$r = x^T y$                                                                 | daxpy                        |
|            |                                                                           | ddot                         |
| Level 2 BLAS | $y = \alpha Ax + \beta y$, general $A$                                    | dgemv                        |
|            | $y = \alpha Ax + \beta y$, symmetric $A$                                  | dsymv                        |
| Level 3 BLAS | $C = \alpha AB + \beta C$                                                 | dgemm                        |
|            | $C = \alpha AB + \beta C$, symmetric $A$                                 | dsymm                        |
|            | $C = \alpha AA^T + \beta C$, symmetric $C$                               | dsyrk                        |
| Auxiliary  | $\|A\|_1$, $\|A\|_{inf}$, $\|A\|_{fro}$, $\|A\|_{max}$                  | dlange (norm, general $A$)   |
|            | $B = A^T$ (out-of-place)                                                   | dlansy (norm, symmetric $A$) |
|            | $A = A^T$ (in-place, square)                                               | dtranspose                   |
|            |                                                                           | dtranspose_inplace           |

Selected routines; complete documentation at [http://icl.utk.edu/magma/](http://icl.utk.edu/magma/)
Matrix layout

General m-by-n matrix, LAPACK column-major layout

- Symmetric / Hermitian / Triangular n-by-n matrix
  - uplo = Lower or Upper
  - Entries in opposite triangle ignored
Simple example

- **Solve** $AX = B$
  - Double precision, GEneral matrix, SolVe (DGESV)
- Traditional LAPACK call

```cpp
#include "magma_lapack.h"
int main( int argc, char** argv )
{
    int n = 100, nrhs = 10;
    int lda = n, ldx = n;
    double *A = new double[ lda*n ];
    double *X = new double[ ldx*nrhs ];
    int* ipiv = new int[ n ];

    // ... fill in A and X with your data
    // A[ i + j*lda ] = A_ij
    // X[ i + j*ldx ] = X_ij

    // solve AX = B where B is in X
    int info;
    lapackf77_dgesv( &n, &nrhs,
                   A, &lda, ipiv,
                   X, &ldx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in X
    delete[] A;
    delete[] X;
    delete[] ipiv;
}```
Simple example

- **MAGMA CPU interface**
  - Input & output matrices in CPU host memory

- Add MAGMA init & finalize

- MAGMA call direct replacement for LAPACK call

```c
#include "magma_v2.h"

int main( int argc, char** argv )
{
    magma_init();

    int n = 100, nrhs = 10;
    int lda = n, ldx = n;
    double *A = new double[ lda*n ];
    double *X = new double[ ldx*nrhs ];
    int* ipiv = new int[ n ];

    // ... fill in A and X with your data
    // A[ i + j*lda ] = A_ij
    // X[ i + j*ldx ] = X_ij

    // solve AX = B where B is in X
    int info;
    magma_dgesv( n, nrhs,
                 A, lda, ipiv,
                 X, ldx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in X
    delete[] A;
    delete[] X;
    delete[] ipiv;

    magma_finalize();
}
```
Simple example

• MAGMA GPU interface
  – Add _gpu suffix
  – Input & output matrices in GPU device memory (“d” prefix on variables)
  – ipiv still in CPU memory
  – Set GPU stride (ldda) to multiple of 32 for better performance
  – roundup returns ceil( n / 32 ) * 32

• MAGMA malloc & free
  – Type-safe wrappers around cudaMalloc & cudaFree

```cpp
int main( int argc, char** argv )
{
    magma_init();

    int n = 100, nrhs = 10;
    int ldda = magma_roundup( n, 32 );
    int lddx = magma_roundup( n, 32 );
    int* ipiv = new int[ n ];

    double *dA, *dX;
    magma_dmalloc( &dA, ldda*n );
    magma_dmalloc( &dX, lddx*nrhs );
    assert( dA != nullptr );
    assert( dX != nullptr );

    // ... fill in dA and dX (on GPU)
    // solve AX = B where B is in X
    int info;
    magma_dgesv_gpu( n, nrhs,
        dA, ldda, ipiv,
        dX, lddx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in dX
    magma_free( dA );
    magma_free( dX );
    delete[] ipiv;
}
```
BLAS example

- Matrix multiply \( C = -AB + C \)
  - Double-precision, GEneral Matrix Multiply (DGEMM)
- Asynchronous
  - BLAS take queue and are async
  - Return to CPU immediately
  - Queue wraps CUDA stream and cuBLAS handle
  - Can create queue from existing CUDA stream and cuBLAS handle, if required

```c
int main( int argc, char** argv )
{
    // ... setup matrices on GPU:
    // m-by-k matrix dA,
    // k-by-n matrix dB,
    // m-by-n matrix dC.

    int device;
    magma_queue_t queue;
    magma_getdevice( &device );
    magma_queue_create( device, &queue );

    // C = -A B + C
    magma_dgemm( MagmaNoTrans,
                 MagmaNoTrans, m, n, k,
                 -1.0, dA, ldda,
                     dB, lddb,
                      1.0, dC, lddc, queue );

    // ... do concurrent work on CPU
    // wait for gemm to finish
    magma_queue_sync( queue );

    // ... use result in dC
    magma_queue_destroy( queue );

    // ... cleanup
}
```
Copy example

- Copy data host ↔ device
  - setmatrix (host to device)
  - getmatrix (device to host)
  - copymatrix (device to device)
  - setvector (host to device)
  - getvector (device to host)
  - copyvector (device to device)

- Default is synchronous
  - Return when transfer is done

- Strides (lda, ldda) can differ on CPU and GPU
  - Set GPU stride (ldda) to multiple of 32 for better performance

```c
int main( int argc, char** argv )
{
    // ... setup A, X in CPU memory;
    // dA, dX in GPU device memory
    int device;
    magma_queue_t queue;
    magma_getdevice( &device );
    magma_queue_create( device, &queue );

    // copy A, X to dA, dX
    magma_dsetmatrix( n, n,
                      A, lda,
                      dA, ldda, queue );
    magma_dsetmatrix( n, nrhs,
                      X, ldx,
                      dX, lddx, queue );

    // ... solve AX = B
    // copy result dX to X
    magma_dgetmatrix( n, nrhs,
                      dX, lddx,
                      X, ldx, queue );

    // ... use result in X
    magma_queue_destroy( queue );
    // ... cleanup
}
```
Async copy

• Add \_async suffix

• Use pinned CPU memory
  – Page locked, so DMA can access it
  – Better performance
  – Required by CUDA for async behavior
  – But pinned memory is limited resource, and expensive to allocate

• Overlap:
  – Sending data (host to device)
  – Getting data (device to host)
  – Host computation
  – Device computation

```c
int main( int argc, char** argv )
{
    // ... setup dA, dX, queue

    // allocate A, X in pinned CPU memory
double *A, *X;
    magma_dmalloc_pinned( &A, lda*n );
    magma_dmalloc_pinned( &X, ldx*nrhs );

    // ... fill in A and X

    // copy A, X to dA, dX, then wait
    magma_dsetmatrix_async( n, n, A, lda, dA, ldda, queue );
    magma_dsetmatrix_async( n, nrhs, X, ldx, dX, lddx, queue );
    magma_queue_sync( queue );

    // ... solve AX = B

    // copy result dX to X, then wait
    magma_dgetmatrix_async( n, nrhs, dX, ldx, X, lddx, queue );
    magma_queue_sync( queue );

    // ... use result in X
    magma_free_pinned( A );
    magma_free_pinned( X );

