LECTURE UNITS
MAT391
Topics in Machine Learning

Kwai Wong
University of Tennessee, Knoxville

September 23, 2022

Acknowledgements:

- Support from NSF, UTK, JICS, ICL, NICS
- LAPENNA, www.jics.utk.edu/lapenna, NSF award #202409
- www.icl.utk.edu, cfdlab.utk.edu, www.xsede.org, www.jics.utk.edu/recsem-reu,
- MagmaDNN is a project grown out from the RECSEM REU Summer program supported under NSF award #1659502
- Source code: www.bitbucket.org/icl/magmadnn
- www.bitbucket.org/cfdl/opendnnwheel

INNOVATIVE COMPUTING LABORATORY

NSF

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Unit 3: DNN Computing Ecosystem

- DNN Training Cycle
- Linear Algebra Revisit, Performance
- GPU Brief Overview

Supervised Learning

Supervised Learning Data:
(x, y) x is data, y is label

Goal:
Learn a function to map x -> y

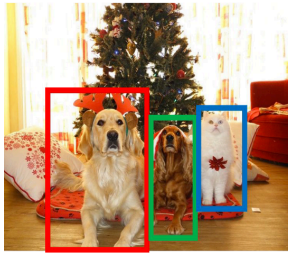
Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

CAT

Classification

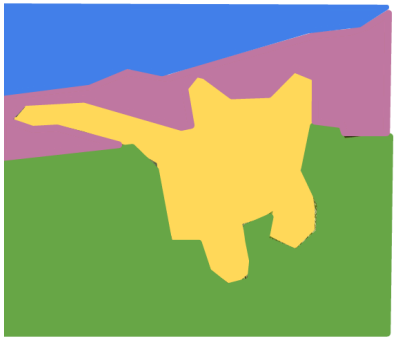
A cat sitting on a suitcase on the floor

Image captioning



DOG, DOG, CAT

Object Detection



GRASS, CAT, TREE, SKY

Semantic Segmentation

Basic Ideas

Typical Neural Network – MLP

STEP 1 : Model Definition

STEP 2 : Cost Function (MSE, CE)

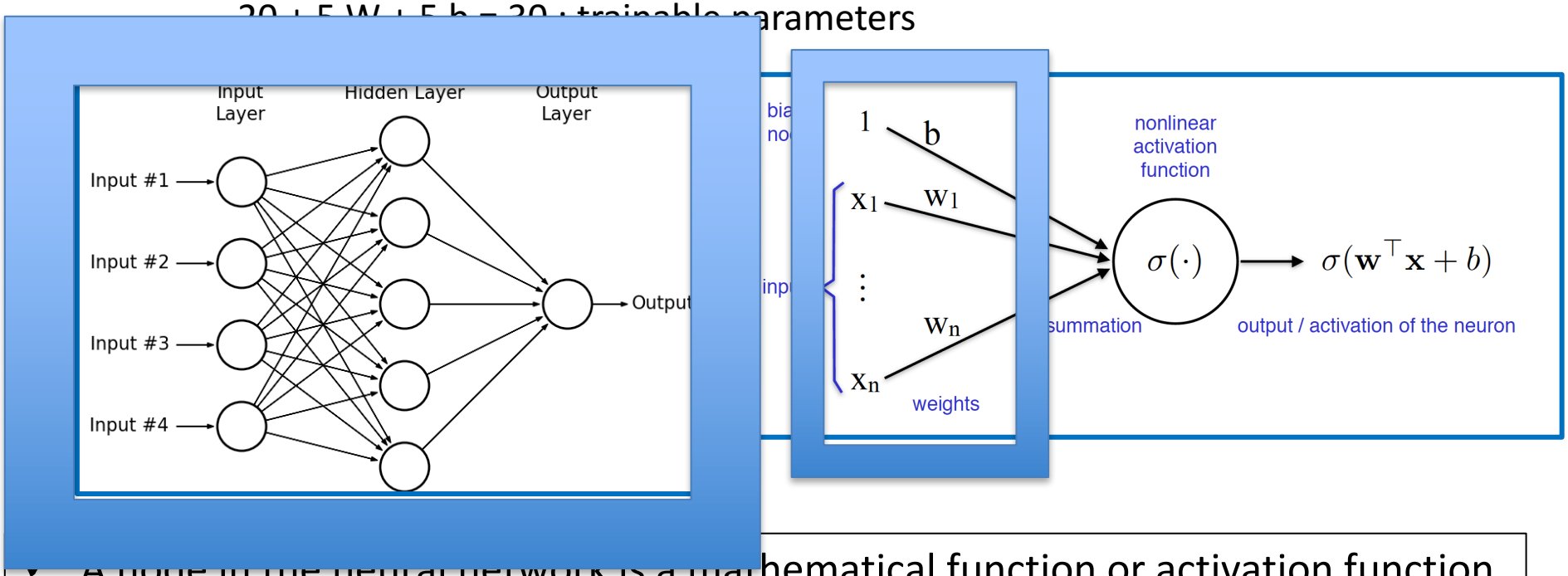
Step 3 : Optimization Scheme (GRADIENT)

Step 4 : Numerical Implementation (fitting)

Step 5 : Evaluation

NN Modeling

20 + 5 W + 5 b = 30 : trainable parameters



- ✓ A node in the neural network is a mathematical function or activation function which maps input to output values.
- ✓ Inputs represent a set of vectors containing weights (w) and bias (b). They are the sets of parameters to be determined.
- ✓ 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.

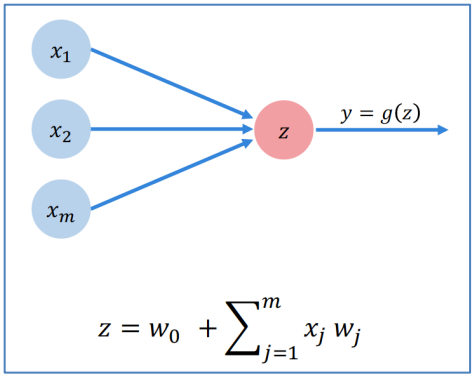
Training a Neural Network

"Training a neural network"
What does it mean?
What does the network train?
What are the training parameters
How does it work?

- ✓ Neural network is composed of many unknown parameters (weights and bias), initialize them randomly in the beginning
- ✓ Perform forward path computations
- ✓ Computed results are compared to ground truth (labels) and errors are calculated → Need to have a cost (loss) function for error evaluation
- ✓ Use the error to adjust the weights and bias (backpropagation)
- ✓ Repeated the process (training) until the model is good enough

Forward and backward computation operators

Z = (x1*w1 + x2*w2) + b = (-1.0*-2.0 + -2.0*3.0) + -3.0 = 4.0 – 3.0 = 1.0



Sigmoid function

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

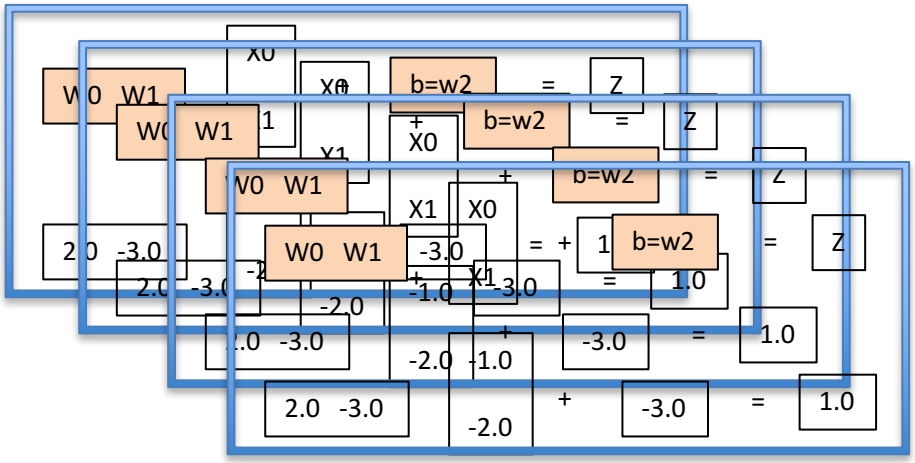
Y = σ (z) = 1 / (1 + e $^{-Z}$)
= 1 / (1 + e $^{-1}$) = 0.731

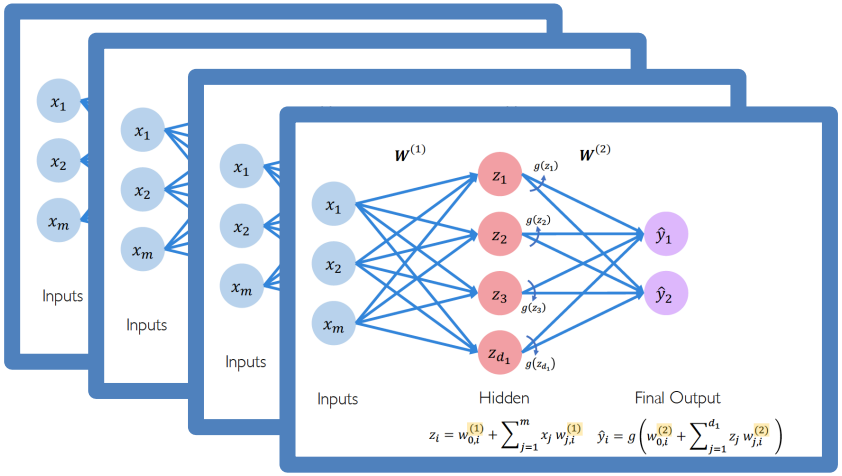
multiple operator :
downstream gradient =
upstream gradient * input value

add operator :
Downstream gradient =
upstream gradient

function operator :
downstream gradient =
Upstream gradient * local function gradient

Input (X, W) → Activation F(X) → Output (Y)





many data points; many neurons → matrix matrix multiplication (GEMM)

Trainable parameters
W (i) = Weights, b = bias
1 weight per connection (link)
1 bias per neuron
Total number of parameters = 3

Weight and bias
are initialized randomly

Goal : Adjust W and b
Training a neural network

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

new W = old W– learning rate * (grad [Y w.r.t W])

(X+, W+) Input (X, W) (grad) ← (grad) Activation F(X) (grad) ← Output (Y) ; grad = 1

Basic Ideas
Typical Neural Network – MLP

STEP 1 : Model Definition

(Simple Network)

STEP 2 : Cost Function

(Y)

Step 3 : Optimization Scheme

new W = old W– learning rate * (grad [Y w.r.t W])

Step 4 : Numerical Implementation

(forward and backward Calculation)

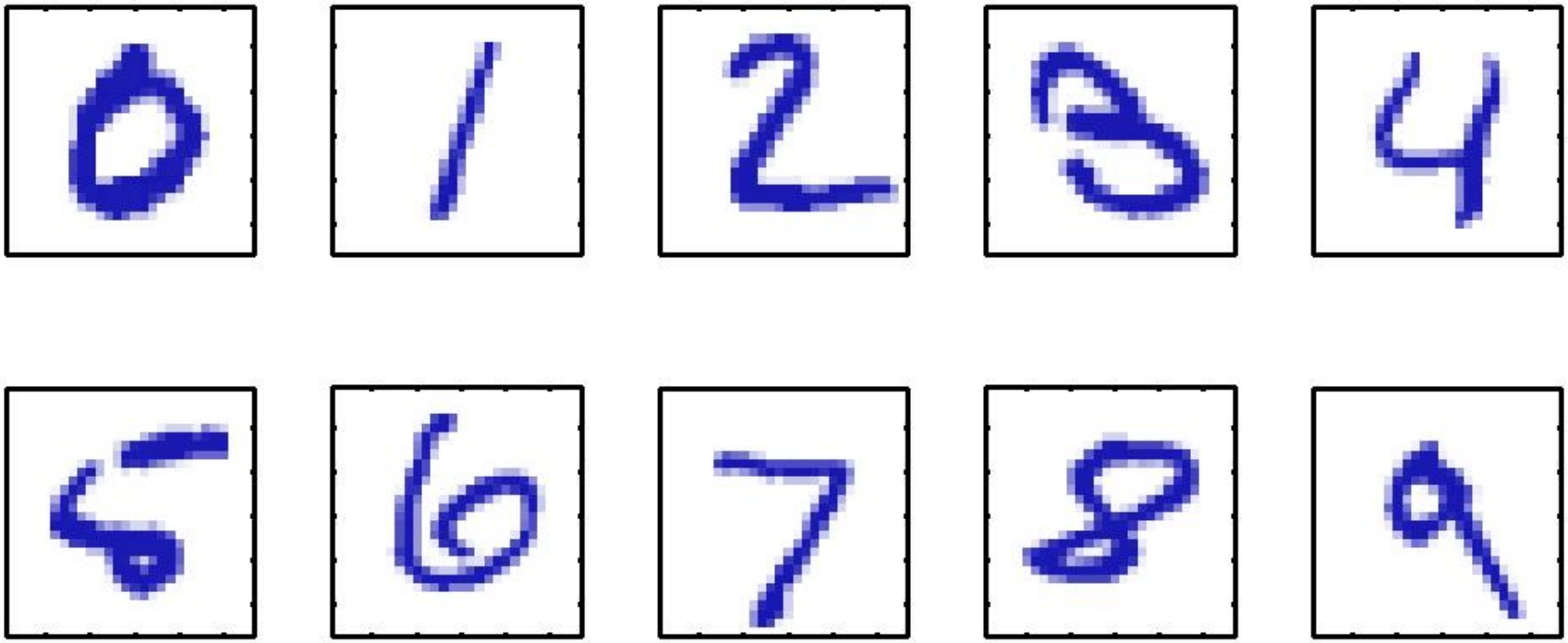
Step 5 : Evaluation

Error (Label – Computed)

Example: how can computer see images?

Handwritten Digit Recognition (MNIST data set)

The **MNIST database** (*Modified National Institute of Standards and Technology database*) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning.[4][5] It was created by "re-mixing" the samples from NIST's original datasets. The creators felt that since NIST's training dataset was taken from American Census Bureau employees, while the testing dataset was taken from American high school students, it was not well-suited for machine learning experiments. Furthermore, the black and white images from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels. The MNIST database contains 60,000 training images and 10,000 testing images. – from Wikipedia



Basic Ideas

Typical Neural Network – MLP

STEP 1 : Model Definition (Network)

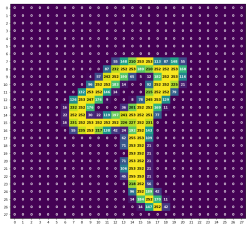
STEP 2 : Cost Function (MSE, CE)

Step 3 : Optimization Scheme (GRADIENT)

Step 4 : Numerical Implementation (fitting)