    // ... cleanup
}
```
Outline

Availability

Routines

Code

Testers

Methodology
Testers: LU factorization (dgetrf)

magma> cd testing
magma/testing> ./testing_dgetrf -n 123 -n 1000:20000:1000 --lapack --check

% MAGMA 2.2.0 compiled for CUDA capability >= 6.0, 32-bit magma_int_t, 64-bit pointer.
% device 0: Tesla P100-PCIE-16GB, 1328.5 MHz clock, 16276.2 MiB memory, capability 6.0

| M  | N  | CPU Gflop/s (sec) | GPU Gflop/s (sec) | |PA-LU|/(N*|A|) |
|----|----|-------------------|-------------------|-------------------|-------------------|
| 123| 123| 0.20 ( 0.01)      | 0.40 ( 0.00)      | 3.59e-18          | ok                |
| 1000| 1000| 10.40 ( 0.06)     | 43.50 ( 0.02)     | 2.76e-18          | ok                |
| 2000| 2000| 111.64 ( 0.05)    | 218.26 ( 0.02)    | 2.68e-18          | ok                |
| 3000| 3000| 288.38 ( 0.06)    | 280.28 ( 0.06)    | 2.65e-18          | ok                |
| 4000| 4000| 305.58 ( 0.14)    | 545.90 ( 0.08)    | 2.81e-18          | ok                |
| 5000| 5000| 396.16 ( 0.21)    | 838.09 ( 0.10)    | 2.71e-18          | ok                |
| 6000| 6000| 413.37 ( 0.35)    | 1088.14 ( 0.13)   | 2.71e-18          | ok                |
| 7000| 7000| 426.71 ( 0.54)    | 1288.60 ( 0.18)   | 2.67e-18          | ok                |
| 8000| 8000| 447.85 ( 0.76)    | 1514.43 ( 0.23)   | 2.66e-18          | ok                |
| 9000| 9000| 461.05 ( 1.05)    | 1621.29 ( 0.30)   | 2.87e-18          | ok                |
| 10000| 10000| 524.06 ( 1.27)   | 1802.39 ( 0.37)   | 2.84e-18          | ok                |
| 11000| 11000| 554.16 ( 1.60)   | 1965.85 ( 0.45)   | 2.84e-18          | ok                |
| 12000| 12000| 559.33 ( 2.06)   | 2090.42 ( 0.55)   | 2.82e-18          | ok                |
| 13000| 13000| 563.56 ( 2.60)   | 2223.62 ( 0.66)   | 2.80e-18          | ok                |
| 14000| 14000| 566.58 ( 3.23)   | 2323.04 ( 0.79)   | 2.78e-18          | ok                |
| 15000| 15000| 567.17 ( 3.97)   | 2431.59 ( 0.93)   | 2.77e-18          | ok                |
| 16000| 16000| 556.86 ( 4.90)   | 2539.66 ( 1.08)   | 2.79e-18          | ok                |
| 17000| 17000| 579.82 ( 5.65)   | 2593.40 ( 1.26)   | 2.75e-18          | ok                |
| 18000| 18000| 584.93 ( 6.65)   | 2694.57 ( 1.44)   | 2.76e-18          | ok                |
| 19000| 19000| 585.78 ( 7.81)   | 2768.67 ( 1.65)   | 2.75e-18          | ok                |
| 20000| 20000| 587.08 ( 9.08)   | 2821.48 ( 1.89)   | 2.74e-18          | ok                |
Testers: LU factorization (dgetrf)
Testers: symmetric matrix-vector multiply (dsymv)

# (abbreviated output)
magma> cd testing
magma/testing> ./testing_dsymv -n 123 -n 1000:20000:1000 --lapack --check
% MAGMA 2.5.0 compiled for CUDA capability >= 6.0, 32-bit magma_int_t, 64-bit pointer.
% device 0: Tesla P100-PCIE-16GB, 1328.5 MHz clock, 16276.2 MiB memory, capability 6.0

% uplo = Lower
% N  MAGMA  Atomics  cuBLAS  CPU  error
%  Gflop/s  Gflop/s  Gflop/s  Gflop/s
%===================================================================

<table>
<thead>
<tr>
<th>N</th>
<th>MAGMA</th>
<th>Atomics</th>
<th>cuBLAS</th>
<th>CPU</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0.76</td>
<td>0.76</td>
<td>0.51</td>
<td>0.58</td>
<td>ok</td>
</tr>
<tr>
<td>1000</td>
<td>27.44</td>
<td>34.41</td>
<td>29.88</td>
<td>8.86</td>
<td>ok</td>
</tr>
<tr>
<td>2000</td>
<td>45.74</td>
<td>70.83</td>
<td>33.91</td>
<td>12.78</td>
<td>ok</td>
</tr>
<tr>
<td>3000</td>
<td>75.30</td>
<td>108.51</td>
<td>40.09</td>
<td>17.76</td>
<td>ok</td>
</tr>
<tr>
<td>4000</td>
<td>100.64</td>
<td>131.23</td>
<td>41.13</td>
<td>17.57</td>
<td>ok</td>
</tr>
<tr>
<td>5000</td>
<td>118.17</td>
<td>162.35</td>
<td>41.46</td>
<td>16.33</td>
<td>ok</td>
</tr>
<tr>
<td>6000</td>
<td>141.21</td>
<td>180.43</td>
<td>42.16</td>
<td>17.55</td>
<td>ok</td>
</tr>
<tr>
<td>7000</td>
<td>157.81</td>
<td>200.44</td>
<td>41.94</td>
<td>19.32</td>
<td>ok</td>
</tr>
<tr>
<td>8000</td>
<td>169.54</td>
<td>198.21</td>
<td>41.78</td>
<td>19.12</td>
<td>ok</td>
</tr>
<tr>
<td>9000</td>
<td>188.40</td>
<td>216.07</td>
<td>42.28</td>
<td>21.50</td>
<td>ok</td>
</tr>
<tr>
<td>10000</td>
<td>195.92</td>
<td>224.50</td>
<td>42.36</td>
<td>17.44</td>
<td>ok</td>
</tr>
<tr>
<td>11000</td>
<td>214.93</td>
<td>237.51</td>
<td>45.91</td>
<td>21.30</td>
<td>ok</td>
</tr>
<tr>
<td>12000</td>
<td>219.33</td>
<td>233.44</td>
<td>45.76</td>
<td>20.81</td>
<td>ok</td>
</tr>
<tr>
<td>13000</td>
<td>217.52</td>
<td>241.45</td>
<td>42.49</td>
<td>21.29</td>
<td>ok</td>
</tr>
<tr>
<td>14000</td>
<td>231.26</td>
<td>249.06</td>
<td>45.84</td>
<td>19.71</td>
<td>ok</td>
</tr>
<tr>
<td>15000</td>
<td>232.12</td>
<td>255.98</td>
<td>45.87</td>
<td>19.60</td>
<td>ok</td>
</tr>
<tr>
<td>16000</td>
<td>239.26</td>
<td>250.89</td>
<td>45.58</td>
<td>22.61</td>
<td>ok</td>
</tr>
<tr>
<td>17000</td>
<td>240.74</td>
<td>257.13</td>
<td>45.69</td>
<td>23.15</td>
<td>ok</td>
</tr>
<tr>
<td>18000</td>
<td>242.45</td>
<td>265.05</td>
<td>45.75</td>
<td>19.09</td>
<td>ok</td>
</tr>
<tr>
<td>19000</td>
<td>242.53</td>
<td>262.48</td>
<td>45.81</td>
<td>22.42</td>
<td>ok</td>
</tr>
<tr>
<td>20000</td>
<td>239.53</td>
<td>258.24</td>
<td>45.63</td>
<td>22.83</td>
<td>ok</td>
</tr>
</tbody>
</table>
Testers: symmetric matrix-vector multiply (dsymv)
Test everything: run_tests.py

• Python script to run:
  – All testers
  – All possible options (left/right, lower/upper, ...)
  – Various size ranges (small, medium, large; square, tall, wide)

• Occasionally, tests fail innocuously
  – E.g., error = 1.1e-15 > tol = 1e-15

• Some experimental routines are known to fail
  – E.g., gegqr_gpu, geqr2x_gpu
  – See magma/BUGS.txt

• Running ALL tests can take > 24 hours
Test everything: run_tests.py

magma/testing> python ./run_tests.py *trsm --xsmall --small > trsm.txt

testing_strsm -SL -L -DN -c ok  # left, lower, non-unit, [no-trans]
testing_dtrsm -SL -L -DN -c ok

testing_ctrsm -SL -L -DN -c ok

testing_ztrsm -SL -L -DN -c ok

testing_strsm -SL -L -DU -c ok  # left, lower, unit, [no-trans]
testing_dtrsm -SL -L -DU -c ok

testing_ctrsm -SL -L -DU -c ok

testing_ztrsm -SL -L -DU -c ok

testing_strsm -SL -L -C -DN -c ok  # left, lower, non-unit, conj-trans

testing_dtrsm -SL -L -C -DN -c ok

testing_ctrsm -SL -L -C -DN -c ok

testing_ztrsm -SL -L -C -DN -c ok

... 
testing_strsm -SR -U -T -DU -c ok  # right, upper, unit, trans

testing_dtrsm -SR -U -T -DU -c ok

testing_ctrsm -SR -U -T -DU -c ok

testing_ztrsm -SR -U -T -DU -c ok

******************************************************************************

summary
******************************************************************************

6240 tests in 192 commands passed
96 tests failed accuracy test
0 errors detected (crashes, CUDA errors, etc.)

routines with failures:
  testing_ctrsm --ngpu 2 -SL -L -C -DN -c
  testing_ctrsm --ngpu 2 -SL -L -C -DU -c

...
One-sided factorizations

- LU, Cholesky, QR factorizations for solving linear systems

![DAG diagram](image)

- Level 2 BLAS on CPU
- Level 3 BLAS on GPU
- Panel

Critical path
Execution trace

- Panels on CPU (green) and set/get communication (brown) overlapped with trailing matrix updates (teal) on GPU

- Goal to keep GPU busy all the time; CPU may idle

LU factorization (dgetrf), n = 20000
P100 GPU, 2 x 10-core 2.3 GHz Haswell

- Optimization: for LU, we transpose matrix on GPU so row-swaps are fast
Two-sided factorizations

- Hessenberg, tridiagonal, bidiagonal factorizations for eigenvalue and singular value problems

\[ y_i = A v_i \]

Panel

Trailing matrix

\[ A = Q^T A Q \]

column \( a_j \)

Level 2 BLAS on CPU

Level 2 BLAS on GPU

Level 3 BLAS on GPU
Numerical Linear Algebra (NLA) in Applications

- For **big** NLA problems
  (BLAS, convolutions, SVD, linear system solvers, etc.)

- Adding in MAGMA application backends for **small** problems

- Machine learning / DNNs
- Data mining / analytics
- High-order FEM,
- Graph analysis,
- Neuroscience,
- Astrophysics,
- Quantum chemistry,
- Signal processing, and more

### Large matrices

- In contemporary libraries:
  - BLAS
  - LAPACK
  - ScaLAPACK
  - **MAGMA** (for GPUs)

### Small matrices / tensors

- Fixed-size batches
- Variable-size batches
- Dynamic batches
- Tensors
Accelerating NN – express with GEMMs of various sizes

- Convolutions can be accelerated in various ways:
  - Unfold and GEMM
  - FFT
  - Winograd minimal filtering – reduction to batched GEMMs

- Use autotuning to handle complexity of tuning

---

Table: Fast Convolution

<table>
<thead>
<tr>
<th>Layer</th>
<th>( m )</th>
<th>( n )</th>
<th>( k )</th>
<th>( M )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12544</td>
<td>64</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>12544</td>
<td>64</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>12544</td>
<td>128</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>12544</td>
<td>128</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6272</td>
<td>256</td>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>6272</td>
<td>256</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6272</td>
<td>256</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>3136</td>
<td>512</td>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>3136</td>
<td>512</td>
<td>512</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>3136</td>
<td>512</td>
<td>512</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>784</td>
<td>512</td>
<td>512</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>784</td>
<td>512</td>
<td>512</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>784</td>
<td>512</td>
<td>512</td>
<td>16</td>
</tr>
</tbody>
</table>

---

Require matrix-matrix products of various sizes, including batched GEMMs
MAGMA Batched Computations

1. Non-batched computation
   - loop over the matrices one by one and compute using multithread (note that, since matrices are of small sizes there is not enough work for all the cores). So we expect low performance as well as threads contention might also affect the performance

   ```
   for (i=0; i<batchcount; i++)
   dgemm(...)
   ```
1. Batched computation

Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

- For very small matrices, assign a matrix/core (CPU) or per TB for GPU
- For medium size a matrix go to a team of cores (CPU) or many TB’s (GPU)
- For large size switch to multithreads classical 1 matrix per round.

Batched_dgemm(...)
How to implement fast batched DLA?

Problem sizes influence algorithms & optimization techniques

Kernels are designed various scenarios and parameterized for autotuning framework to find “best” performing kernels

Optimizing GEMM’s: Kernel design

- Reading/writing the elements is based on the TB size (# threads) and so is an extra parameter.
- Also it could be different for A, B and C
MAGMA Mixed Precision algorithms

Idea: use lower precision to compute the expensive flops \((LU \ O(n^3))\) and then iteratively refine the solution in order to achieve the FP64 arithmetic

- Achieve higher performance \(\rightarrow\) faster time to solution
- Reduce power consumption by decreasing the execution time \(\rightarrow\) Energy Savings !!!

Reference:

   Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers,

2. Haidar, A., Tomov, S., Dongarra, J., Higham, NJ,
   Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers,
LU factorization is used to solve a linear system $Ax=b$.

The system can be split into:

1. Lower Triangular System
   
   $$Ly = b$$

2. Upper Triangular System
   
   $$Ux = y$$

With matrix size:

- **FP16 hgetrf LU factorization Tensor Cores**
- **FP16 hgetrf LU factorization**
- **FP32 sgetrf LU factorization**
- **FP64 dgetrf LU factorization**

**Motivation**

Leveraging Half Precision in HPC on V100

Study of the LU factorization algorithm on Nvidia V100

```
A \quad x = b

LUx = b

Ly = b

then

Ux = y
```
Leveraging Half Precision in HPC on V100
Performance Behavior

Performance of solving Ax=b
using FP64 or IR with GMRes to achieve FP64 accuracy

- solving Ax = b using FP64 LU
- solving Ax = b using FP32 LU and iterative refinement to achieve FP64 accuracy
- solving Ax = b using FP16 LU and iterative refinement to achieve FP64 accuracy
- solving Ax = b using FP16 Tensor Cores LU and iterative refinement to achieve FP64 accuracy

Flops = $2n^3/(3\text{ time})$ meaning twice higher is twice faster
Examples / Exercises

• Every routine has an example / benchmark
• Examples / benchmarks are in directory “testing”
  
  > cd testing

• To generate a particular test, e.g., matrix-matrix product
  
  > make testing_dgemm
Examples / Exercises

• Solving a linear system of equations

\[ Ax = b \]

#include "magma_v2.h"
#include "magma_lapack.h"

magma_init();

... double *hA, *hB; // A is a typical array on the CPU
magma_dmalloc_pinned(&hA, lda * N); // A can be allocated in pinned memory
magma_dmalloc_cpu( &hB, nrhs* M );
...
init_matrix(opts, M, N, hA, lda);
...

magma_dgetrf( M, N, hA, lda, ipiv, &info); // LU factorization (using CPU+GPU)
lapackf77_dgetrs( MagmaNoTransStr, &N, &nrhs, hA, &lda, ipiv, hB, &ldb, &info ); // Solve on CPU with LAPACK

Alternatively, there is a direct MAGMA function to solve:
see testing_dgesv.cpp and testing_dgetrf.cpp
Examples / Exercises

• Solving an eigenvalue problem
  \[ Ax = \lambda x \]

```c
#include "magma_v2.h"
#include "magma_lapack.h"

magma_init();
...
double *hA;
magma_dmalloc_pinned( &hA, lda * N);
...
magma_dsyevd( opts.jobz, opts.uplo,
             N, hA, lda, w1,
             h_work, lwork,
             iwork, liwork, &info );

see testing_dsyevd.cpp
```
## Examples / Exercises

- **Writing a hybrid algorithm**

  Develop a hybrid CPU-GPU algorithm for this algorithm (Cholesky QR for a “tall-and-skinny” matrix $A$)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>MATLAB</th>
<th>Hybrid CPU-GPU algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G = A^T A$</td>
<td>$G = A'*A; $</td>
<td>cublasDgemm( ... );</td>
</tr>
<tr>
<td></td>
<td></td>
<td>magma_dgetvector( ... );</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// Send $G$ from GPU to CPU memory,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// e.g. hG</td>
</tr>
<tr>
<td>$G = LL^T$</td>
<td>$U = \text{chol}(G); $</td>
<td>dpotrf_( ... )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>magma_dsetvector( ... );</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// LAPACK on CPU to do Cholesky on hG</td>
</tr>
<tr>
<td>$Q = A (L^T)^{-1}$</td>
<td>$Q = A * \text{inv}( U )$</td>
<td>magma_dtrsm( ... );</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// Triangular matrix solve on the GPU</td>
</tr>
</tbody>
</table>

- First check that indeed $A = QR$, where $Q$ is orthogonal ($Q^T Q = I$)
- Compare your solution to the one implemented in dgegqr_gpu.cpp
CUDA references

- NVIDIA CUDA Programming Guide

- CUDA Samples
  Typically installed with CUDA, e.g., in `/usr/local/cuda/samples`
CUDA Software Stack

(Source: NVIDIA CUDA Programming Guide)
CUDA Programming Model

- **Grid of thread blocks**
  (blocks of the same dimension, grouped together to execute the same kernel)

- **Thread block**
  (a batch of threads with fast shared memory executes a kernel)

- **Sequential code launches asynchronously GPU kernels**

```c
// set the grid and thread configuration
Dim3 dimGrid(2, 3);
Dim3 dimTBlock(3, 5);

// Launch the device computation
MatVec<<<dimGrid, dimTBlock>>>(...);

__global__ void MatVec(...) {
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    ...
}
```
Hello World!

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no device code

**Output:**

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$
```
Hello World!
with Device Code

```c
__global__ void mykernel(void) {
    printf("Hello World!\n");
}

int main(void) {
    mykernel<<<1,1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```
Hello World!
with Device Code

```c
__global__ void mykernel(void) {
    printf("Hello World from block %d, thread %d!\n", blockIdx.x, threadIdx.x);
}

int main(void) {
    mykernel<<<10,10>>>();
    cudaDeviceSynchronize();
    return 0;
}
```
Parallel Programming in CUDA C/C++

• But wait... GPU computing is about massive parallelism!

• We need a more interesting example...

• We’ll start by adding two integers and build up to vector addition

...
Handling Arbitrary Vector Sizes

• Typical problems are not friendly multiples of $\text{blockDim.x}$

• Avoid accessing beyond the end of the arrays:

```c
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

• Update the kernel launch:

```c
add<<<(N + M-1) / M, M>>>(d_a, d_b, d_c, N);
```
Small red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads.

SGEMM Example

http://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf
GEMM in MAGMA


- Add register blocking
- Parameterized for autotuning for particular size GEMMs and portability across GPUs

*Small* red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads

A thread computes part of a row (16 values) of the C block

A thread computes a block of C (4 x 4 values in this case)
Implementing a DNN Framework

Daniel Nichols
Before You Start Writing Code

- **Why?**
  - **High Performance**
    - C++, C++/Python
    - Hardware accelerators
  - **Portability**
    - Python, C++/Python
    - Containers
  - **Minor Feature**
    - Extend existing open source framework
- **Design**
  - Write example use cases
  - Design with modularity in mind
Goals

Easy to use tensors and file I/O:
Tensor dataSet = io::from_csv("data.csv");

Fast tensor math backend:
Tensor A = ..., B = ...
Tensor C = math::matmul(A, B);

Compute graph:
Op weights = ..., input = ..., bias = ...
Op outputOp = op::add(op::matmul(weights, input), bias);
Tensor output = outputOp->evaluate();

Gradient computation:
GradientTable grads(outputOp);
grads.grad(outputOp, weights); /*wrt weights*/

Optimize compute graphs:
SGDOptimizer optim(0.01);
optim.minimize(outputOp, weights);

Model construction:
Layer fc0 = layer::dense(512);
Layer fc1 = layer::dense(fc0, 128);
Layer fc2 = layer::dense(fc1, 10);
Model network({fc0, fc1, fc2}, SGD, ...);

network.fit(trainDataset);
DNN Workflow

1. Load Data
2. Preprocessing
3. Create/Load Model
4. Train Model
   - Export Model
   - Predict
Software Structure of a DNN Framework

- Core
  - Systems
    - Memory (CPU, GPU, Hybrid)
    - Communication (MPI, OpenMP)
  - Math
    - Tensor data structures and operations
    - Blas/Lapack, Magma
    - DNN accelerator (CuDNN, hipDNN, OneDNN)
- Frontend
  - Compute Graph
  - Optimizers
  - Models
MagmaDNN's Structure