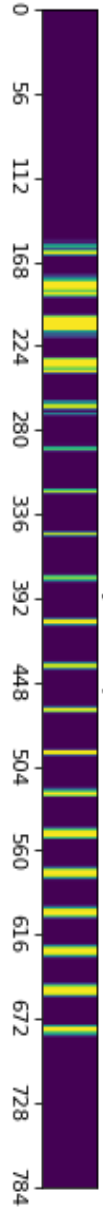
Step 5 : Evaluation

Flatten the 28 x 28 matrix to a vector of 28 x 28 of 784 elements vector = Input



Simple MNIST MLP Network

Google Colab Code



Flatten Layer Output

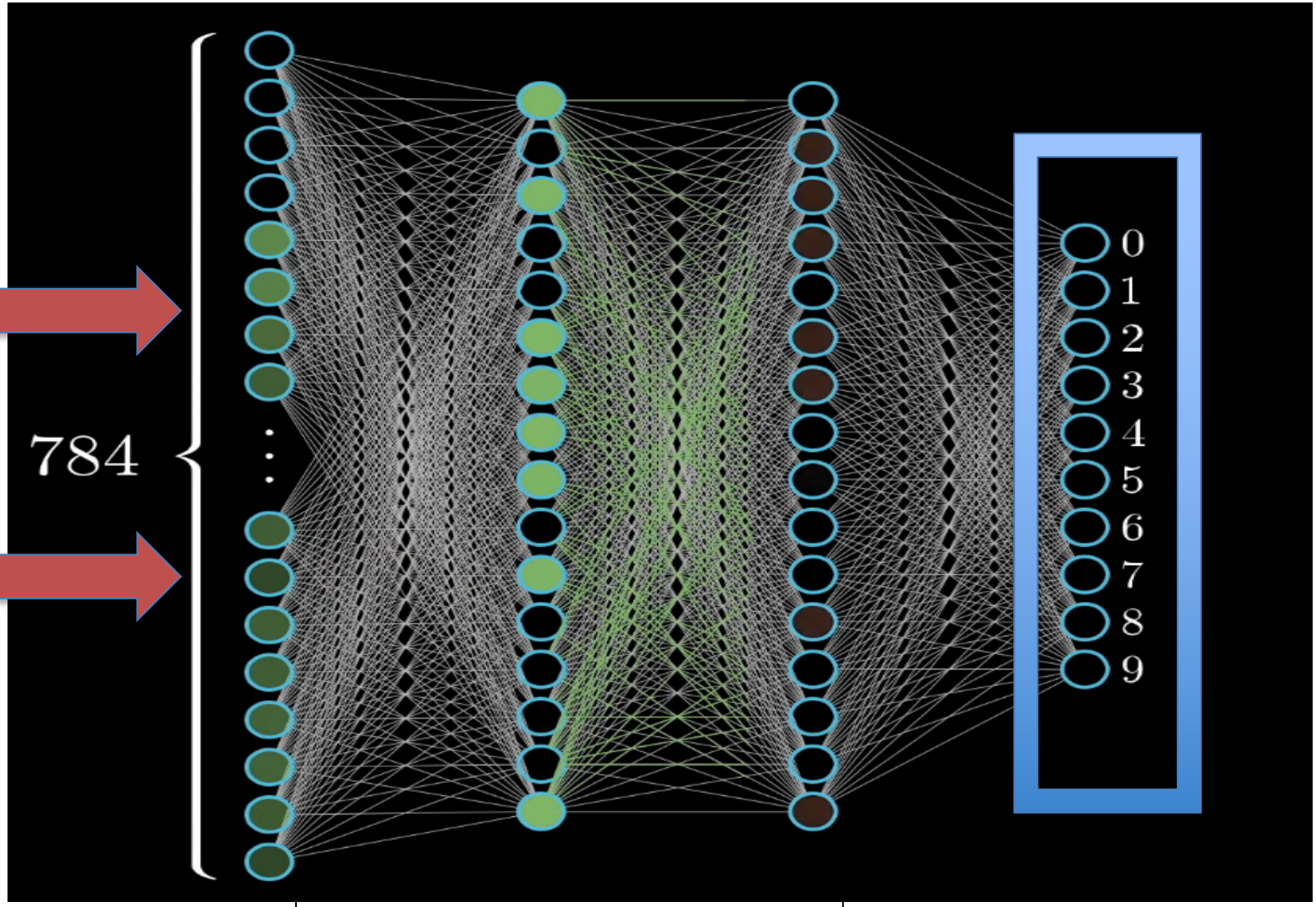


Image:https://www.3blue1brown.com/

Basic Ideas

Typical Neural Network – MLP

STEP 1 : Model Definition (Network)

STEP 2 : Cost Function (MSE, CE)

Step 3 : Optimization Scheme (GRADIENT)

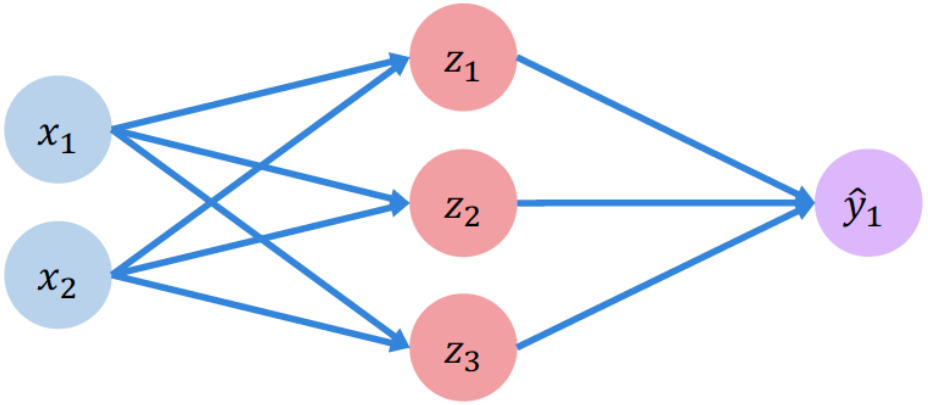
Step 4 : Numerical Implementation (fitting)

Step 5 : Evaluation

Mean Squared Error Loss

Mean squared error loss *can be used with regression models that output continuous real numbers*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$	y
30	90
80	20
85	95
⋮	⋮

Final Grades (percentage)

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)}; \boldsymbol{W}\right)\right)^2$$

Actual Predicted

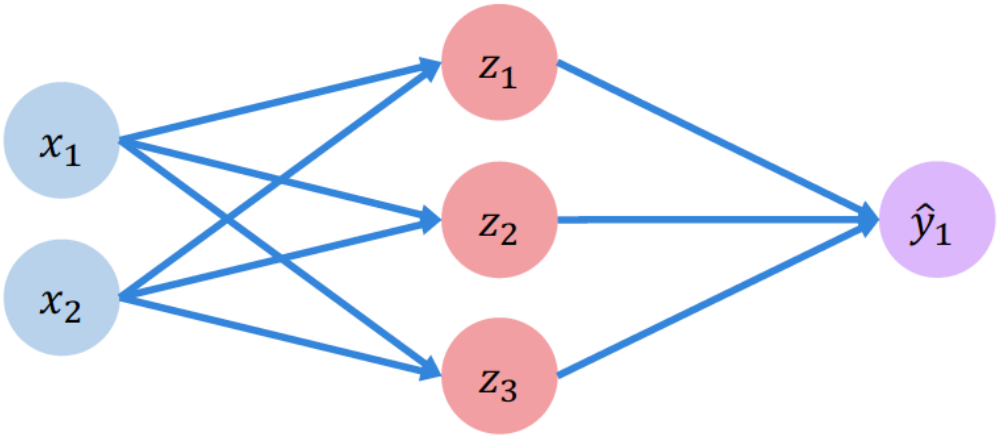
```
loss = tf.reduce_mean( tf.square(tf.subtract(model.y, model.pred) )
```

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Cross Entropy Loss

Cross entropy loss *can be used with models that output a probability between 0 and 1*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$$f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \qquad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )
```

Basic Ideas

Typical Neural Network – MLP

STEP 1 : Model Definition (Network)

STEP 2 : Cost Function (MSE, CE)

Step 3 : Optimization Scheme (GRADIENT)

Step 4 : Numerical Implementation (fitting)

Step 5 : Evaluation

Derivatives and Gradient descent

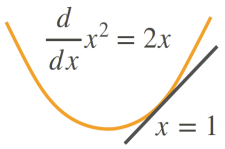
Review Scalar Derivative

y	a	x^n	$\exp(x)$	$\log(x)$	$\sin(x)$
$\frac{dy}{dx}$	0	nx^{n-1}	$\exp(x)$	$\frac{1}{x}$	$\cos(x)$

a is not a function of x

y	$u+v$	uv	$y=f(u),\ u=g(x)$
$\frac{dy}{dx}$	$\frac{du}{dx}+\frac{dv}{dx}$	$\frac{du}{dx}v+\frac{dv}{dx}u$	$\frac{dy}{du}\frac{du}{dx}$

Derivative is the slope of the tangent line



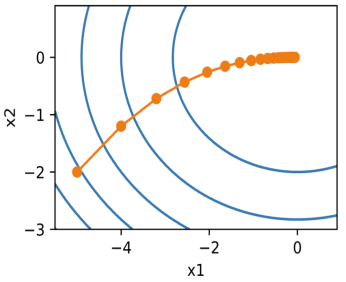
The slope of the tangent line is 2

Algorithm

- Choose initial $\mathbf{x}_0$
- At time $t = 1, \ldots, T$

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta \nabla f(\mathbf{x}_{t-1})$$

- η is called learning rate



Generalize to Vectors

- Chain rule for scalars:

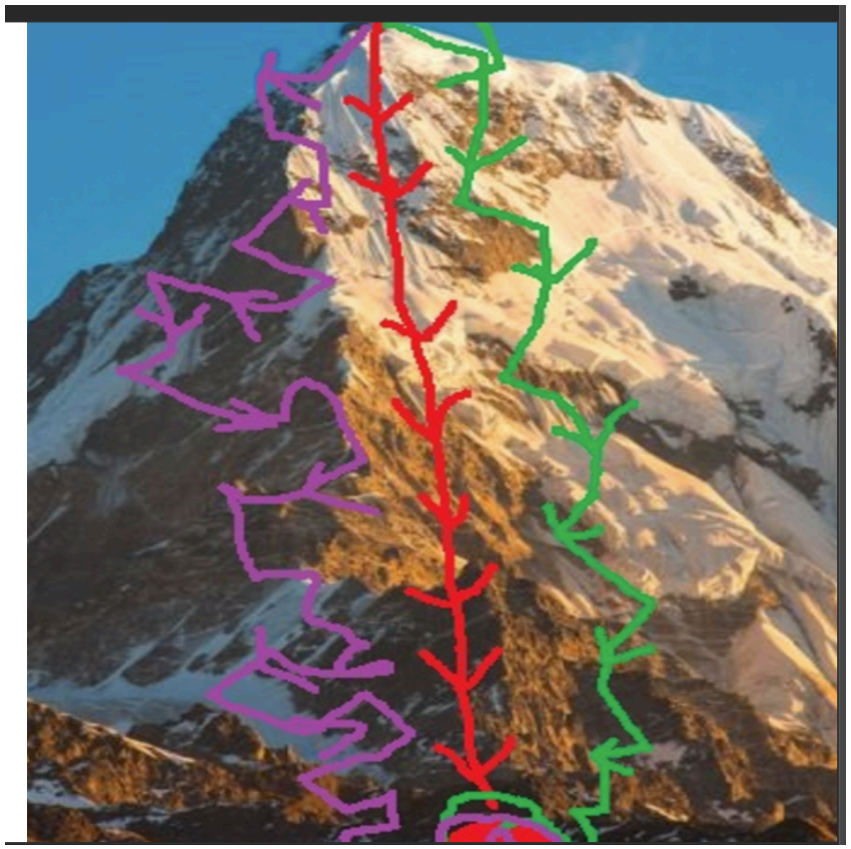
$$y = f(u),\ u = g(x) \qquad \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

- Generalize to vectors straightforwardly

$$\frac{\partial y}{\partial \mathbf{x}} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial \mathbf{x}} \qquad \frac{\partial y}{\partial \mathbf{x}} = \frac{\partial y}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{x}} \qquad \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$$

$$(1,n)\quad (1,)\ (1,n) \qquad (1,n)\quad (1,k)\ (k,n) \qquad (m,n)\quad (m,k)\ (k,n)$$

Optimization : Derivatives and Gradient Descent



Batch Gradient Descent
Mini-batch Gradient Descent
Stochastic Gradient Descent

Simple story

Derivative is the slope of the tangent line

$$\frac{d}{dx}x^2 = 2x$$

$x = 1$

The slope of the tangent line is 2

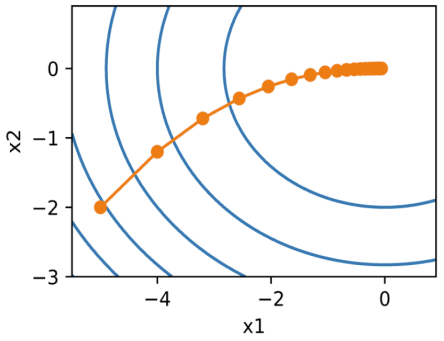
$x)$

Algorithm

- Choose initial $\mathbf{x}_0$
- At time $t = 1, \ldots, T$

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta \nabla f(\mathbf{x}_{t-1})$$

- η is called learning rate



Training

"Training a neural network"

What does it mean?

What does the network train?

minimize error of the computed values against the labeled values (loss function)

What are the training parameters

W and b

How does it work?

optimization based on gradient descent algorithm

go back and forth in the NN model to adjust for w and b => need derivatives w.r.t. w and b

Go through forward calculation

training with backpropagation

Error and Scheme

neuralnetworksanddeeplearning.com

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

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

Find a scheme to minimize the cost *C*(*w*,*b*) as a function of the weights and biases, casting it as an optimization problem using the ***gradient descent algorithm***.

Find a way of iterating Δ*w*j and Δ*b* so as to make Δ*C* (Δ*output*) negative, Pushing MSE (C) smaller and smaller, ideally to zero

Be negative
Drive the cost to zero

=

$$\Delta \text{output} \approx \sum_j \frac{\partial \,\text{output}}{\partial w_j} \Delta w_j + \frac{\partial \,\text{output}}{\partial b} \Delta b,$$

For output = C ; wj = v1 ; b = v2

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2.$$

$$\Delta C \approx \nabla C \cdot \Delta v.$$

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

Be negative

$$\Delta C \approx \nabla C \cdot \Delta v.$$

Just Choose

$$\Delta v = -\eta \nabla C,$$

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

$$v \to v' = v - \eta \nabla C.$$

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

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Mini-Batch Stochastic Gradient Descent

- ✓ Neural network is an optimization process to find a set of parameters (W, weights) that ensure the prediction function f(x, W) to be as close as possible to the true solution (labeled input) y for any input x. We use gradient descent to find the weights *w* and biases *b* which minimize the cost function, C.
- ✓ To compute the gradient ∇C we need to compute the gradients ∇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.
- ✓ A way is to use mini-batched ***stochastic gradient descent*** to speed up learning. The idea is to estimate the gradient ∇C by computing a small sample of randomly chosen training inputs, refer to as a ***mini-batch*** *of input* (mini-batched SGD).
- ✓ By averaging over a small sample, we can quickly get a good estimate of the true gradient ∇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 ∇CXj roughly equals to the average over all ∇Cx.
- ✓ Then, another randomly chosen mini-batch are selected and trained, until all the training inputs are used. It is said to complete an *epoch* (iteration) of training.

$$\frac{\sum_{j=1}^{m}\nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

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

$$b_l \to b_l' = b_l - \eta\frac{\partial C}{\partial b_l}.$$

DNN Forward Backward Calculation : Weights and bias

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

```
weights = tf.random_normal(shape, stddev=sigma)
```

2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$

```
grads = tf.gradients(ys=loss, xs=weights)
```

4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$