- Model/Network
- Compute Graph
- Operation
  - Tensor Math
  - BLAS/LAPACK/MAGMA
  - CuDNN/MKLDNN
  - Tensor
  - MemoryManager
  - MPI Wrapper
Memory Management

Goals

- Abstraction
  - Provide basic memory operations for supported hardware
    - copy, allocate, free, put, get
  - Support inter-device operations for all device combinations

- Virtualization
  - Emulate capabilities not supported by hardware
    - synchronized memory, async prefetching

- Optimized
  - Parameterized for auto-tuning
class MemoryManager {
    MemoryManager(size_t n) {...}
    ~MemoryManager() {...}

    void copy(MemoryManager const& m){.}
    void swap(MemoryManager& m){...}

    template <class T>
    void set(T *arr) {...}

    template <class T>
    T* get() {...}

    void setDevice(device_t dev) {...}
    device_t getDevice() {...}
};

Interface
- Simple user exposed interface
- Modern C++ style code
  - Pointers are necessary for Memory Management
- Can use helper classes to support combinatorial operations
Memory Manager

```cpp
/* (1) */
MemoryManager<T>(size_t n): data_(new T[n]) {}

/* (2) */
MemoryManager(size_t n, data_t type) {
    switch (type) {
        case FLOAT:
            this->data_ = new float[n];
            ...
    }
}

/* (3) */
MemoryManager(size_t n, data_t type) {
    this->data_ =
        (void*) new uint8_t[n * numBytes(type)];
}
```

Three ways to store typed data
- **(1) Generics/Templates**
  - New class for every type
  - Types must be known at compile time
- **(2) Switch on every type**
  - If `data_t` is enumerated, then jump tables are used
- **(3) Binary Large Object (BLOB)**
  - Store bytes and type information separately
Memory Manager

```cpp
auto cudaPtr = [](size_t s) { void *p;
          cudaMalloc((void **) &p, s); return p; };

auto deleter = [](float *p) { cudaFree(p); };

std::unique_ptr<float[]>, decltype(deleter)>(
  (float *) cudaPtr(len*sizeof(float)), deleter);
```

Make use of language features

- **std::unique_ptr<T>**
  - Since C++11
  - Frees when object leaves scope
  - Implements proper Copy/Swap/Move semantics
- **std::iterator**
  - Allows the use of std functions
  - Can use pointers as iterators
Optimizations

- Asynchronous Operations
  - Move next set of memory into GPU while other operations are still completing

- Tunable parameters
  - Block Size
  - Buffering
  - Can be tuned to hardware at library compile time
Communication

- Memory
  - Move memory between devices/nodes seamlessly
- Math Operations
  - Allreduce, Reduce
- Implementation
  - MPI Wrappers
  - Hidden implementation
  - Well optimized
  - Error checking
**Communication**

Abstract Send/Receive Paradigm
- Make communication details implicit
- Allow explicit customization
- ::asyncCopy(...)
  - MPI_ISend(...)
  - MPI_IRecv(...)
- Check for errors!

Node $i$

```cpp
MPI_Send(data_, this->size(), ...);
```

Node $j$

```cpp
MPI_Recv(data_, this->size(), ...);
```

![Diagram of communication between nodes $i$ and $j$ with memory exchange through MemoryManager::copy(...)]
Communication

Important Collectives

- AllReduce(*), Reduce, Bcast, AllGather, ReduceScatter
- Use most optimal
  - MPI, Cuda-aware MPI
  - Horovod(*)
  - NCCL1 (intra-node), NCCL2 (inter-node)
- Use RDMA and GPU Direct where applicable
Implementation
- Wrappers for communication primitives
  - Send, Recv, Reduce, AllReduce, AllGather, Bcast, ReduceScatter
  - Allows easy change of backend (MPI, Horovod, etc...)
- Optimized
  - Tune MPI to system hardware
  - Use state-of-the-art techniques
    - Toroidal-Reduce, Pipelined Communication, Asynchronous Updates
- Error Check
  - Handle and recover from node failures
  - MPI is not fault tolerant!
Tensors

Data Structure
- Multi-Dimensional Wrapper around MemoryManager
- Conceptually $N$-dimensional, but should support 1-8 dimensions
- Equivalent to NumPy’s Ndarray
- Support multiple data layouts
Tensors

class Tensor {
    public:
        Tensor(vector<size_t> const& shape) {}
        Tensor(initializer_list<size_t> ...) {}

        template <class T>
        void set(vector<size_t> const& idx, T val) {...}
        template <class T>
        T get(vector<size_t> const& idx) {...}
    private:
        MemoryManager memoryManager_;
        data_t dtype_;

        size_t getFlattenedIndex(vector<size_t> const& idx) { ... }
};

Implementation
- Use memory manager to store data
- Tensor class provides structure to linear memory
- Support several data layouts
  - Stride from leading dimension
  - Stride from trailing dimension
  - Tiled
- Shallow vs Deep Copies
Tensor Math

Design
- Provide commonly used math operations for tensors
- Support all hardware
- Use modular backend

Implementation
- Instantiate versions of functions based on available libraries and hardware
- Clean wrappers for underlying libraries (BLAS, LAPACK, etc...)
Tensor Math

template <device_t dev>
Tensor matmul<dev>(Tensor const& A, Tensor const& B);

template <>
Tensor matmul<CPU>(Tensor const& A, Tensor const& B) {
    if (A.dtype() == FLOAT_T) {
        cblas_sgemm(...);
    } else ...
}

template <>
Tensor matmul<GPU>(Tensor const& A, Tensor const& B) {
    if (A.dtype() == FLOAT_T) {
        magma_sgemm(...);
    } else ...
}

Instantiate by Hardware/Accelerator
- If C++, use templates
- Only compiles if available

Modular Design
- Multiple Backends
- Easy to switch drivers
  - MagmaBLAS, CuBLAS, CBLAS, MKL BLAS
  - oneDNN, hipDNN, cuDNN
Common Tensor Math Functions

**Matmul** - math::matmul(A, B, C)

**Dot** - math::dot(A, B) *(gemv or gemm)*

**Reduce** - math::reduce(A, Operation, axis)
- Min, Max
- Sum
- Product
- ArgMin, ArgMax

**Elementwise Unary Func** - math::unary(A, Op)
- Negate
- Invert (element-wise)
- Square
- Softmax
- ReLU, Leaky ReLU
- Tanh
- Sigmoid

**Elementwise Binary Func** - math::binary(A, B, Op)
- Add
- Product (hadamard)
- Cross Entropy

**Norm** - math::norm(A, type)

**Convolutions** -
- math::conv1d(data, filter, params…)
- math::conv2d(data, filter, params…)
- math::conv3d(data, filter, params…)

**Pooling** - math::pooling2d(data, AVG/MAX, …)
Tensor Math

Optimizations

- Use best backends
  - Tune at compile time
  - Modular design
  - Choose based on data size/type
- Minimize call path to core function
- Use table or state machine to give most optimal function

template <>
Tensor matmul<CPU>(Tensor const& A, Tensor const& B) {
    if (A.dtype() == FLOAT_T
        && isTallSkinny(A)
        && A.isCuda()) {
        magma_sgemm(...);
    } else if (A.dtype == DOUBLE_T
               && A.isCPU()
               && A.dataLayout() == TILE_LAYOUT) {
        slate::gemm(...);
    } else ...
}
Tensor Math

cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t handle,
    const void *alpha,
    const cudnnTensorDescriptor_t xDesc,
    const void *x,
    const cudnnTensorDescriptor_t dyDesc,
    const void *dy,
    const cudnnConvolutionDescriptor_t convDesc,
    cudnnConvolutionBwdFilterAlgo_t algo,
    void *workSpace,
    size_t workSpaceSizeInBytes,
    const void *beta,
    const cudnnFilterDescriptor_t dwDesc,
    void *dw
)

DNN Libraries
- CuDNN, OneDNN, hipDNN
- Best Use
  - Only call core routines in training
  - Preallocate as much memory as possible
  - Error Check!
- Profile, profile, profile...
Tensor Math

**CUDNN, OneDNN, hipDNN**

- Best Use
  - Only call core routines in training
  - Preallocate as much memory as possible
  - Error Check!
- Profile, profile, profile...

```c
#include <cudnn.h>

// CUDNN convolution backward filter function

/_mozlayercontent_919034_1534046285021_2

cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t handle,
    const void *alpha,
    const cudnnTensorDescriptor_t xDesc,
    const void *x,
    const cudnnTensorDescriptor_t dyDesc,
    const void *dy,
    const cudnnConvolutionDescriptor_t convDesc,
    cudnnConvolutionBwdFilterAlgo_t algo,
    void *workSpace,
    size_t workSpaceSizeInBytes,
    const void *beta,
    const cudnnFilterDescriptor_t dwDesc,
    void *dw
)
```
Tensor Math

- Only call core routines in as possible
- Error Check!
- Profile, profile, profile...

```c
cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t handle,
    const void *alpha,
    const cudnnTensorDescriptor_t xDesc,
    const void *x,
    const cudnnTensorDescriptor_t dyDesc,
    const void *dy,
    const cudnnConvolutionDescriptor_t convDesc,
    cudnnConvolutionBwdFilterAlgo_t algo,
    void *workSpace,
    size_t workspaceSizeInBytes,
    const void *beta,
    const cudnnFilterDescriptor_t dwDesc,
    void *dw
)
```
Tensor Math

- **Only call core routines in as possible**

```c
#include <cudnn.h>

// cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t handle,
    const void *alpha,
    const cudnnTensorDescriptor_t xDesc,
    const void *x,
    const cudnnTensorDescriptor_t dyDesc,
    const void *dy,
    const cudnnConvolutionDescriptor_t convDesc,
    cudnnConvolutionBwdFilterAlgo_t algo,
    void *workSpace,
    size_t workSpaceSizeInBytes,
    const void *beta,
    const cudnnFilterDescriptor_t dwDesc,
    void *dw)

void cudnnCreate(cudnnHandle_t *handle)

void cudnnCreateTensorDescriptor(cudnnTensorDescriptor_t *tensorDesc)

void cudnnCreateTensorDescriptor(cudnnTensorDescriptor_t *tensorDesc)

void cudnnCreateConvolutionDescriptor(cudnnConvolutionDescriptor_t *convDesc)
```
- Math

- DNN Libraries
  - CuDNN, OneDNN, hipDNN

- Best Use
  - Only call core routines in training
  - Preallocate as much memory as possible

- Error Check!
  - Profile, profile, profile...

```c
#include <cudnn.h>

extern "C" {
    cudnnStatus_t cudnnCreate(cudnnHandle_t *handle)
    cudnnStatus_t cudnnCreateTensorDescriptor(cudnnTensorDescriptor_t *tensorDesc)
    cudnnStatus_t cudnnCreateConvolutionDescriptor(cudnnConvolutionDescriptor_t *convDesc)
}
```

- cudnnGetConvolutionBackwardDataAlgorithm_v7(
  cudnnHandle_t handle,
  const cudnnFilterDescriptor_t *wDesc,
  const cudnnTensorDescriptor_t *dyDesc,
  const cudnnConvolutionDescriptor_t *convDesc,
  const cudnnTensorDescriptor_t *dxDesc,
  const int requestedAlgoCount,
  int *returnedAlgoCount,
  cudnnConvolutionBwdDataAlgo_t *algo,
  void *workSpace,
  size_t workSpaceSizeInBytes,
  const void *beta,
  const cudnnFilterDescriptor_t *dwDesc,
  void *dw
)
- **Tensor Math**

- **DNN Libraries**
  - CuDNN,
  - OneDNN,
  - hipDNN

- **Best Use**
  - Only call core routines in training
  - Preallocate as much memory as possible
  - Error Check!

```c
void cudnnCreateFilterDescriptor(cudnnFilterDescriptor_t *filterDesc)
```

```c
void cudnnCreateTensorDescriptor(cudnnTensorDescriptor_t *tensorDesc)
```

```c
void cudnnCreateConvolutionDescriptor(cudnnConvolutionDescriptor_t *convDesc)
```

```c
cudnnStatus_t cudnnGetConvolutionBackwardDataWorkspaceSize(cudnnHandle_t handle,
const cudnnFilterDescriptor_t wDesc,
const cudnnTensorDescriptor_t dyDesc,
const cudnnConvolutionDescriptor_t convDesc,
const cudnnTensorDescriptor_t dxDesc,
cudnnConvolutionBwdDataAlgo_t algo,
size_t *workSpace, size_t *workSpaceSizeInBytes)
```

```c
cudnnStatus_t cudnnGetConvolutionBackwardDataAlgorithm_v7(cudnnHandle_t handle,
const cudnnFilterDescriptor_t wDesc,
const cudnnTensorDescriptor_t dyDesc,
const cudnnConvolutionDescriptor_t convDesc,
const cudnnTensorDescriptor_t xDesc,
const cudnnConvolutionDescriptor_t dxDesc,
cudnnConvolutionBwdFilterAlgo_t algo,
void *alpha, const void *x,
void *dy, const void *y,
void *dw, const void *w,
const int requestedAlgoCount,
int *returnedAlgoCount,
cudnnConvolutionBwdDataAlgoPerf_t *perfResults)
```

```c
cudnnStatus_t cudnnCreate(cudnnHandle_t *handle)
```

```c
cudnnStatus_t cudnnCreateTensorDescriptor(cudnnTensorDescriptor_t *tensorDesc)
```

- **Innovative Computing Laboratory**
Frontend

- Core (memory, tensors, communication, math kernels)
  - Completely separate from frontend
  - Correct & Fast
- Frontend (Compute Graph, Optimizers, Models)
  - Hardware agnostic
  - Clean interface
  - Use core API
Compute Graph

- Represents training control flow/math operations
- Directed Acyclic Graph
- Eager vs. Non-Eager
- Represents every computation

\[ \sigma(W.X + b) \]
class Operation {
    virtual Tensor eval() = 0;

    /* Compute gradient wrt x
       G is incoming backprop gradient */
    virtual Tensor grad(Operation *x, Tensor const& G) = 0;

    vector<Operation*> children_;
    Operation *parent_;
};

Base Class
- *Operation* in MagmaDNN
- Offers eval/grad functions
- Abstract
class MatmulOp : public Operation {
  MatmulOp(Operation *A, Operation *B) :
    children_({A,B}), A_(A), B_(B) {}

  Tensor eval() {
    return math::gemm(A->eval(), B->eval());
  }

  Tensor grad(Operation *x, Tensor const& G) {
    if (x == A_)
      return math::gemm(G, B_->eval(), B_TRANSPOSE);
    else
      return math::gemm(A_->eval(), G, A_TRANSPOSE);
  }
private:
  Operation *A_, *B_;}

Example Matmul Operation

eval()
- Evaluates children
- Returns tensor with computed result

grad()
- Computes gradient wrt an input
- Used in backpropagation algorithm
Compute Graph

Usability
- Define wrapper functions to construct nodes behind scene
- Pointers necessary for Data Structures
- Use C++ to manage memory

/* (1) */
Operation *matmul(Operation *A, Operation *B) {
    return new MatmulOp(A, B);
}

/* (2) */
typedef std::unique_ptr<Operation> Op;
Op&& matmul(Op A, Op B) {
    return std::make_unique<MatmulOp>(A.get(), B.get());
}
Compute Graph

Tensor xTensor, wTensor, bTensor;

auto x = var(xTensor);
auto w = var(wTensor);
auto b = var(bTensor);

auto result = add(matmul(w, x), b);
Tensor resultTensor = result->eval();

Usability
- Create Variable operation as entry points to graph
- End user can now construct graph easily with wrapper functions
- Must eval to get final result
Optimizations
- Mixed Precision

\[ \text{gemm} + \text{b(L)} \]

\[ \text{FP32 to FP16} \]

\[ \text{FP32 to FP16} \]

\[ \text{W(L)} \]

\[ \text{H(L)} \]
Compute Graph

Optimizations
- Fused Operations
Optimizations
- All optimizations
- Tunable

Compute Graph

\[
\sigma \rightarrow \text{gemm} \rightarrow W(L) \oplus H(L) \rightarrow \text{gemm} \rightarrow \text{BIAS AND ACTIVATE} \rightarrow b(L)
\]
map<Op, Tensor> getGradTable(x, G, T):
  map<Op, Tensor> gradTable
  gradTable[x] = 1
  for operation in Graph:
    setGradient(operation, Graph, trainable, gradTable)
  return gradTable

setGradient(x, G, T, table):
  grad = table[x]
  if grad exists:
    return grad
  G = 0
  for consumer in x.consumers():
    setGradient(child, G, T, table)
    G = G + consumer->grad(x, grad)
  table[x] = G

Computing Gradients
- Simplified version
- Dynamic Programming
- Only works for 1 output

Other Implementations and Discussions
- See “Deep Learning” by Ian Goodfellow
- See “autograd” or “jax”
Optimizer

Interface
- Minimize compute graph node w.r.t. an input node
- ex. \( \text{min } \text{pow}(X, 2) \) w.r.t. \( x \)
- Support iterative methods

Abstraction
- Generic optimizer class used by fit functions
- Support many types SGD, SGD+Nesterov Momentum, Adam, RMSProp, ...
- Distributed Optimizers

Implementation
- Use existing tensor math core
- Easy hyperparameter manipulation
Optimizer

Example Use Case
- Minimize any compute graph
- Easy for end user

auto x = op::var<float>("x", NONE);
auto c = op::var<float>("c", {CONSTANT, -2.0f});

optimizer::GradientDescent opt(0.05);

opt.minimize( op::add(op::pow(x, 2), c), {x});
class Optimizer {
    Optimizer(params...) {}

    virtual void minimize(Op x, Op wrt) {
        for (1 ... limit) {
            step(x, wrt);
        }
    }

    virtual void step(Op x, Op wrt) = 0;
};

The Interface
- “Minimize”
  - Where to implement minimization routine
  - Minimizes x with respect to wrt
- “Step”
  - A single step for iterative methods
- Use Grad Table to minimize for gradient based methods
class SGD : public Optimizer {
    Optimizer(float lr): lr_(lr) {}

    virtual void step(Op x, Op wrt) override {
        auto grads = getGradTable(x, wrt,{x});

        math::sgd_update(x, grads[wrt], lr_);
    }
};

An Example Optimizer -- SGD
- Use grad table to get gradients
- Math Core kernels to update
- Use a learning rate
- Only need to override step function
Optimizer

**Easy Change To Run Distributed**
- User only has to change optimizer type
- Uses communication backend
  - Best performance
  - Best portability
- Still need to run with mpirun

```cpp
optimizer::GradientDescent opt(...);

optimizer::DistributedGradientDescent opt(..., Strategy);
```

Custom distribution strategies
Model

Tie Everything Together
- Everything needed to run a DNN is implemented
- Wrap DNN utilities in model class for users
- Can also support other ML models

Implementation
- Use rest of backend and frontend to implement
- Should just “work”

Optimization
- Use Core API
The Layer Class
- A helper class
- Help construct multiple compute graph nodes at once
- More familiar construct to data scientist
- Define helper functions like with operations
Layers

Example Constructing Layers
- act2->output() is top of compute graph
- act2->output()->eval() is a working neural network
- Almost there!