```
weights_new = weights.assign(weights - lr * grads)
```

5. Return weights

In one epoch

1. Pick a mini-batch (32)
2. Feed it to Neural Network
3. Forward path calculation
4. Calculate the mean gradient of the mini-batch
5. Use the mean gradient calculated in step 4 to update the weights
6. Repeat steps 1–5 for the mini-batches we created

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

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

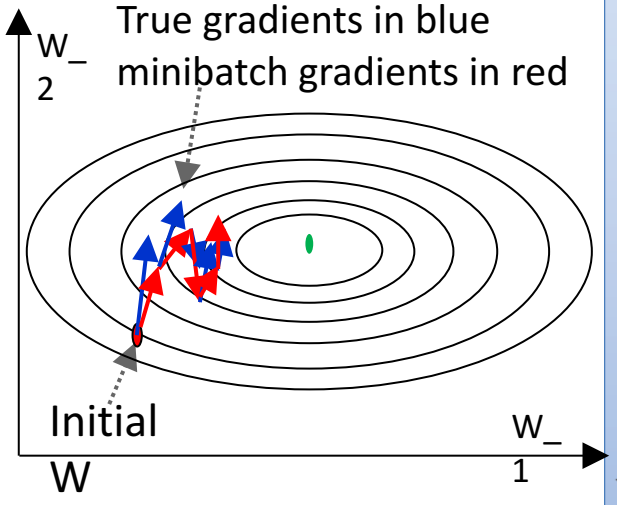
$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Optimization: Batch SGD

Instead of computing a gradient across the entire dataset with size N, divides the dataset into a fixed-size subset ("minibatch") with size m, and computes the gradient for each update on this smaller batch:

(N is the dataset size, m is the minibatch size)

$$\mathbf{g} = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} E_i(\mathbf{x}^{(n_i)}, y^{(n_i)}, \boldsymbol{\theta}),$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \mathbf{g},$$



Given: training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
Initialize all $\Theta^{(l)}$ randomly (NOT to 0!)
Loop // each iteration is called an epoch
  Loop // each iteration is a mini-batch
    Set $\Delta_{i,j}^{(l)} = 0 \quad \forall l, i, j$ (Used to accumulate gradient)
    Sample m training instances $\mathcal{X} = \{(\mathbf{x}'_1, y'_1), \ldots, (\mathbf{x}'_m, y'_m)\}$ without replacement
    For each instance in $\mathcal{X}$, $(\mathbf{x}_k, y_k)$:
      Set $\mathbf{a}^{(1)} = \mathbf{x}_k$
      Compute $\{\mathbf{a}^{(2)}, \ldots, \mathbf{a}^{(L)}\}$ via forward propagation
      Compute $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y_k$
      Compute errors $\{\boldsymbol{\delta}^{(L-1)}, \ldots, \boldsymbol{\delta}^{(2)}\}$
      Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
    Compute mini-batch regularized gradient $D_{ij}^{(l)} = \begin{cases} \frac{1}{m}\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{m}\Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$
    Update weights via gradient step $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$
  Until all training instances are seen
Until weights converge or max #epochs is reached

1875 TIMES OF BATCH ITERATION

BATCH LOOP

Momentum

A method which is supposed to help learn faster in the face of noisy gradients (as in our case)

We introduce a new variable which can be thought of as the velocity of learning.

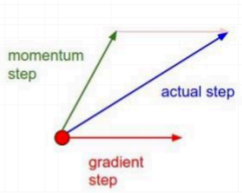
The velocity is basically an exponentially decaying average of the step size:

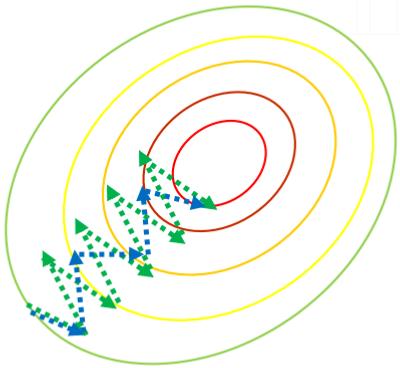
$$v_{j_t} = \gamma v_{j_{t-1}} + \alpha \frac{\partial E_t}{\partial w_j}$$
$$w_j = w_j - v_{j_t}$$

This is similar to a ball rolling down the gradient, and having momentum in a certain direction.

Size of steps depends on size of gradients, but also on how aligned they are.

Common value for $\gamma = 0.9$, which can be thought of as friction.





Global loss function

Mini-Batch Gradient Descent

Mini-Batch Gradient Descent with momentum

Adaptive learning rates + momentum

1. ADAM (Adaptive Moment Estimation):

$$v_{j_t} = \beta_1 v_{j_{t-1}} + (1-\beta_1)\frac{\partial E}{\partial w_j}$$
$$m_{j_t} = \beta_2 m_{j_{t-1}} + (1-\beta_2)\frac{\partial E^2}{\partial w_j}$$
$$w_j = w_j - \frac{\alpha}{\sqrt{m_{j_t}}} v_{j_t}$$

The hyperparameter in general:
- ✓ *beta1* = 0.9
- ✓ *beta_2* = 0.99.

2. NADAM(Nesterov-accelerated Adaptive Moment Estimation):
   1. Use ADAM with Nestrov momentum

https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

Basic Ideas
Typical Neural Network – MLP

STEP 1 : Model Definition (Network)

STEP 2 : Cost Function (MSE, CE)

Step 3 : Optimization Scheme (GRADIENT)

Step 4 : Numerical Implementation (fitting w,b)

Step 5 : Evaluation

Tensorflow Example

```
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(16, activation='relu'),
 tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Tensorflow is imported

Sets the mnist dataset to variable "mnist"

Loads the mnist dataset

Builds the layers of the model
4 layers in this model

```
model.compile(optimizer='sgd', loss='SparseCategoricalCrossentropy',
metrics=['accuracy'])

model.summary()

logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-
%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)
```

Loss Function : Y the end of the NN

Compiles the model with the SGD optimizer

Print summary

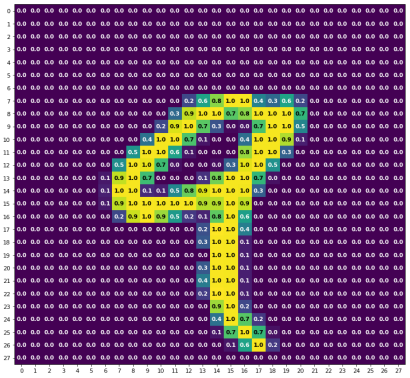
Use tensorborad

```
model.fit(x_train, y_train, epochs=5, batch_size=10,
validation_data=(x_test, y_test),  callbacks=[tensorboard_callback])

model.evaluate(x_test,  y_test, verbose=2)
%tensorboard --logdir logs
```

Adjusts model parameters to minimize the loss
Tests the model performance on a test set

TensorFlow 2.0 : Forward Pass



```
tf.reshape(tensor, shape)
#example
tf.reshape(x_train[0], [784])
```

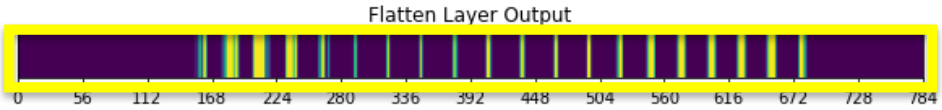
Flatten a single image

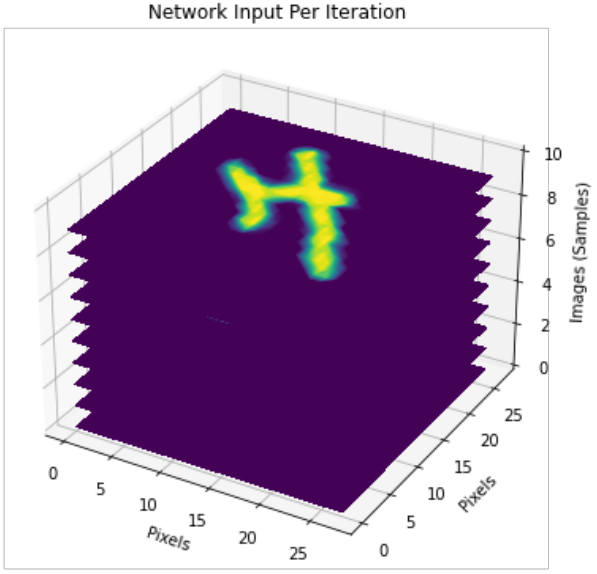
```
tf.reshape(tensor, shape)
#example
tf.reshape(x_train[0:10], [10: 784])
```

Flatten a batch of images

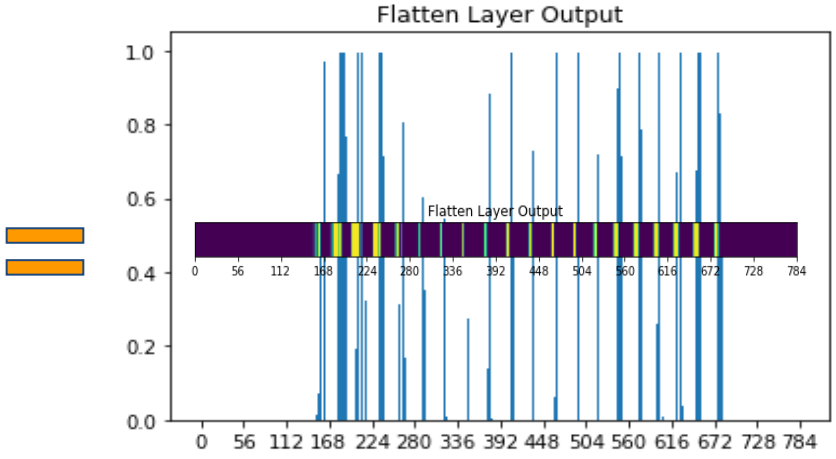
Flatten

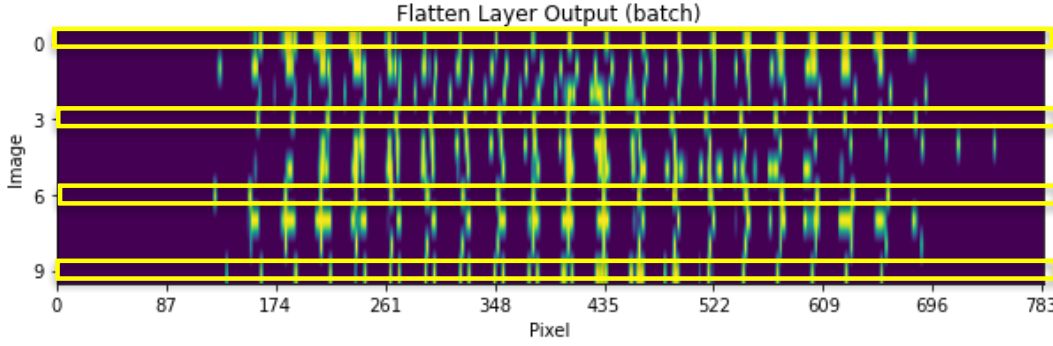
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28))
```





=

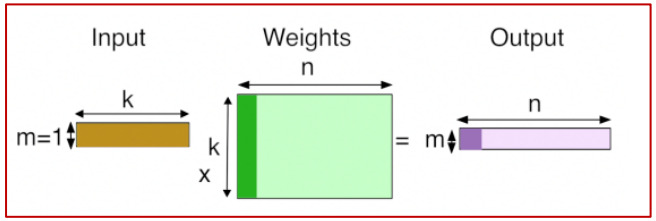




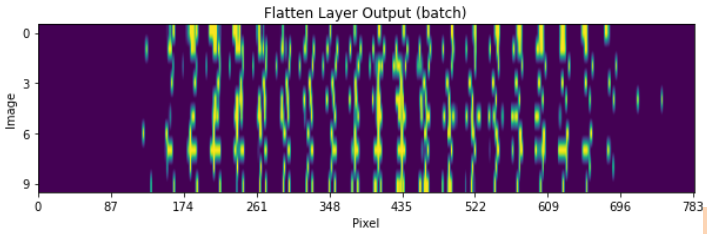
TensorFlow 2.0 : Forward Pass

1ST Fully Connected Layer with 16 neurons
Matrix multiplication

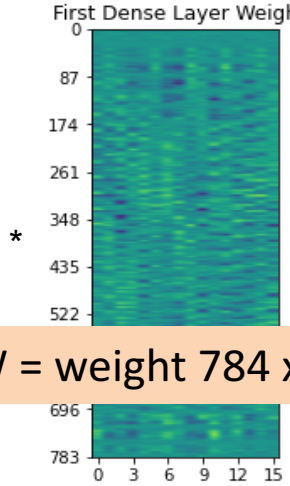
```
...
 tf.keras.layers.Flatten(input_shape=(28, 28)),
 tf.keras.layers.Dense(16, activation='relu'),
...
```



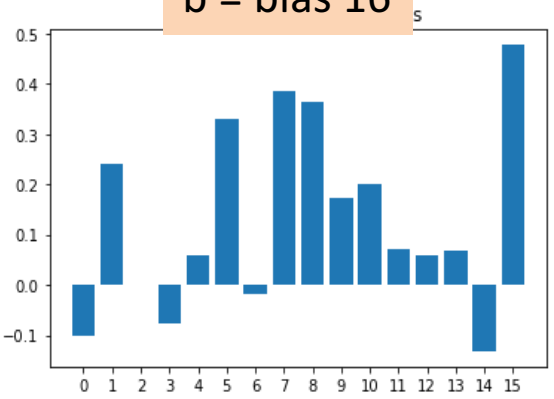
$$a^1 = \text{reLU}(X * W^1 + b^1)$$



X = Input 10 x 784



W = weight 784 x 16

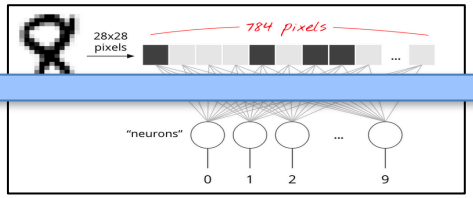


b = bias 16

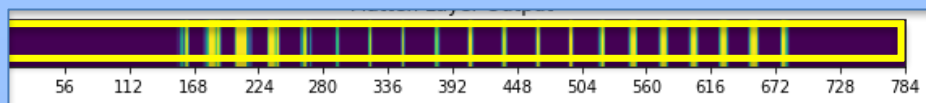
Weights and bias are initialized with random values

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



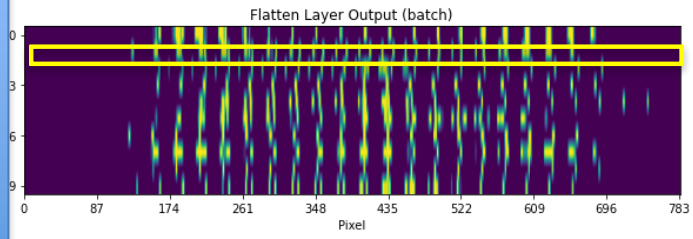
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

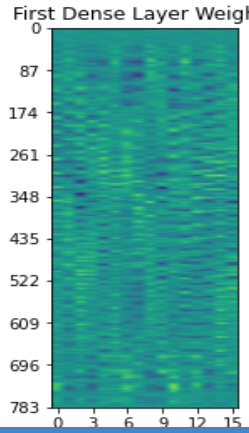
16 neurons
784 x 16 W matrix

10 x 16 output matrix

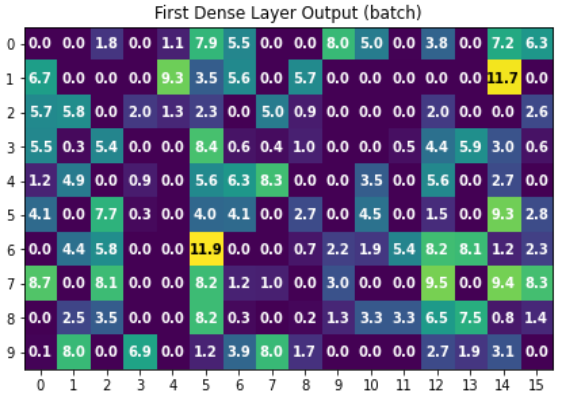
Input to 2nd layer



*



=

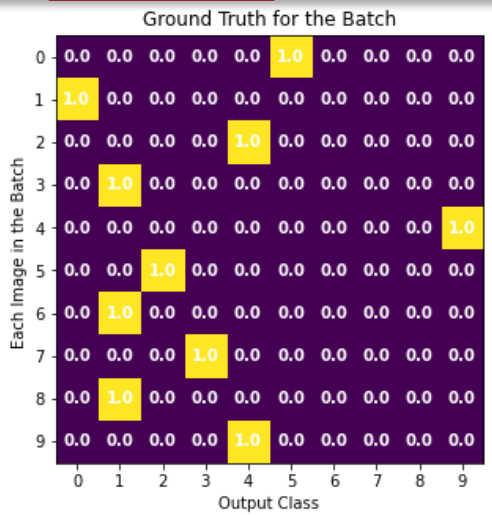


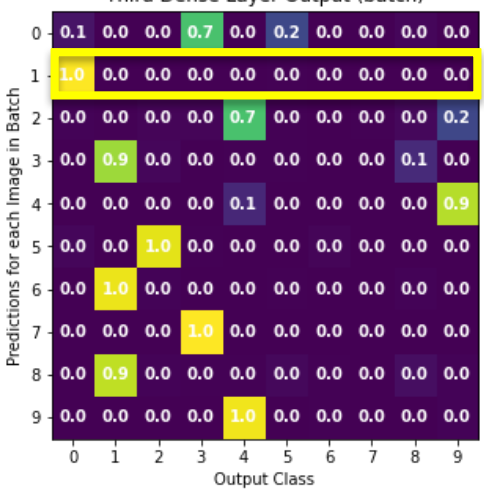
10 x 16 matrix

10 x 10 labeled matrix

10 x 10 output matrix

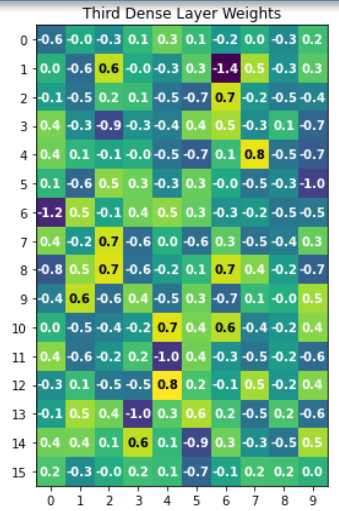
*





=

10 x 10 output matrix



16 neurons
16 x 10
W matrix

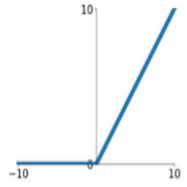
TensorFlow 2.0 : Forward Pass

Second Fully Connected Layer with 16 neurons

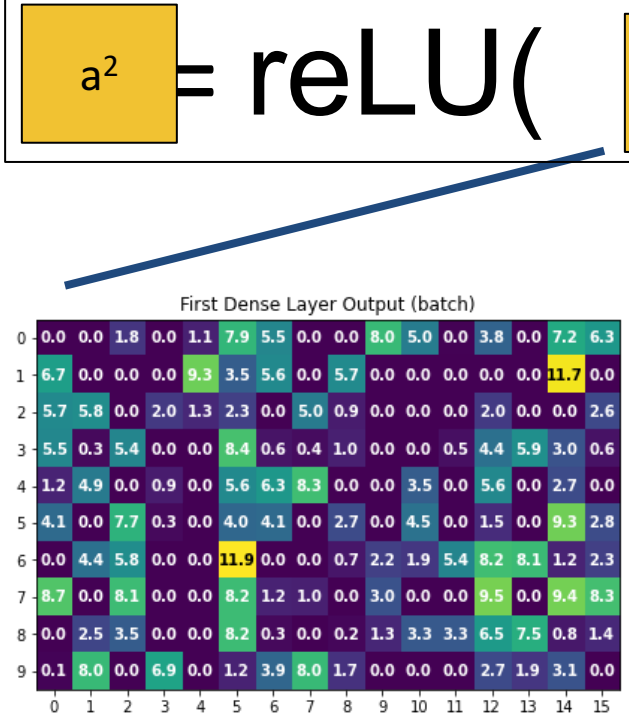
```
model = tf.keras.models.Sequential([
 tf.keras.layers.Flatten(input_shape=(28, 28)),
 tf.keras.layers.Dense(16, activation='relu'),
 tf.keras.layers.Dense(16, activation='relu'),
...
```

ReLU

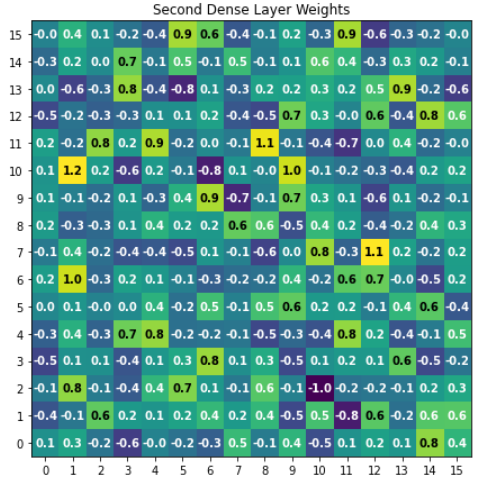
$\max(0, x)$



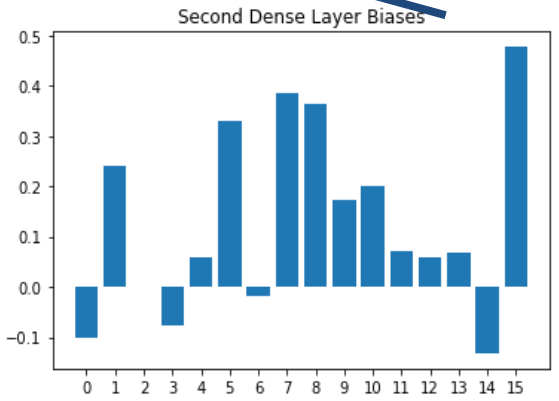
$a^2 = \text{reLU}(a^1 * W^2 + b^2)$



*



+



TensorFlow 2.0 : Forward Pass

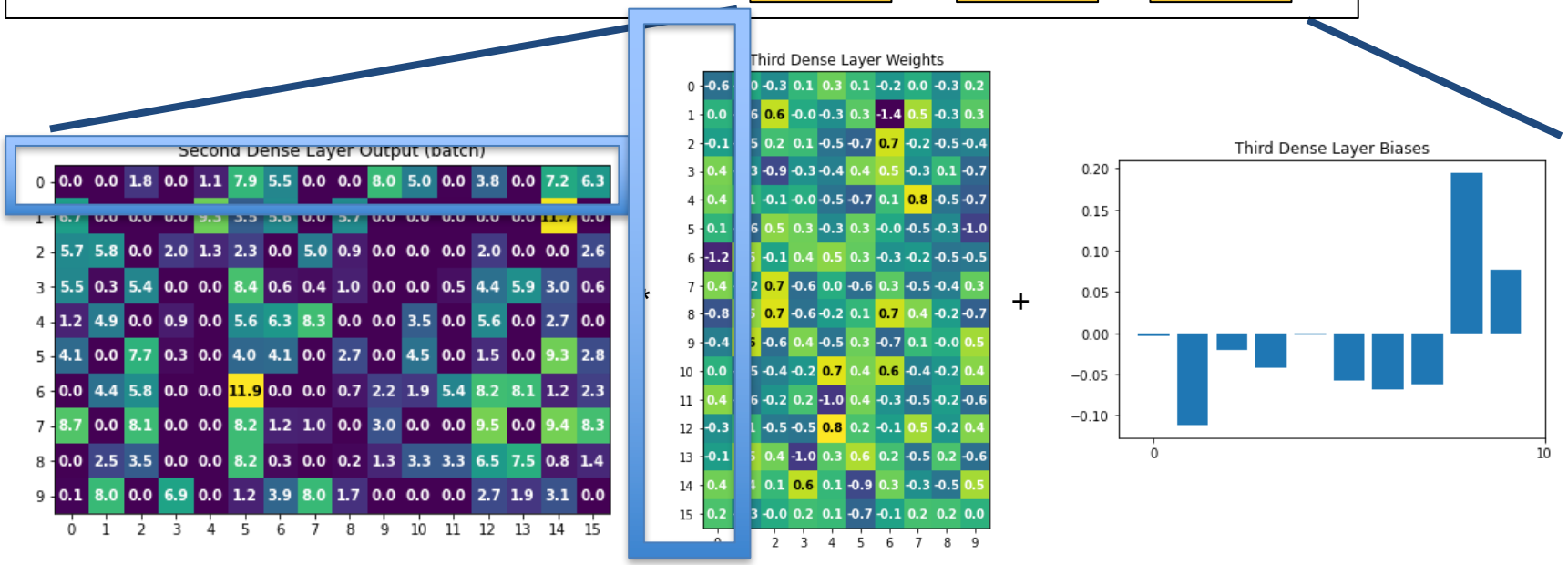
Final Fully Connected Layer with 10 neurons : 10 classes

$$\text{Softmax} = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- ✓ The softmax function is a function that turns a vector of K real values into values between 0 and 1, and the values sum up to 1, so that they can be interpreted as probabilities.
- ✓ This is because the softmax is a generalization of logistic regression that can be used for multi-class classification.

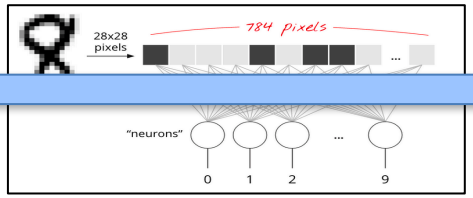
https://deepai.org/machine-learning-glossary-and-terms/softmax-layer

$a^3 = \text{softmax}(a^2 * W^3 b^3$

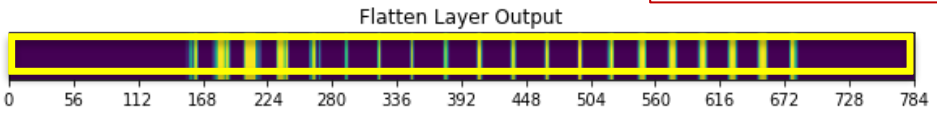


Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



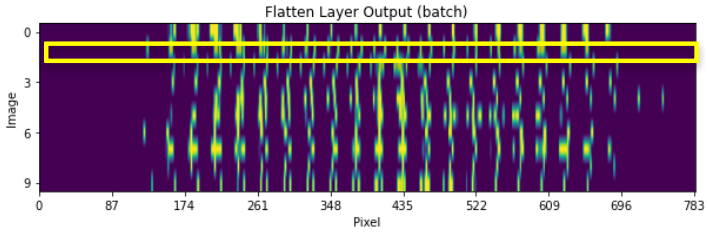
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

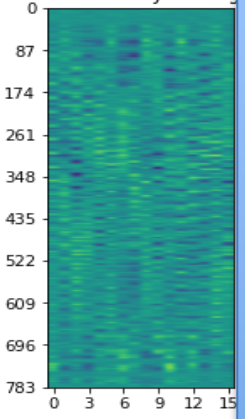
16 neurons
784 x 16 W matr

10 x 16 output matrix

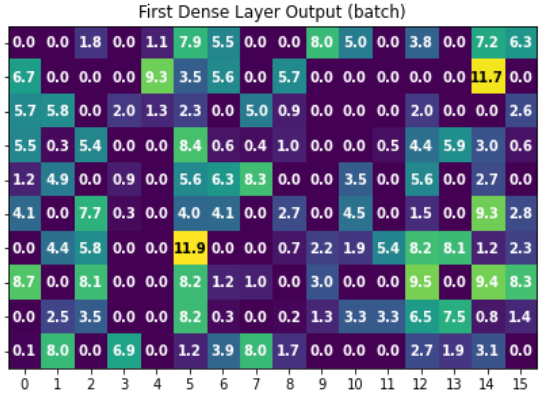
Input to 2nd layer



*



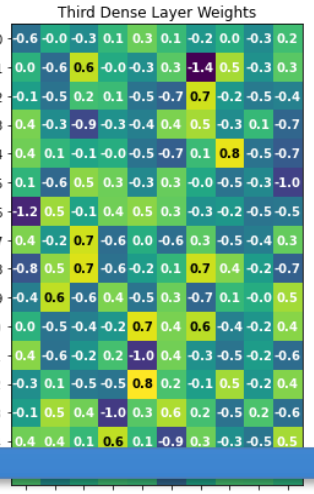
=



Input to 2nd layer

10 x 16 matrix

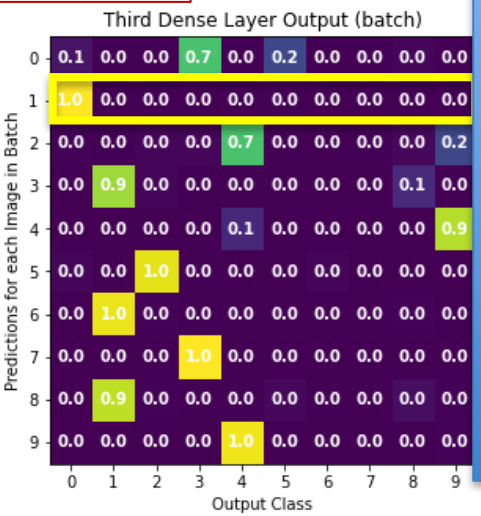
*



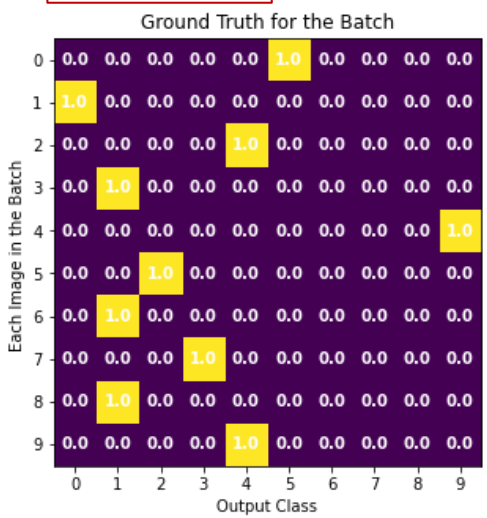
16 neurons
16 x 10 W matrix

=

10 x 10 output matrix



10 x 10 output matrix



10 x 10 labeled matrix

Basic Ideas
Typical Neural Network – MLP

STEP 1 : Model Definition (Network)

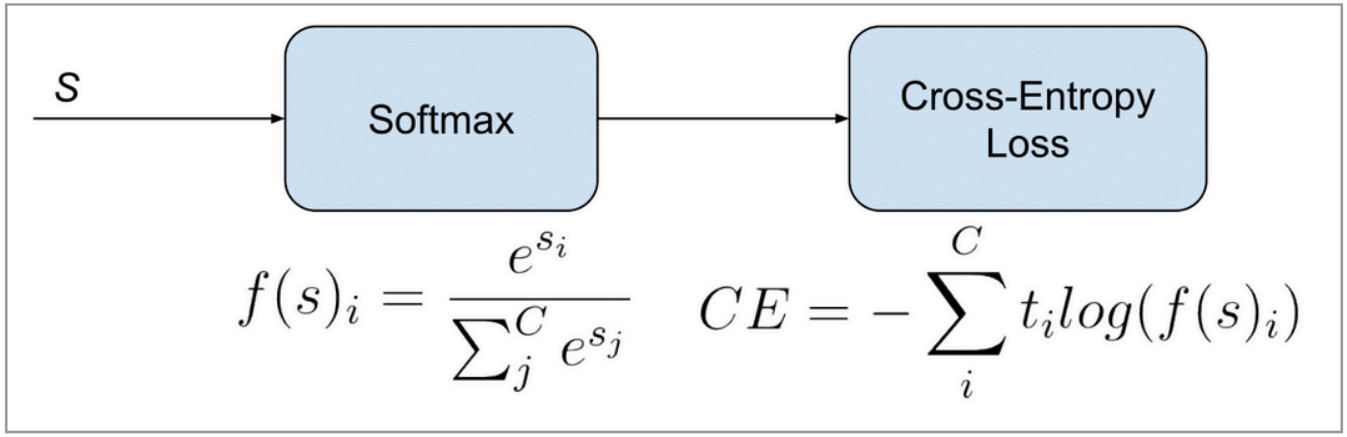