Op input = Var({batchSize, inputFeatures});
Layer *fc1 = fullyConnected(input, 512, true);
Layer *act1 = activation(fc1->output(), RELU);
Layer *fc2 = fullyconnected(act1->output(), 32);
Layer *act2 = activation(fc2->output(), RELU);
Model

The Base Model
- Every Model has a “fit” or “train” function
  - Where learning happens
- Every model has a “predict” function
  - To use the model for actual classification tasks

class Model {
    virtual void summary() = 0;
    virtual void fit(Tensor &x, Tensor &y,...)=0;
    virtual Tensor predict(Tensor &x) = 0;
};
class NeuralNetwork : public Model {

    virtual void train(X, Y, epochs, optimizer) {
        for i ... epochs {
            for n ... batchesPerEpoch {
                input_.copy(batches.get(n));

                optimizer.minimize(lossFunc->output(),{weights});

            }
        }
    }
};

Model

Training Routine (Abbreviated)
- Put loss function on last layer
- For every batch
  - Copy data into input
  - Run optimizer
- Network is training!
- Can also record loss, accuracy, and training time here
Model

Example Use Case
- Training routines are now easy
- Remove error prone code

NeuralNetwork nn(layers);

nn.fit(xData, yData, nEpochs, SGD);
Software Structure of a DNN Framework

- **Core**
  - **Systems**
    - Memory (CPU, GPU, Hybrid)
    - Communication (MPI, OpenMP)
  - **Math**
    - Tensor data structures and operations
    - Blas/Lapack, Magma
    - DNN accelerator (CuDNN, hipDNN, OneDNN)
- **Frontend**
  - Compute Graph
  - Optimizers
  - Models
MagmaDNN 1.2

Rocco Febbo

Introduction | Examples | Core DNN Concepts

Development Version
https://github.com/MagmaDNN/magmadnn

Release Version
https://bitbucket.org/icl/magmadnn/src/master/
What’s New in MagmaDNN 1.2

- **oneDNN(MKLDNN) Support**
  - fully connected, convolutional, and pooling layers
- **CMake Build System**
- **New Examples**
  - CNN
  - ResNet
  - AlexNet
  - LeNet
  - MNIST and CIFAR interactive
  - VGG16
  - Tensor Math
- **C++ Style Formatter**
- **Modularized Distributed Optimizer**
- **CIFAR10, CIFAR100, MNIST dataloaders**
- **Cuda Stream**
- **Model Summary**
- **Spack Package Manager Support**
MagmaDNN primarily interfaces with memory in the form of a Tensor. The shape of the tensor is defined when it is created along with how it should be filled and what device it should be stored on.

```cpp
Tensor<float> *A_tensor = new Tensor<float> ({M, N}, {CONSTANT, {3}}, HOST);
Tensor<float> *x_tensor = new Tensor<float> ({N, 1}, {CONSTANT, {2}}, HOST);
Tensor<float> *b_tensor = new Tensor<float> ({M, 1}, {ONE, {}}, HOST);
```

The Tensor interface has many built-in functions. A few are shown below.

```cpp
A_tensor->get_shape(); //gets the shape in the form of a vector of unsigned ints
A_tensor->get({0,0}); //gets the first element
A_tensor->get(0); //gets the first element using flattened index
A_tensor->get_size(); //gets the number of elements in tensor
A_tensor->set({0,0}); //sets the first element
```

Tensors can be initialized using one of the nine predefined formats as shown below.

```cpp
/** Different ways to initialize the tensor on creation. */
enum tensor_fill_t { UNIFORM, GLOROT, MASK, CONSTANT, ZERO, ONE, DIAGONAL, IDENTITY, NONE };
```
MagmaDNN performs computations by creating a compute graph then evaluating it. This has the benefit of allowing the compute graph to be optimized before the calculation is evaluated. Below is a tensor calculation of $Ax + b$.

First the variables are created with initialized values. Here $A$ is an $M$ by $N$ matrix filled with the value 3. ‘$x$’ is a vector of length $N$ filled with 2. ‘$b$’ is a vector of length $M$ filled with the value 1. Each of these variables are stored on system RAM due to the ‘HOST’ specification. This means that any calculations will be performed by the CPU.

Here an operation using those three variables is defined.

```cpp
auto A = op::var<float>("A", {M, N}, {CONSTANT, {3}}, HOST);
auto x = op::var<float>("x", {N, 1}, {CONSTANT, {2}}, HOST);
auto b = op::var<float>("b", {M, 1}, {ONE, {}}, HOST);

auto aff = op::add(op::matmul(A, x), b);
```

Now, the operation is evaluated on the CPU.

```cpp
/* get the final tensor result by evaluating the compute tree */
Tensor<float> *final_val = aff->eval();
```
When building your own neural network, creating a data loader interface can be useful. In MagmaDNN, there are built-in interfaces for loading the MNIST datasets and CIFAR datasets.

```cpp
// Location of the MNIST dataset
std::string const mnist_dir = ".";
// Load MNIST training dataset
magmadnn::data::MNIST<T> train_set(mnist_dir, magmadnn::data::Train);
magmadnn::data::MNIST<T> test_set(mnist_dir, magmadnn::data::Test);
```

Data can also be loaded manually using the tensor interfaces. When a tensor is initialized, it’s shape and memory type are specified.

```cpp
/* allocate tensor */
data = new Tensor<float>({n_images, n_rows, n_cols}, {NONE, {}}, HOST);

for (uint32_t i = 0; i < n_images; i++) {
    FREAD_CHECK(fread(bytes, sizeof(char), n_rows * n_cols, file), n_rows * n_cols);

    for (uint32_t r = 0; r < n_rows; r++) {
        for (uint32_t c = 0; c < n_cols; c++) {
            val = bytes[r * n_cols + c];
            data->set(i * n_rows * n_cols + r * n_cols + c, (val / 128.0f) - 1.0f);
        }
    }
}
```
The parameters for batch size, epochs, and learning rate are defined in a struct and passed to the model when it is created.

```c
model::nn_params_t params;
params.batch_size = 128; /* batch size: the number of samples to process in each mini-batch */
params.n_epochs = 10;   /* # of epochs: the number of passes over the entire training set */
params.learning_rate = 0.05;
```

An operation is defined for the input to the model with the needed shape. The memory type selects which device our network will train on.

```c
auto x_batch = op::var<float>("x_batch", // Name
{params.batch_size, train_set.nchannels(), train_set.nrows(), train_set.ncols()}, // Shape
{NONE, {}}, // Initialization
training_memory_type); // HOST for CPU, DEVICE for GPU
```

This will be passed into the first layer of our network.

```c
auto input = layer::input(x_batch);
```
DNN Programming Structure

Exercises

Activation and Fit Functions

As of version 1.2 MagmaDNN has four activation functions, four optimizers, and two loss functions.

```
enum activation_t { SIGMOID, TANH, RELU, SOFTMAX };
enum optimizer_t {
  SGD,
  ADAGRAD,
  RMSPROP,
  ADAM,
};
```

**Adam**

\[
w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\]

**RMSProp**

\[
w_t = w_{t-1} - \eta \frac{\delta C}{\sqrt{E[g]^2}}
\]

**Sigmoid**

\[\sigma(x) = \frac{1}{1 + e^{-x}}\]

**tanh**

\[tanh(x)\]

**ReLU**

\[\text{max}(0, x)\]

image: https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044
Now, the structure of our network can be defined!