STEP 2 : Cost Function (MSE, CE)

Step 3 : Optimization Scheme (GRADIENT)

Step 4 : Numerical Implementation (fitting w,b)

Step 5 : Evaluation

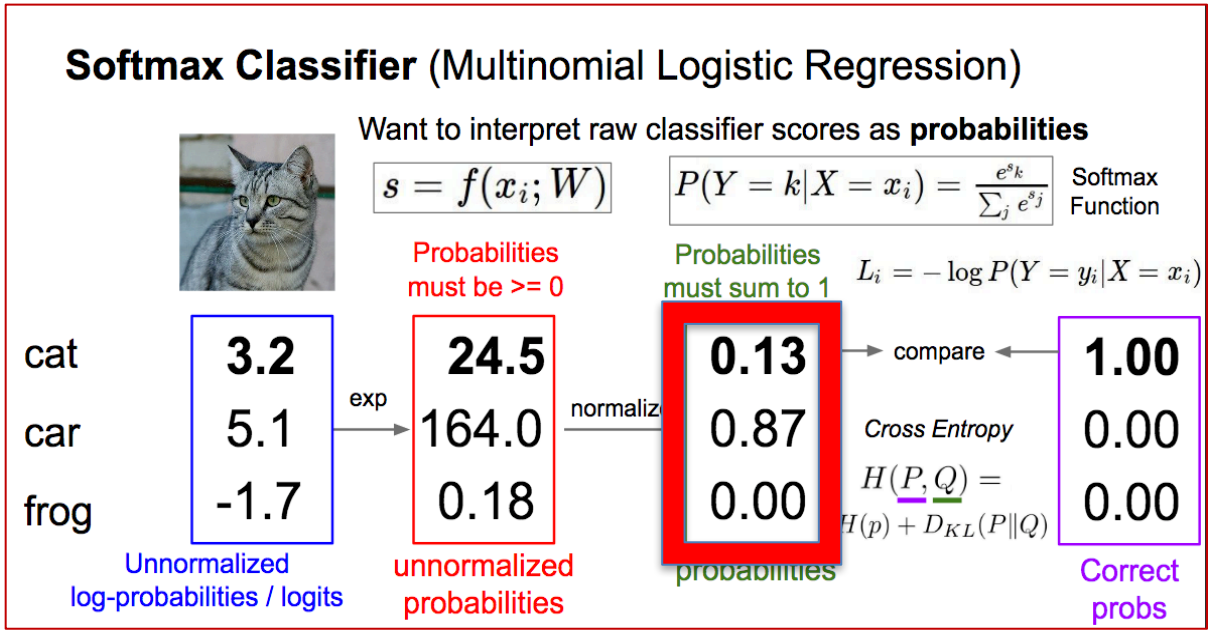
Categorical (Softmax) Cross Entropy Loss (Statistical Learning)



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

T = 1

ti and si are the groundtruth (label) and the computed score for each class i in C.



e^3.2 = 24.5

e^5.1 = 164.0

e^-1.7 = 0.18

188.68

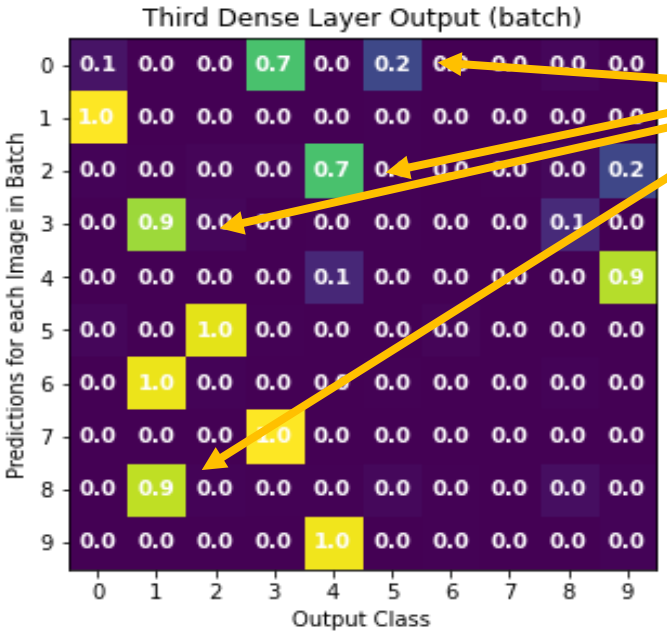
Softmax = 24.5 / 188.68 = 0.13

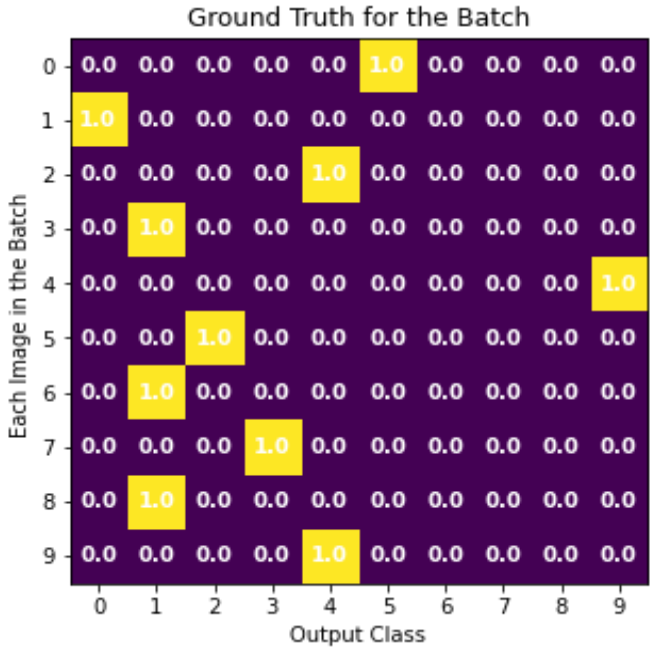
Softmax = 164 / 188.68 = 0.87

Softax = -1.7 / 188.68 = 0.00

https://gombru.github.io/2018/05/23/cross_entropy_loss/

Evaluating the Error





Z

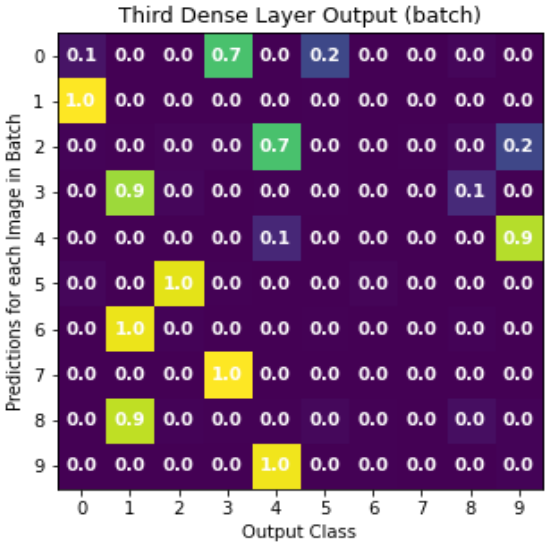
Categorical Cross Entropy Loss

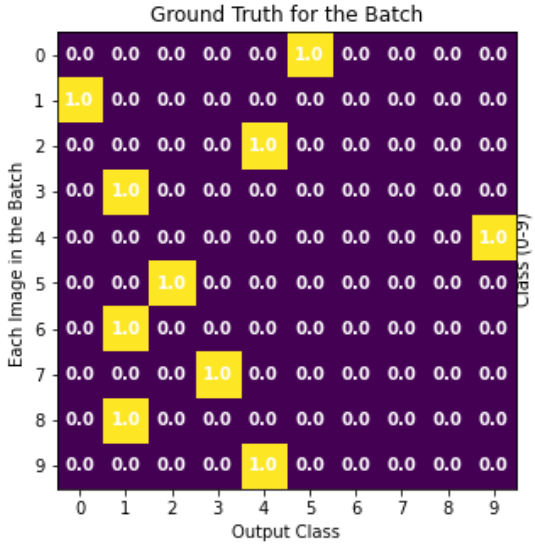
$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

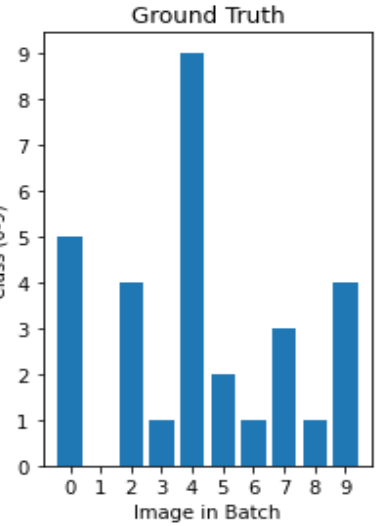
L (i) = - log0.2 = 1.61	Image 0 = 5
- log1.0 = 0	Image 1 = 0
- log0.7 = 0.356	Image 2 = 4
- log0.9 = 0.105	Image 3 = 1
- log0.9 = 0.105	Image 4 = 9
- log1.0 = 0	Image 5 = 2
- log1.0 = 0	Image 6 = 1
- log1.0 = 0	Image 7 = 3
- log0.9 = 0.105	Image 8 = 2
- log1.0 = 0	Image 9 = 4

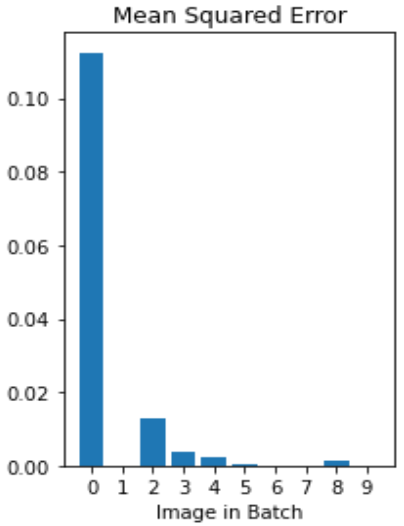
Evaluating the Error

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$









Categorical Cross Entropy Loss

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{sj}}\right)$$

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ Softmax Function

$$L_i = -\log P(Y = y_i|X = x_i)$$

	Probabilities must be >= 0	exp	Probabilities must sum to 1	normalize		compare	
cat	**3.2**		**24.5**		**0.13**		**1.00**
car	5.1		164.0		0.87		0.00
frog	-1.7		0.18		0.00		0.00
	Unnormalized log-probabilities / logits		unnormalized probabilities		probabilities		Correct probs

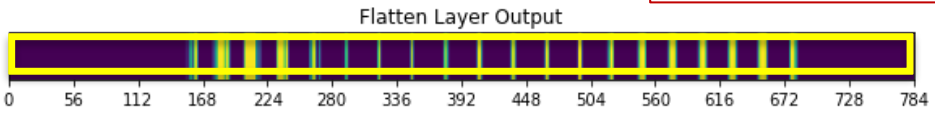
Cross Entropy

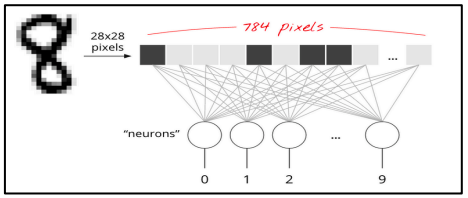
$$H(P, Q) = H(p) + D_{KL}(P\|Q)$$

L (i) = - log0.2 = 1.61
- log1.0 = 0
- log0.7 = 0.356
- log0.9 = 0.105
- log0.9 = 0.105
- log1.0 = 0
- log1.0 = 0
- log1.0 = 0
- log0.9 = 0.105
- log1.0 = 0

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784

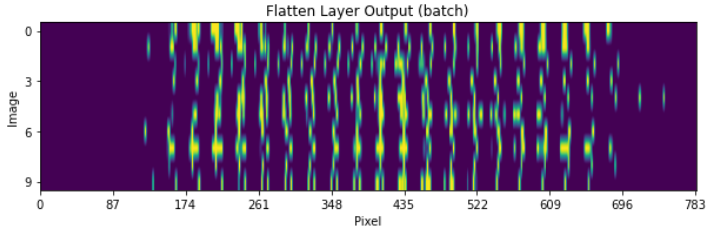




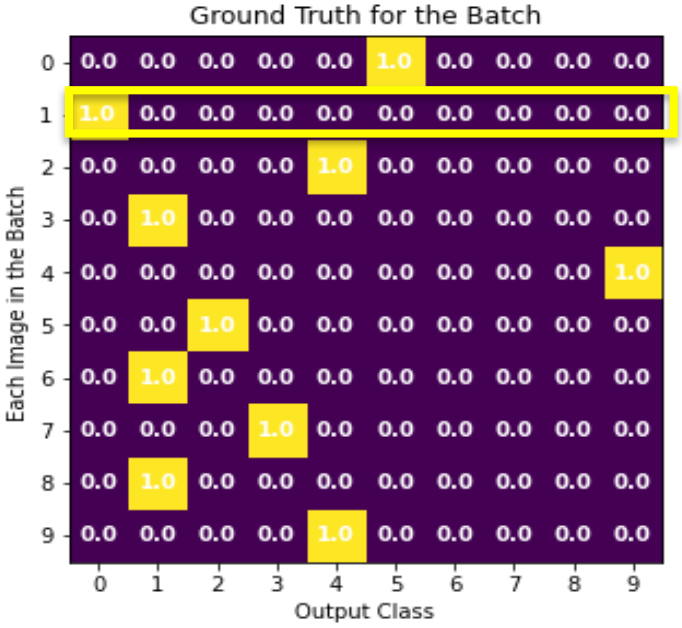
An image of 28x28 pixels

Input to 1st layer

10 images : batch of 10
10 x 784 matrix

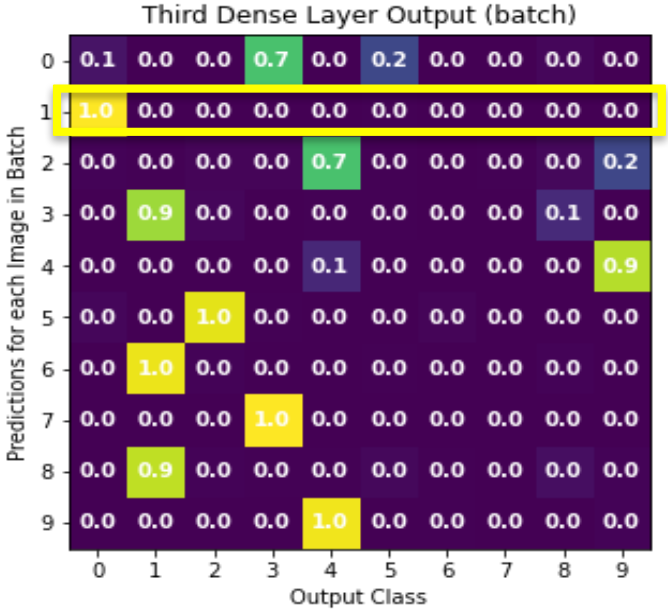


10 x 10 labeled matrix



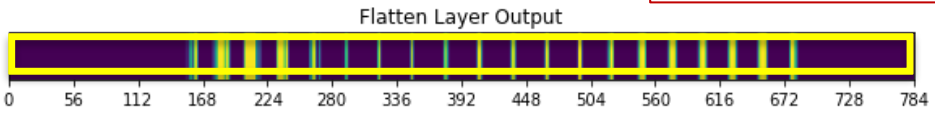
Comparing Computed NN results To Labeled results (target)

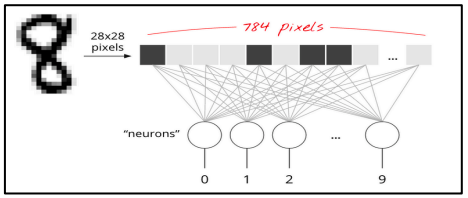
10 x 10 output matrix



Summary : Flow of MLP Dense layer NN

1 image = a vector of 784





An image of 28x28 pixels

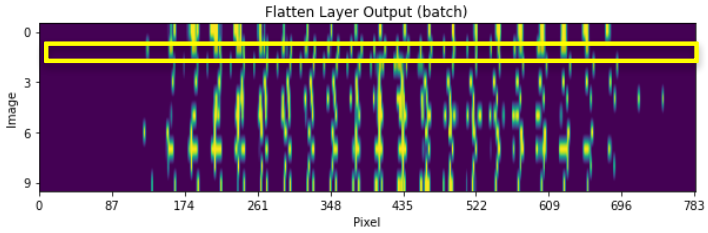
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

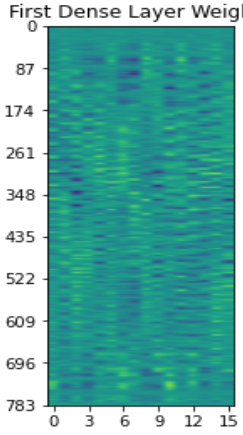
16 neurons
784 x 16 W matrix

10 x 16 output matrix

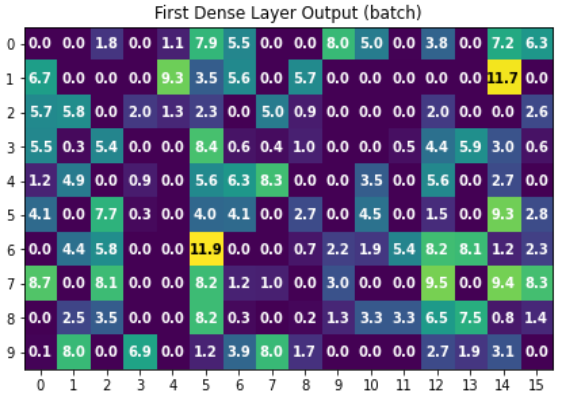
Input to 2nd layer



*

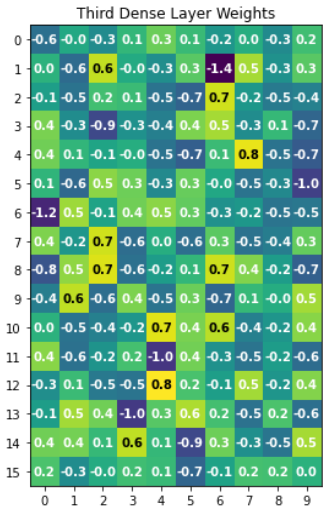


=



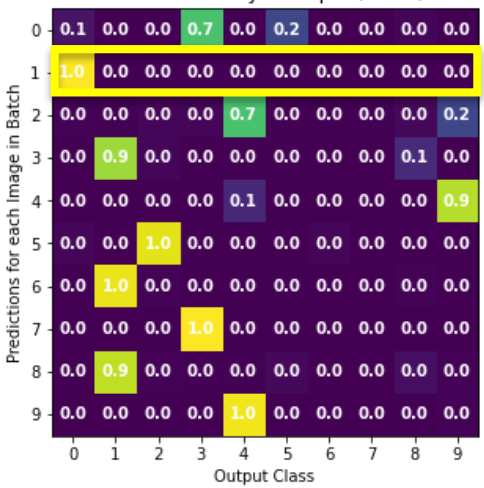
10 x 16 matrix

*

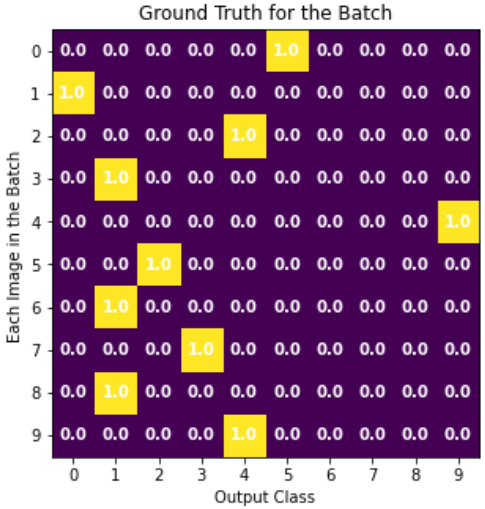


16 neurons
16 x 10
W matrix

=



10 x 10 output matrix



10 x 10 labeled matrix

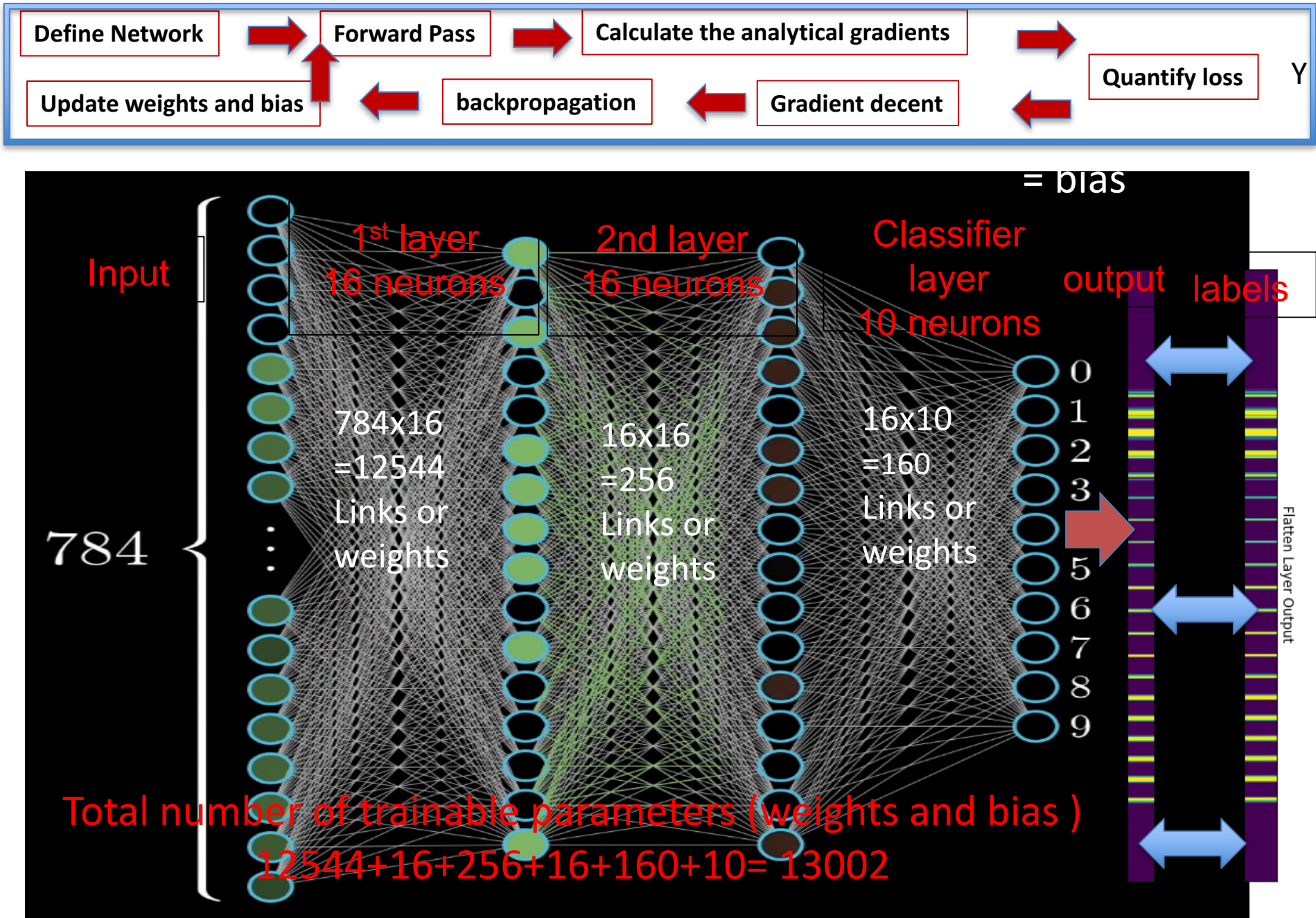
Tensorflow Example

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%

```
2020-07-29 19:53:40.592860: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1108]      0
2020-07-29 19:53:40.592871: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1121] 0:   N
2020-07-29 19:53:40.593073: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:981] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2020-07-29 19:53:40.593535: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:981] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2020-07-29 19:53:40.593973: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:981] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2020-07-29 19:53:40.594387: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1247] Created TensorFlow device (/job:lo
calhost/replica:0/task:0/device:GPU:0 with 7219 MB memory) -> physical GPU (device: 0, name: GeForce GTX 1080, pci bus
id: 0000:26:00.0, compute capability: 6.1)
2020-07-29 19:53:40.623075: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x55d981198b80 initialized fo
r platform CUDA (this does not guarantee that XLA will be used). Devices:
2020-07-29 19:53:40.623096: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): GeForce GTX
1080, Compute Capability 6.1
2020-07-29 19:53:52.215159: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic
 library libcublas.so.10
Epoch 1/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.2982 - accuracy: 0.9127
Epoch 2/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.1433 - accuracy: 0.9571
Epoch 3/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.1097 - accuracy: 0.9663
Epoch 4/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.0893 - accuracy: 0.9730
Epoch 5/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.0777 - accuracy: 0.9750
313/313 - 0s - loss: 0.0727 - accuracy: 0.9776
```

Batch = 60000/1875 = 32, exhaust it => 1 epoch