For this network, we will use 2 hidden layers to create a multi-layer perceptron network. First we flatten the input since it’s likely multidimensional. When each layer is defined, the input to that layer is provided. This is usually the output of the previous layer. Then, the number of hidden units is declared and if a bias for that layer is needed. The layers are then put in a vector to pass to the model.

```cpp
auto flatten = layer::flatten(input->out());
auto fc1 = layer::fullyconnected(flatten->out(), 784, false);
auto act1 = layer::activation(fc1->out(), layer::RELU);
auto fc2 = layer::fullyconnected(act1->out(), 500, false);
auto act2 = layer::activation(fc2->out(), layer::RELU);
auto fc3 = layer::fullyconnected(act2->out(), train_set.nclasses(), false);
auto act3 = layer::activation(fc3->out(), layer::SOFTMAX);
auto output = layer::output(act3->out());
// Wrap each layer in a vector of layers to pass to the model
std::vector<layer::Layer<float> *> layers = {input, flatten, fc1, act1, fc2, act2, fc3, act3, output};
```
The layers, loss function, optimizer, and model parameters are specified when a model is initialized. The network begins training when the “fit” member function is called using the loaded data.

```cpp
// - layers: the previously created vector of layers containing our
// network
// - loss_func: use cross entropy as our loss function
// - optimizer: use stochastic gradient descent to optimize our
// network
// - params: the parameter struct created earlier with our network
// settings
model::NeuralNetwork<float> model(layers, optimizer::CROSS_ENTROPY, optimizer::SGD, params);

// metric_t records the model metrics such as accuracy, loss, and
// training time
model::metric_t metrics;

/* fit will train our network using the given settings.
   X: independent data
   y: ground truth
   metrics: metric struct to store run time metrics
   verbose: whether to print training info during training or not */
model.fit(&train_set.images(), &train_set.labels(), metrics, true);
```
Creating a convolutional neural network is as simple as adding convolutional layers to a network. In the following code we will create a network with 2 convolutional layers each followed by a max pooling layer. We will then flatten the output of the pooling layer and pass it to 2 MLP layers.

```cpp
auto conv2d1 = layer::conv2d(input->out(), {5, 5}, 32, {0, 0});
auto act1 = layer::activation(conv2d1->out(), layer::RELU);
auto pool1 = layer::pooling(act1->out(), {2, 2}, {0, 0}, {2, 2});

auto conv2d2 = layer::conv2d(pool1->out(), {5, 5}, 32, {0, 0});
auto act2 = layer::activation(conv2d2->out(), layer::RELU);
auto pool2 = layer::pooling(act2->out(), {2, 2}, {0, 0}, {2, 2});

auto flatten = layer::flatten(pool2->out());

auto fc1 = layer::fullyconnected(flatten->out(), 768, true);
auto act3 = layer::activation(fc1->out(), layer::RELU);

auto fc2 = layer::fullyconnected(act3->out(), 500, true);
auto act4 = layer::activation(fc2->out(), layer::RELU);

auto fc3 = layer::fullyconnected(act4->out(), n_classes, false);
auto act5 = layer::activation(fc3->out(), layer::SOFTMAX);

auto output = layer::output(act5->out());
```
Using the network from the previous example, MagmaDNN can output a model summary similar to Tensorflow. When the summary member function is called, each layer is shown in order with the output shape of that layer shown in the middle column and the number of parameters within that layer shown on the right column.

```cpp
std::vector<layer::Layer<float> *> layers =
  {input,
   conv2d1, act1, pool1,
   conv2d2, act2, pool2,
   flatten,
   fc1, act3,
   fc2, act4,
   fc3, act5,
   output};

model::NeuralNetwork<float> model(layers,
  optimizer::CROSS_ENTROPY,
  optimizer::SGD, params);

model::metric_t metrics;

model.summary();
model.fit(images_host, labels_host, metrics, true);
```
Included as one of the examples in MagmaDNN is the MNIST interactive example. This example gives you an untrained neural network with the options to:

- Show Sample
  - show a user selected image in the terminal
- Train
  - trains the network
- Predict
  - predict a specific sample with confidence matrix
- Test
  - Show image indexes with bad predictions

![MNIST Interactive Example](image-url)
Included as one of the examples in MagmaDNN is the CIFAR10 interactive examples. These examples give you an untrained neural network with the options to:

- **Show Sample**
  - show a user selected image in the terminal
- **Train**
  - trains the network
- **Predict**
  - predict a specific sample with confidence matrix
When building MagmaDNN there are several build options at your disposal. Some of which determine the use of external APIs such as CuDNN, Magma, openBLAS, oneDNN, etc. The Cuda, CuDNN, and CuBLAS libraries require an Nvidia GPU to be installed on the system. Likewise, in order to use oneDNN (MKLDNN) an Intel CPU is required to be installed on the system.

MagmaDNN Built-in Functionality

If you are building MagmaDNN on a system incompatible with these libraries, you can use MagmaDNN's built-in tools to perform complex operations on your CPU. The default build behavior will automatically build MagmaDNN this way. Also for compatibility, MagmaDNN includes support for building using an included makefile and CMake.

### Makefile

1. Make a copy of the make.inc example
2. Adjust parameters accordingly
3. Type ‘make install’
4. Type ‘make examples’ to build examples

### CMake

1. Adjust CMakeLists.txt parameters accordingly
2. Create a directory to contain build files (ex. ‘mkdir ..//build.magmadnn’)
3. cd into that directory
4. Type ‘cmake ..//magmadnn’
5. Type ‘make install’
### DNN Programming Structure

#### Exercises

**MagmaDNN Hardware Support**

<table>
<thead>
<tr>
<th>Operation</th>
<th>CPU(native)</th>
<th>CPU(MKLDNN)</th>
<th>GPU(Cuda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adagrad</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>adam</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>rmsprop</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>SGD</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>add</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>argmax</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>UNSUPPORTED</td>
</tr>
<tr>
<td>batch norm</td>
<td>UNSUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>Bias Add</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>tensor concat</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>UNSUPPORTED</td>
</tr>
<tr>
<td>conv2d</td>
<td>SUPPORTED</td>
<td>SUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>cross entropy</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>div</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>dot</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>dropout</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>flatten</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>linear</td>
<td>SUPPORTED</td>
<td>SUPPORTED</td>
<td>SUPPORTED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>CPU(native)</th>
<th>CPU(MKLDNN)</th>
<th>GPU(Cuda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>log</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>matmul</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>MSE</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>pooling</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>pow</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>product</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>relu</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>sigmoid</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>softmax</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>sum</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>tanh</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>transpose</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>reduce sum</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>scalar product</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
<tr>
<td>tensor fill</td>
<td>SUPPORTED</td>
<td>UNSUPPORTED</td>
<td>SUPPORTED</td>
</tr>
</tbody>
</table>
When building MagmaDNN with CMake the build options are specified in CMakeLists.txt.

```bash
# MagmaDNN options
option(MAGMADNN_ENABLE_CUDA "Enable use of CUDA library and compilation of CUDA kernel" OFF)
option(MAGMADNN_ENABLE_MPI "Enable distributed memory routines using MPI" OFF)
option(MAGMADNN_ENABLE_OMP "Enable parallelization using OpenMP library" OFF)
option(MAGMADNN_ENABLE_MKLDNN "Enable use of MKLDNN library" OFF)
option(MAGMADNN_BUILD_MKLDNN "Enable build of MKLDNN from source" OFF)
option(MAGMADNN_BUILD_DOC "Generate documentation" OFF)
option(MAGMADNN_BUILD_EXAMPLES "Build MagmaDNN examples" ON)
option(MAGMADNN_BUILD_TESTS "Generate build files for unit tests" OFF)
option(MAGMADNN_BUILD_SHARED_LIBS "Build shared (.so, .dylib, .dll) libraries" ON)
```

As with CMake there is no need to specify library locations.
Building MagmaDNN: Makefile Options

The install location is specified as the prefix parameter

```bash
# install location
prefix = /usr/local
```

When using the makefile build system MagmaDNN will defaults attempt to build using CUDA unless explicitly specified.

```bash
# set to 0 if you don't want to compile with CUDA
TRY_CUDA = 1
```

MagmaDNN defaults to not compiling with MKLDNN.

```bash
# set to 1 if you want to compile with MKLDNN
USE_MKLDNN = 0
```

To compile with CUDA, Magma, and a BLAS lib the library locations should be specified here.

```bash
# LINKS to locations of CUDA and MAGMA
CUDADIR ?= /usr/local/cuda
MAGMADIR ?= /usr/local/magma

# LINKS to your CPU C BLAS library (i.e. atlas, mkl, openblas)
BLASDIR ?= /usr/local/openblas
BLASLIB ?= openblas
```
MagmaDNN has a set of tests you can run after compiling to ensure correctness of computations and that you are ready to build a neural network.

```verbatim
# Testing managed copy
Testing managed get/set... Success!
Testing CUDA managed get/set... Success!
Testing host->host copy... Success!
Testing host->managed copy... Success!
Testing host->managed copy... Success!
Testing device->host copy... Success!
Testing device->device copy... Success!
Testing device->managed copy... Success!
Testing device->managed copy... Success!
Testing managed->host copy... Success!
Testing managed->device copy... Success!
Testing managed->managed copy... Success!
Testing cuda->managed copy... Success!
Testing cuda->managed copy... Success!
Testing cuda->device copy... Success!
Testing cuda->managed copy... Success!

== TESTNG testing tensor ==
Testing indexing on device host... Success!
Testing indexing on device device... Success!
Testing indexing on device managed... Success!
Testing indexing on device cuda managed... Success!
Testing fill constant on host... Success!
Testing fill constant on managed... Success!
Testing fill constant on cuda managed... Success!
Testing copying on device host... Success!
Testing copying on device device... Success!
Testing copying on device managed... Success!
Testing copying on device cuda managed... Success!

== TESTNG testing math ==
Testing host matmul... Success!
Testing device matmul... Success!
Testing managed matmul... Success!
Testing cuda managed matmul... Success!
Testing host pow... Success!
Testing device pow... Success!
Testing cuda managed pow... Success!

Testing full grad on device... Success!
Testing full grad on managed... Success!
Testing full grad on cuda managed... Success!
Testing optimization on host device copy... Success!
Testing optimization on device... Success!
Testing optimization on managed... Success!
Testing optimization on cuda managed... Success!

== TESTING testing layers ==
Testing host input... Success!
Testing device input... Success!
Testing managed input... Success!
Testing host fully connected... Success!
Testing device fully connected... Success!
Testing managed fully connected... Success!
Testing cuda managed fully connected... Success!
Testing host activation... Success!
Testing device activation... Success!
Testing managed activation... Success!
Testing cuda managed activation... Success!
Testing host dropout... Success!
Testing device dropout... Success!
Testing managed dropout... Success!
Testing host flattened... Success!
Testing device flattened... Success!
Testing managed flattened... Success!
Testing cuda managed flattened... Success!
Testing host layers... Success!
Testing device layers... Success!
Testing managed layers... Success!
Testing cuda managed layers... Success!
Testing host layer container... Success!
Testing device layer container... Success!
Testing managed layer container... Success!
Testing cuda managed layer container... Success!

== TESTING testing model ==
Testing host MLP... Success!
Testing device MLP... Success!
Testing managed MLP... Success!
```