```
model.fit(x_train, y_train, epochs=3, batch_size=10)
```



DNN Model

Applications

+

Data Ensemble + Input

+

It's all about

linear algebra calculations

DNN in particular

+

Algorithms, Software

+

Output + Data Analysis

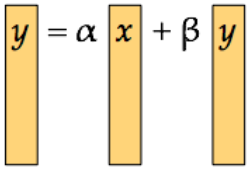
+

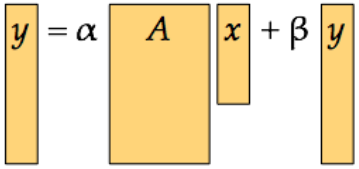
Hardware

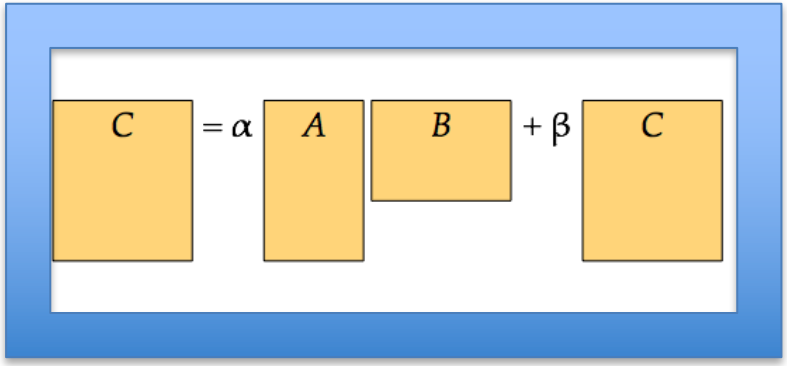
LOTS of MM (BLAS3), Parallel !!

Computational Intensity = FLOPS/ Memory Access

- ✓ 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

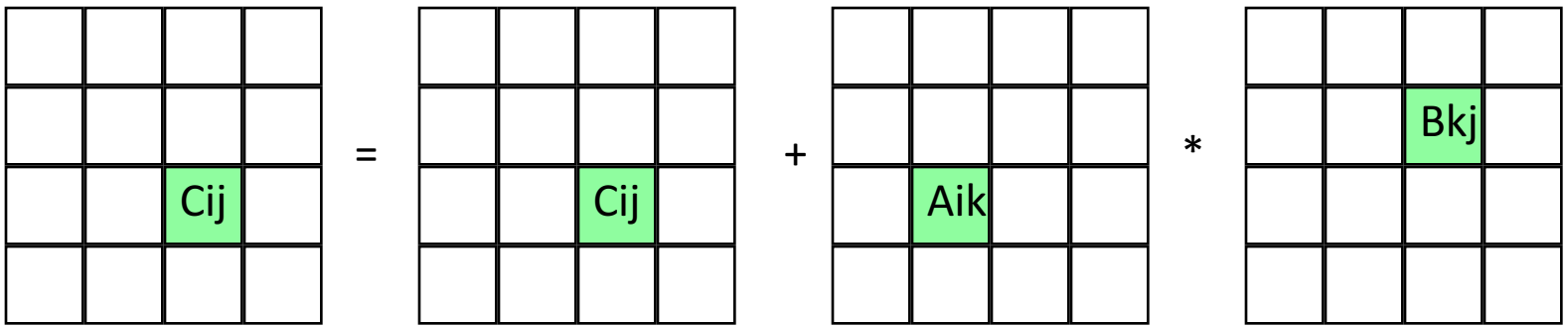






Optimized : Block MM

- q = f/m = (2*n^3) / ((2*N + 2) * n^2) ~ n / N
- If N is equal to 1, the algorithm is ideal. However, N is bounded by the amount of fast cache memory. However, N can be taken independently to the size of matrix, n.
- The optimal value of N = sqrt (size of fast memory / 3)



$$Cij = Cij + \sum_{k=1}^{n} Aik * Bkj$$

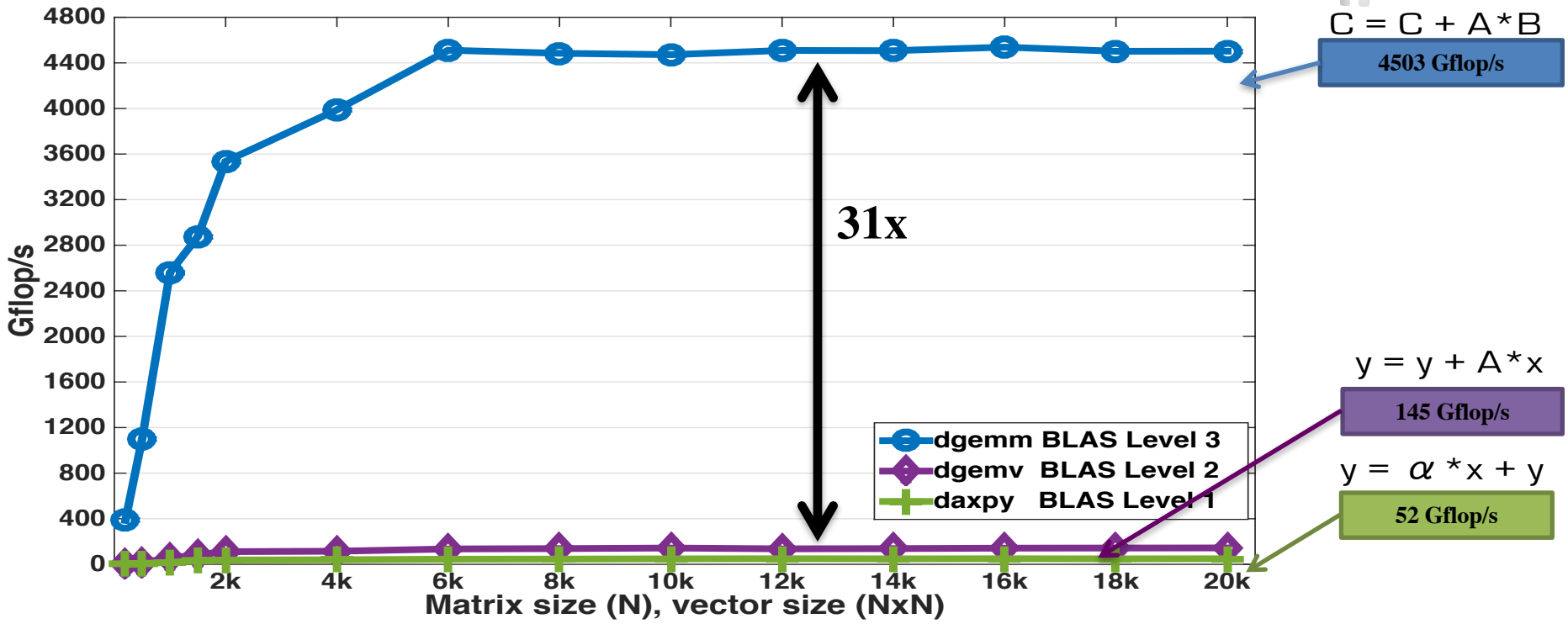
Level 1, 2 and 3 BLAS on GPU

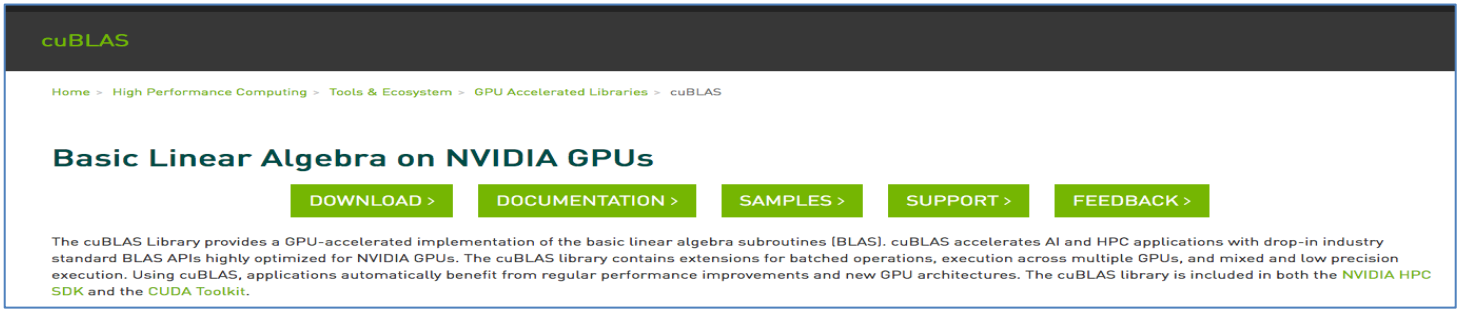
Joint Institute for Computational Sciences

INNOVATIVE COMPUTING LABORATORY

Nvidia P100, The theoretical peak double precision is 4700 Gflop/s, CUDA version 8.0

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





LAB 2
Problem 4

Write a python code to compute C=AXB and plot a curve of the FLOPS against the matrix size N (take N=2000, 4000, 6000) using single precision when the computation is done on a CPU. DO the same on the GPU

```
import numpy as np
import time

A = np.random.rand(2000, 2000).astype('float32')
B = np.random.rand(2000, 2000).astype('float32')
%timeit np.dot(A,B)
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm2000 = timeend - timestt
gf2000 = 2*2*2*2/tmm2000
print('time = ',tmm2000)
print('GFLOPS = ', gf2000)

A = np.random.rand(4000, 4000).astype('float32')
B = np.random.rand(4000, 4000).astype('float32')
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm4000 = timeend - timestt
gf4000 = 2*4*4*4/tmm4000
print('time = ',tmm4000)
print('GFLOPS = ', gf4000)
```