**Compiling and Training the Network**

1. Install MagmaDNN
2. Include MagmaDNN headers in your cpp file
3. Link MagmaDNN and any external libraries

```g++
conv.cpp -I/opt/cuda/include/ -I/usr/local/magma/include/ -L/opt/cuda/lib/ -L/usr/local/magma/lib -lmagmadnn -lcublas -lcudnn -lmagma -o train

./train
```

Preparing to read 60000 images with size 28 x 28 ...
finished reading images.
Preparing to read 60000 labels with 10 classes ...
finished reading labels.
Epoch (0/20): accuracy=0.93 loss=0.222 time=1
Epoch (1/20): accuracy=0.9534 loss=0.1458 time=2
Epoch (2/20): accuracy=0.9638 loss=0.1126 time=3
Epoch (3/20): accuracy=0.97 loss=0.09276 time=4
...
Epoch (19/20): accuracy=0.9915 loss=0.0241 time=19
Final Training Metrics: accuracy=0.9915 loss=0.0241 time=19
model.fit(&train_set.images(), &train_set.labels(), metrics, true);

// Compute accuracy of the model on the test set
uint32_t total_correct = 0;

Tensor<T> sample({test_set.nchannels(), test_set.nrows(), test_set.ncols()}, {NONE, {}}, test_set.images().get_memory_type());

for (uint32_t i = 0; i < test_set.images().get_shape(0); ++i) {
    sample.copy_from(test_set.images(), i * sample.get_size(), sample.get_size());
    auto predicted_class = model.predict_class(&sample);

    auto actual_class = test_set.nclasses() + 1;
    for (uint32_t j = 0; j < test_set.nclasses(); j++) {
        if (fabs(test_set.labels().get(i * test_set.nclasses() + j) - 1.0f) <= 1E-8) {
            actual_class = j;
            break;
        }
    }

    if (actual_class == predicted_class) {
        total_correct++;
    }
}

double accuracy = static_cast<double>(total_correct) / static_cast<double>(test_set.images().get_shape(0));
std::cout << "Model accuracy on testset: " << accuracy << std::endl;
MagmaDNN has built-in support to perform distributed training using MPI by averaging the neural network weights across each node after each batch for data parallelism.

**MPI_Init is called before magmadnn_init()**

```c
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nnodes);
```

After the network architecture is defined, an optimizer is created using the DistributedGradientDescent class and passed to the model:

```cpp
/* here use nnodes processors */
auto optim = new optimizer::DistributedGradientDescent<float>(params.learning_rate);
model::NeuralNetwork<float> model(layers, optimizer::CROSS_ENTROPY, optim, params);
```

The fit member function is called and training begins across all nodes.

```cpp
model.fit(images_host, labels_host, metrics, true);
```
One of the examples included in MagmaDNN is a LeNet network. The image below shows how the layers are defined in MagmaDNN.

```
auto conv2d1 = layer::conv2d(input->out(), {5, 5}, 32, {0, 0}, {1, 1}, {1, 1});
auto act1 = layer::activation(conv2d1->out(), layer::TANH);
auto pool1 = layer::pooling(act1->out(), {2, 2}, {0, 0}, {2, 2}, AVERAGE_POOL);
auto conv2d2 = layer::conv2d(pool1->out(), {5, 5}, 32, {0, 0}, {1, 1}, {1, 1});
auto act2 = layer::activation(conv2d2->out(), layer::TANH);
auto pool2 = layer::pooling(act2->out(), {2, 2}, {0, 0}, {2, 2}, AVERAGE_POOL);
auto flatten = layer::flatten(pool2->out());
auto fc1 = layer::fullyconnected(flatten->out(), 120, true);
auto act3 = layer::activation(fc1->out(), layer::TANH);
auto fc2 = layer::fullyconnected(act3->out(), 84, true);
auto act4 = layer::activation(fc2->out(), layer::TANH);
auto fc3 = layer::fullyconnected(act4->out(), train_set.nclasses(), false);
auto act5 = layer::activation(fc3->out(), layer::SOFTMAX);
```

Another example network in MagmaDNN is AlexNet. This network trains on the CIFAR100 dataset.

The ResNet example in MagmaDNN uses basic blocks to create the network.

```cpp
std::vector<layer::Layer<T> *> blocks(0, nullptr);
for (int i = 0; i < num_stacked_blocks; ++i) {
    op::Operation<T>* block1_input = nullptr;
    if (i == 0) {
        // First block
        block1_input = act1->out();
    } else {
        // Subsequent block: input from previous stacked block output
        block1_input = blocks.back()->out();
    }
    auto block1 = basic_block(
        block1_input, 16, {1, 1}, enableShortcut);
    auto block2 = basic_block(
        block1.back()->out(), 16, {1, 1}, enableShortcut);
    blocks.insert(std::end(blocks), std::begin(block1), std::end(block1));
    blocks.insert(std::end(blocks), std::begin(block2), std::end(block2));
}
```
The VGG16 example in MagmaDNN layer structure is outlined below.

```c
/* CHUNK 1 */
auto conv1 = layer::conv2d<float>(input_layer->out(), {3, 3}, 64, layer::SAME);
auto act1 = layer::activation<float>(conv1->out(), layer::RELU);
/* batchnorm */
auto dropout1 = layer::dropout<float>(act1->out(), 0.3);
auto conv2 = layer::conv2d<float>(dropout1->out(), {3, 3}, 64, layer::SAME);
auto act2 = layer::activation<float>(conv2->out(), layer::RELU);
/* batchnorm */
auto pool1 = layer::pooling<float>(act2->out(), {2, 2}, layer::SAME, {1, 1}, MAX_POOL);
/* CHUNK 2 */
auto conv3 = layer::conv2d<float>(pool1->out(), {3, 3}, 128, layer::SAME);
auto act3 = layer::activation<float>(conv3->out(), layer::RELU);
/* batchnorm */
auto dropout2 = layer::dropout<float>(act3->out(), 0.4);
auto conv4 = layer::conv2d<float>(dropout2->out(), {3, 3}, 128, layer::SAME);
auto act4 = layer::activation<float>(conv4->out(), layer::RELU);
/* batchnorm */
auto pool2 = layer::pooling<float>(act4->out(), {2, 2}, layer::SAME, {1, 1}, MAX_POOL);
/* CHUNK 3 */
auto conv5 = layer::conv2d<float>(pool2->out(), {3, 3}, 256, layer::SAME);
auto act5 = layer::activation<float>(conv5->out(), layer::RELU);
/* batchnorm */
auto dropout3 = layer::dropout<float>(act5->out(), 0.4);
auto conv6 = layer::conv2d<float>(dropout3->out(), {3, 3}, 256, layer::SAME);
auto act6 = layer::activation<float>(conv6->out(), layer::RELU);
/* batchnorm */
auto dropout4 = layer::dropout<float>(act6->out(), 0.4);
auto conv7 = layer::conv2d<float>(dropout4->out(), {3, 3}, 256, layer::SAME);
auto act7 = layer::activation<float>(conv7->out(), layer::RELU);
/* batchnorm */
auto pool3 = layer::pooling<float>(act7->out(), {2, 2}, layer::SAME, {1, 1}, MAX_POOL);
/* CHUNK 4 */
auto conv8 = layer::conv2d<float>(pool3->out(), {3, 3}, 512, layer::SAME);
auto act8 = layer::activation<float>(conv8->out(), layer::RELU);
/* batchnorm */
auto dropout5 = layer::dropout<float>(act8->out(), 0.4);
auto conv9 = layer::conv2d<float>(dropout5->out(), {3, 3}, 512, layer::SAME);
auto act9 = layer::activation<float>(conv9->out(), layer::RELU);
/* batchnorm */
auto dropout6 = layer::dropout<float>(act9->out(), 0.4);
auto conv10 = layer::conv2d<float>(dropout6->out(), {3, 3}, 512, layer::SAME);
auto act10 = layer::activation<float>(conv10->out(), layer::RELU);
/* batchnorm */
auto pool4 = layer::pooling<float>(act10->out(), {2, 2}, layer::SAME, {1, 1}, MAX_POOL);
/* CHUNK 5 */
auto conv11 = layer::conv2d<float>(pool4->out(), {3, 3}, 512, layer::SAME);
auto act11 = layer::activation<float>(conv11->out(), layer::RELU);
/* batchnorm */
auto dropout7 = layer::dropout<float>(act11->out(), 0.4);
auto conv12 = layer::conv2d<float>(dropout7->out(), {3, 3}, 512, layer::SAME);
auto act12 = layer::activation<float>(conv12->out(), layer::RELU);
/* batchnorm */
auto dropout8 = layer::dropout<float>(act12->out(), 0.4);
auto conv13 = layer::conv2d<float>(dropout8->out(), {3, 3}, 512, layer::SAME);
auto act13 = layer::activation<float>(conv13->out(), layer::RELU);
/* batchnorm */
auto pool5 = layer::pooling<float>(act13->out(), {2, 2}, layer::SAME, {1, 1}, MAX_POOL);
auto dropout9 = layer::dropout<float>(pool5->out(), 0.5);
auto flat = layer::flatten<float>(dropout9->out());
auto fc1 = layer::fully_connected<float>(flat->out(), 512, False);
auto act14 = layer::activation<float>(fc1->out(), layer::RELU);
/* batchnorm */
```
MagmaDNN 1.2

Introduction | Examples | Core DNN Concepts

Development Version
https://github.com/MagmaDNN/magmadnn

Release Version
https://bitbucket.org/icl/magmadnn/src/master/