```
A = np.random.rand(6000, 6000).astype('float32')
B = np.random.rand(6000, 6000).astype('float32')
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm6000 = timeend - timestt
gf6000 = 2*6*6*6/tmm6000
print('time = ',tmm6000)
print('GFLOPS = ', gf6000)

import torch
A = torch.randn(6000, 6000).cuda()
B = torch.randn(6000, 6000).cuda()
timestt = time.time()
C=torch.matmul(A,B)
timeend = time.time()
tmm6000 = timeend - timestt
gf6000 = 2*6*6*6/tmm6000
print('time = ',tmm6000)
print('GFLOPS = ', gf6000)
```

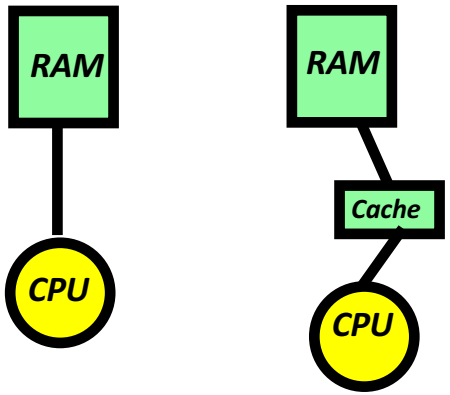
```
1 loop, best of 5: 228 ms per loop
time =  0.2360849380493164
GFLOPS =  67.7722184744277
time =  1.8578176498413086
GFLOPS =  68.8980428251037
time =  6.155170440673828
GFLOPS =  70.18489644824643
```

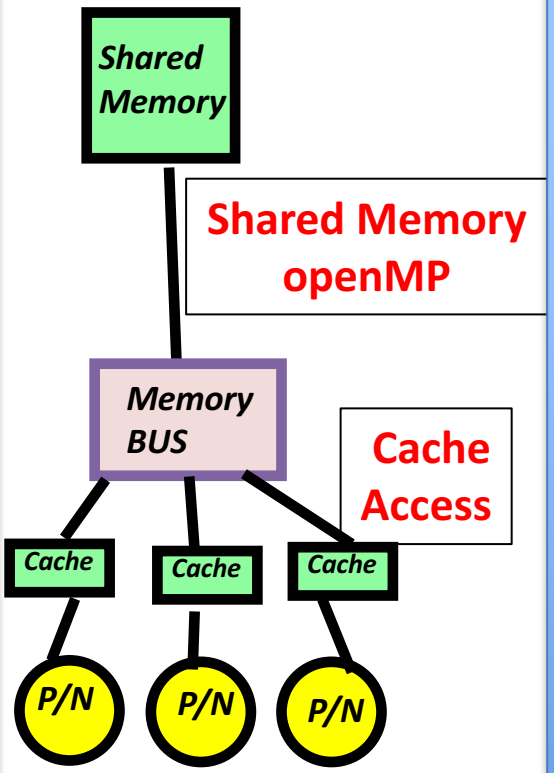
```
1 loop, best of 5: 214 ms per loop
time =  0.22870469093322754
GFLOPS =  69.95921218192831
time =  1.8288803100585938
GFLOPS =  69.98817762759944
time =  6.09820103645324 7
GFLOPS =  70.84056386754575
```

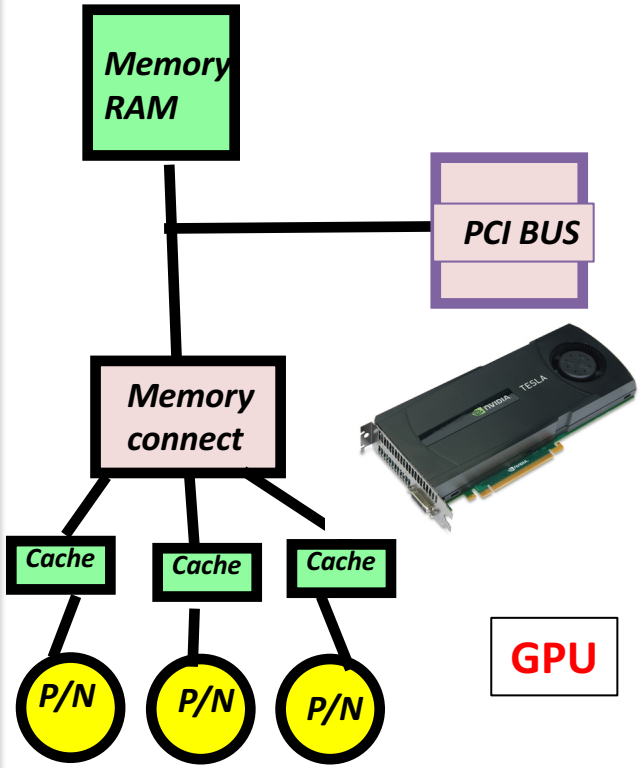
```
time =  6.029452800750732
GFLOPS =  71.6482928510878
time =  0.06104087829589844
GFLOPS =  7077.224510202169
```

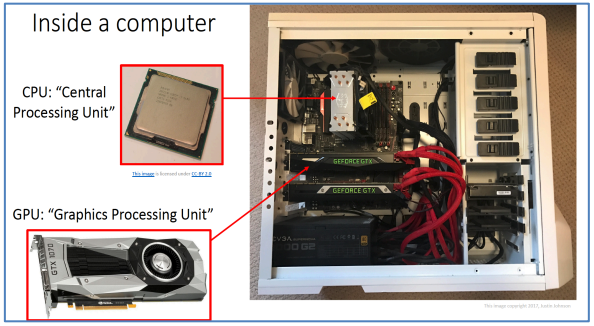
Simple story of Computers

Major Bottleneck - Communication, Memory Access

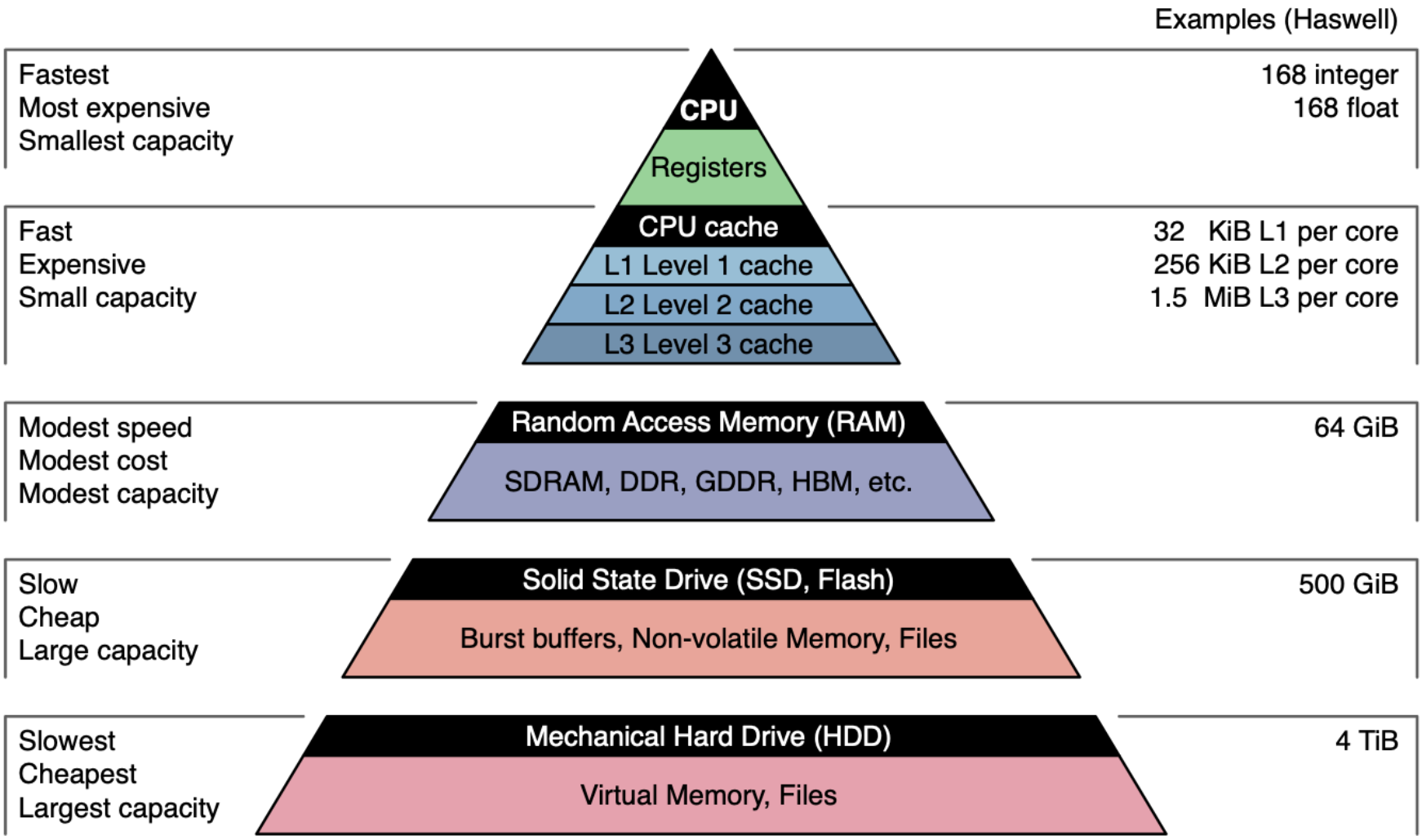








Memory hierarchy

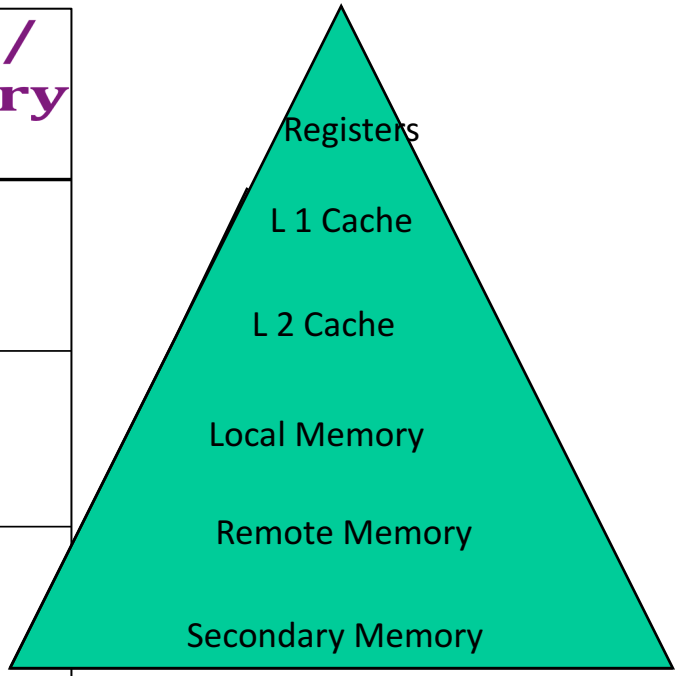


Adapted from illustration by Ryan Leng

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

BLAS	Memory Refs	Flops	Flops/ Memory Refs
Level 1 $y=y+\alpha x$	$3n$	$2n$	$2/3$
Level 2 $y=y+Ax$	n^2	$2n^2$	2
Level 3 $C=C+AB$	$4n^2$	$2n^3$	$n/2$



Modeling Cycle and Emergent Technology

Big Science, Big Data, AI
Learn the pattern and behavior hidden in data
Predict or classify new entities based on the learned pattern

A growing field with evolving technology
Statistical Machine Learning →
Deep Neural Network → Supercomputing (HPC)
→ Real Time Applications

Time (Speed) and Length (Memory) Scales

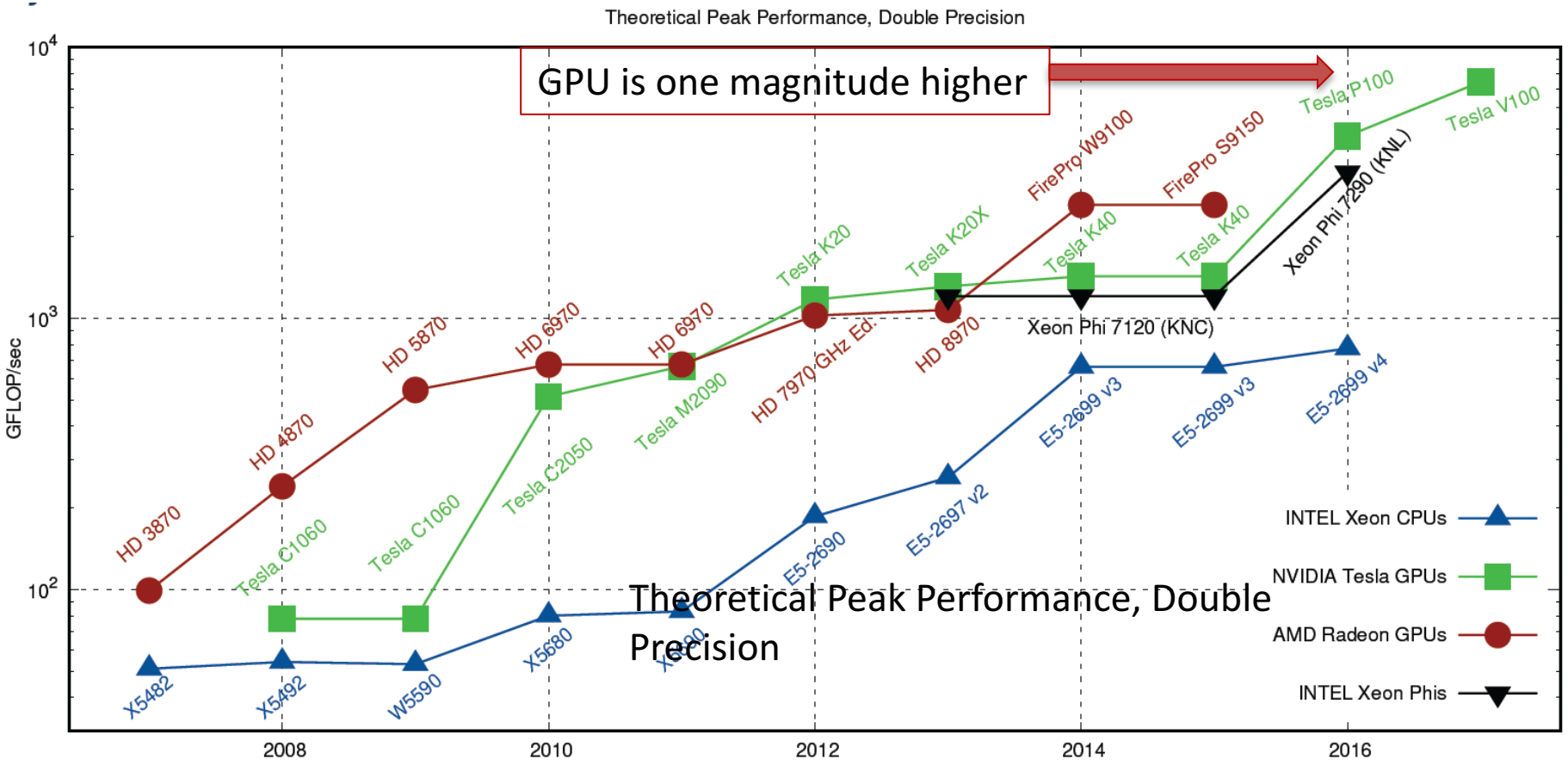
Big Data → Deep Learning → Computing capacity
Combining the Learned the pattern and behavior hidden in sensor or computed data to predict complicated phenomena

Fast enough to get approximate results (time scale)
Enough memory to process the huge amount of data (length scale)

A growing field with evolving technology
Parallel + Data + model-based simulation --> Big Iron (supercomputer, clusters of computers)

Why GPUs? Time Scale !

✓ Far better peak performance than CPUs, per unit of power consumption

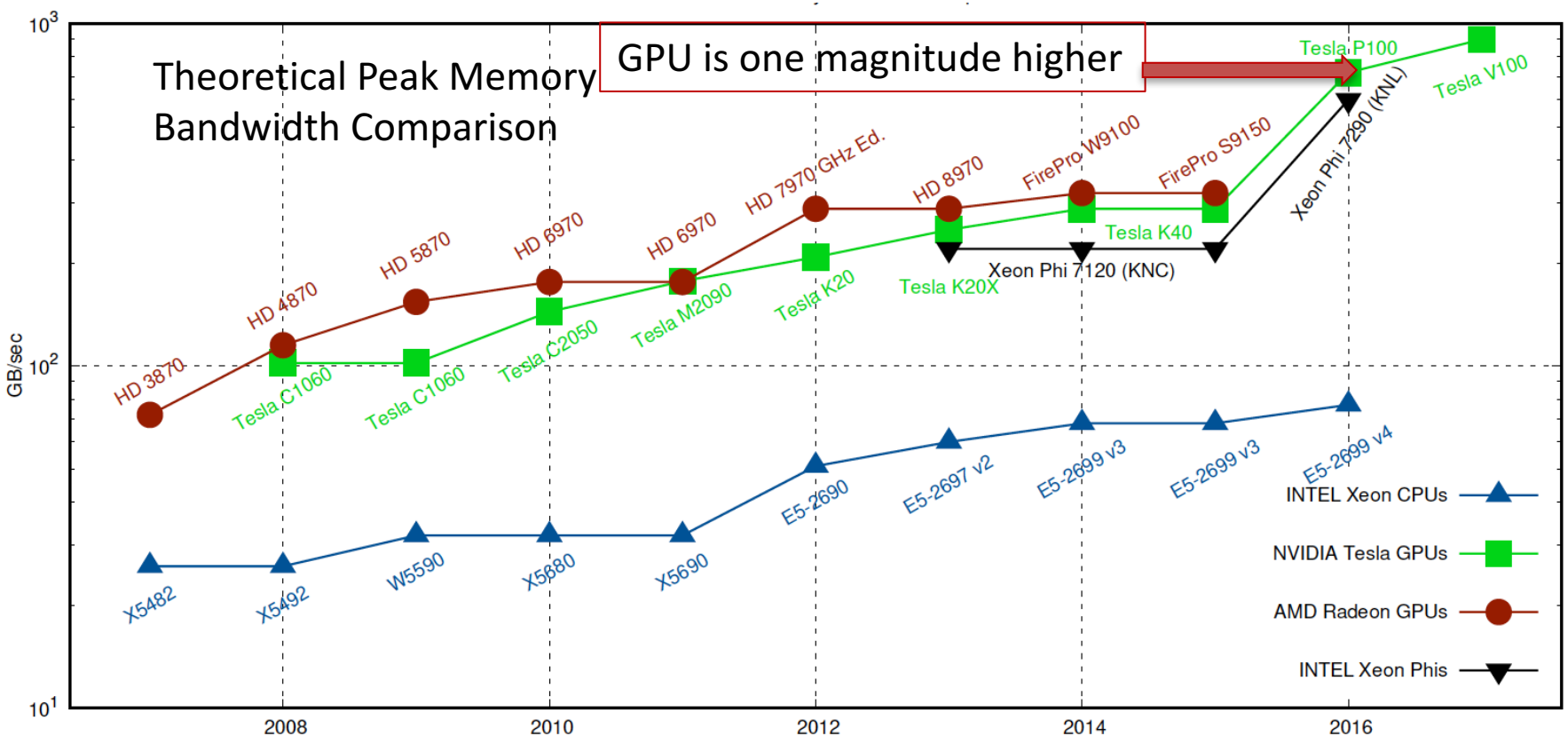


Floating-Point Operations per Second for the CPU and GPU

PRACE training course "GPU programming with CUDA" @ JSC April 2017

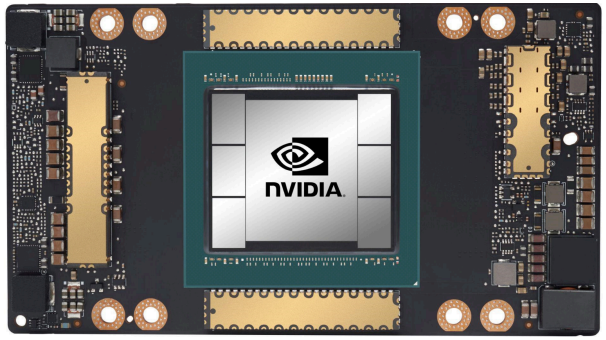
Why GPUs? ~Length Scale !

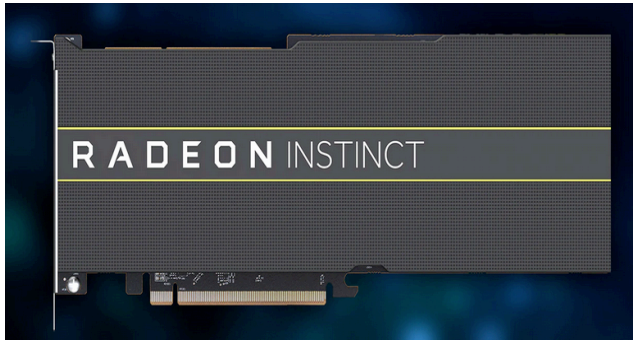
✓ Fast processing along with high memory bandwidth

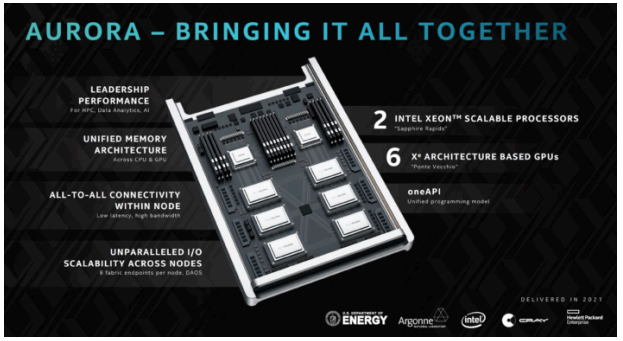


Memory Bandwidth for the CPU and GPU

GPU : Software Stack : Single Precision Computation







nvcc, openACC, CUDA
cuBLAS, cuDNN, NSIGHT,
grid, block, thread, warp

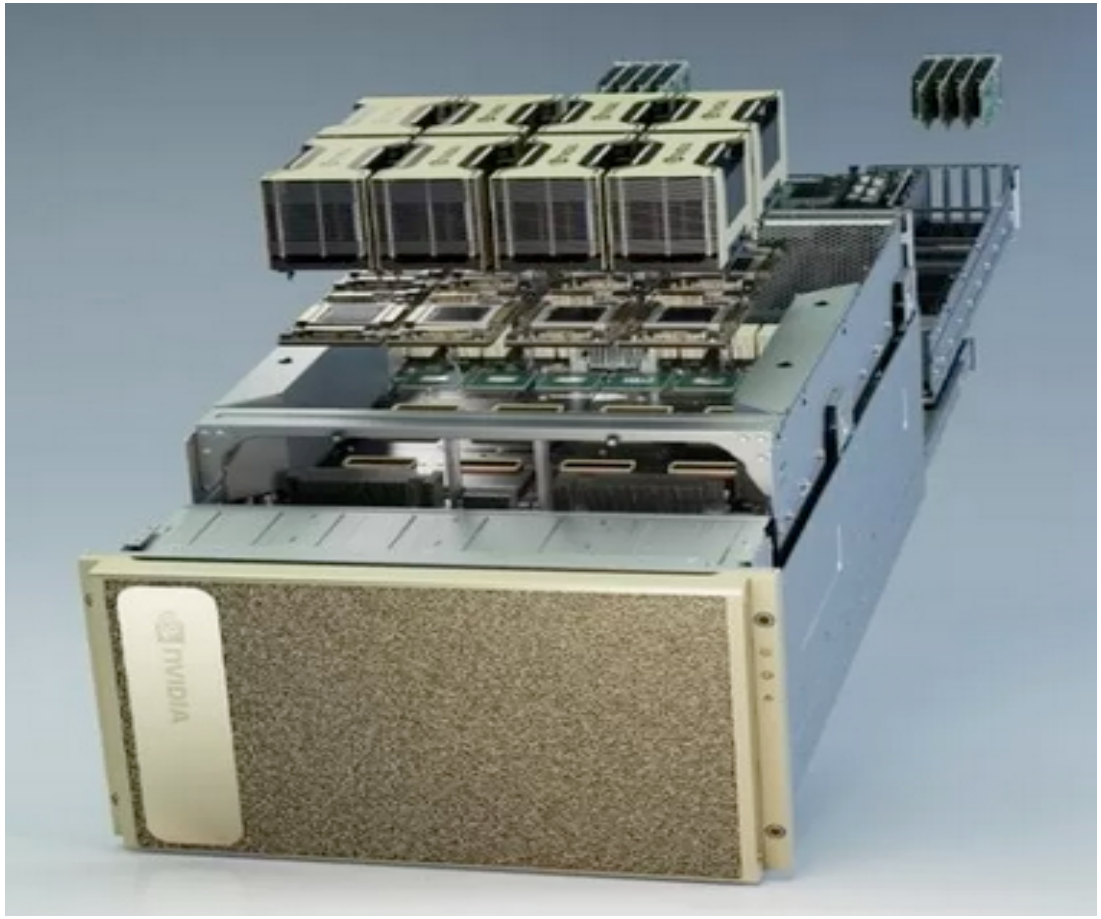
hipcc, openMP, HIP, rocprofiler
hipBLAS, Tensile (GEMM), ROCm
Grid, block, thread, wavefront

Intel, openMP,DPC++, oneAPI,
Vtune, sycl, oneMKL, , oneDNN.

	bits	precision		epsilon (2 × unit roundoff)	underflow (min normal)	overflow (max)
bfloat16	16	8 bits	≈ 2 digits	7.81×10^{-3}	1.18×10^{-38}	3.40×10^{38}
half	16	11 bits	≈ 3 digits	9.77×10^{-4}	6.10×10^{-5}	65504
single (float)	**32**	**24 bits**	**≈ 7 digits**	$\mathbf{1.19 \times 10^{-7}}$	$\mathbf{1.18 \times 10^{-38}}$	$\mathbf{3.40 \times 10^{38}}$
double	**64**	**53 bits**	**≈ 16 digits**	$\mathbf{2.22 \times 10^{-16}}$	$\mathbf{2.23 \times 10^{-308}}$	$\mathbf{1.79 \times 10^{308}}$
double-double	64x2	107 bits	≈ 32 digits	1.23×10^{-32}	2.23×10^{-308}	1.79×10^{308}
quad	128	113 bits	≈ 34 digits	1.93×10^{-34}	3.36×10^{-4932}	1.19×10^{4932}

Google TPU implements bfloat16, NVIDIA implements half, quad is double double

DGX A100 Box



SYSTEM SPECIFICATIONS	
GPUs	8x NVIDIA A100 Tensor Core GPUs
GPU Memory	320 GB total
Performance	5 petaFLOPS AI 10 petaOPS INT8
NVIDIA NVSwitches	6
System Power Usage	6.5kW max
CPU	Dual AMD Rome 7742, 128 cores total, 2.25 GHz (base), 3.4 GHz (max boost)
System Memory	1TB
Networking	8x Single-Port Mellanox ConnectX-6 VPI 200Gb/s HDR InfiniBand 1x Dual-Port Mellanox ConnectX-6 VPI 10/25/50/100/200Gb/s Ethernet
Storage	OS: 2x 1.92TB M.2 NVME drives Internal Storage: 15TB (4x 3.84TB) U.2 NVME drives
Software	Ubuntu Linux OS
System Weight	271 lbs (123 kgs)
Packaged System Weight	315 lbs (143kgs)
System Dimensions	Height: 10.4 in (264.0 mm) Width: 19.0 in (482.3 mm) MAX Length: 35.3 in (897.1 mm) MAX
Operating Temperature Range	5°C to 30°C (41°F to 86°F)

Single Box = DP 9.7x8=77.6 TF (155.3 TF TC),
TF32 TC: 312x8=2.5PFLOPS; FP16 TC : 2.5 PF x2 – 5 PF AI

Cost : assume $100,000 x 6000 = 600 M, Frontier @ORNL = 600M
Aggregate performance DP 9.7 x 8 x 6000 ~= 465 PF, (931PF TC)
AI Performance : FP16 TC : 5 PF x 8 x 6000 = 240 ExaFLOPS !!!!

GPU : 32bit ML : Best performance / cost

DNN Computing – Why GPU

Lots of MM (BLAS3) – Performance

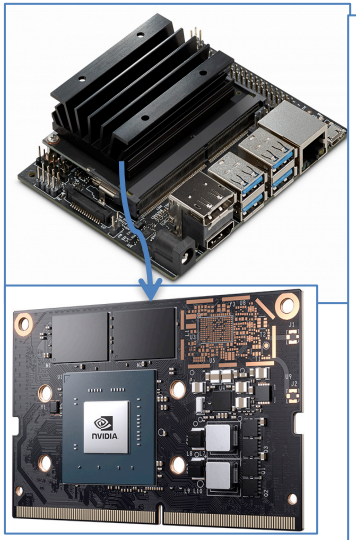
Data in 32 bits or 16 bits – Reduce Memory Access

Lots of Small Parallel Computing Unit – Time Scale

Power Usage - Low , Efficiency

Fast Memory Access Within GPU

GPU architectures



DEVELOPER KIT	
GPU	128-core Maxwell
CPU	Quad-core ARM A57 @ 1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	microSD (not included)
Video Encoder	4K @ 30 | 4x 1080p @ 30 | 9x 720p @ 30 (H.264/H.265)
Video Decoder	4K @ 60 | 2x 4K @ 30 | 8x 1080p @ 30 | 18x 720p @ 30| (H.264/H.265)
Camera	1x MIPI CSI-2 DPHY lanes
Connectivity	Gigabit Ethernet, M.2 Key E
Display	HDMI 2.0 and eDP 1.4
USB	4x USB 3.0, USB 2.0 Micro-B
Others	GPIO, I²C, I²S, SPI, UART
Mechanical	100 mm x 80 mm x 29 mm

Jetson Nano Developer Kit

Nsight Systems	2019.3
Nsight Graphics	2018.7
Nsight Compute	1.0
Jetson GPIO	1.0
Jetson OS	Ubuntu 18.04
CUDA	10.0.166
cuDNN	7.3.1.28
TensorRT	5.0.6.3

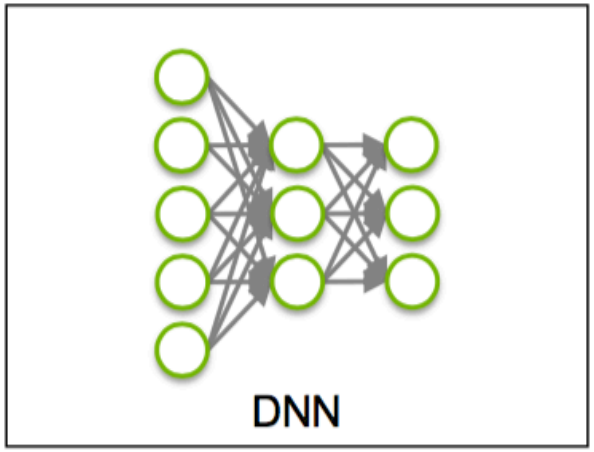
Other NVIDIA GPUs used in this workshop: GTX 1650 , K80, P100, V100, A100, ..

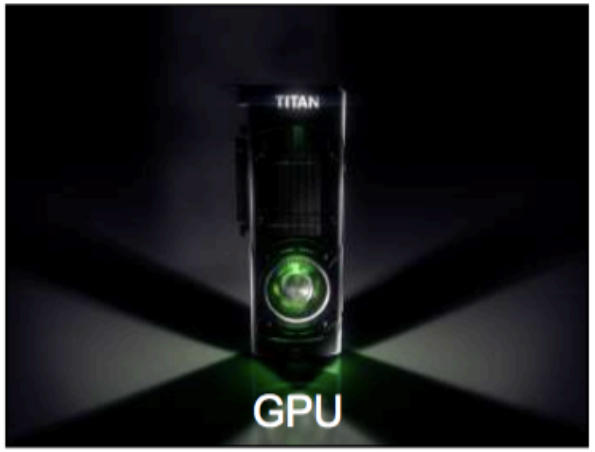
PRODUCT SPECIFICATIONS	
NVIDIA® CUDA Cores	896
Clock Speed	1485 MHz
Boost Speed	1725 MHz
Memory Speed (Gbps)	8
Memory Size	4GB GDDR5
Memory Interface	128-bit
Memory Bandwidth (Gbps)	128

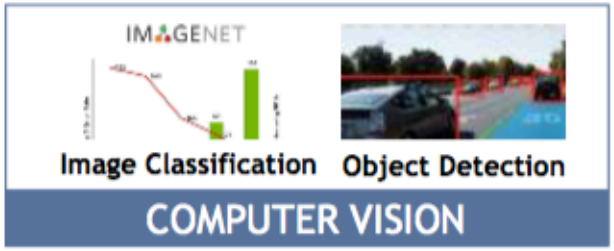
NVIDIA V100 on Summit

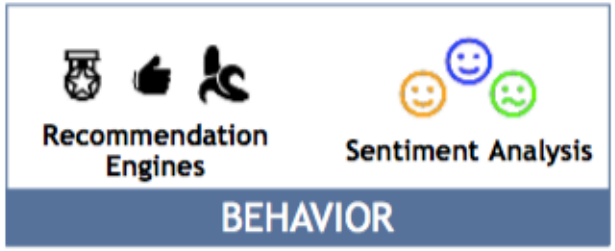
GPU Architecture	NVIDIA Volta	
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS
Memory Bandwidth	900GB/sec	

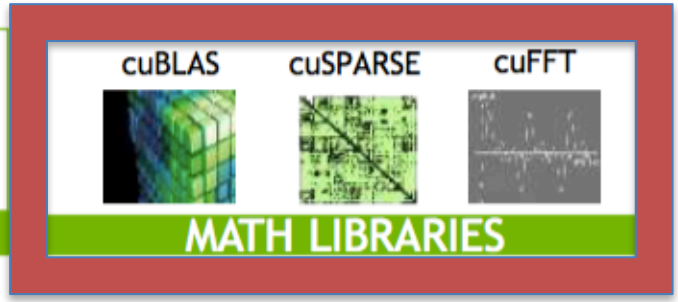
Machine Learning :LA (BLAS3) – GPU acceleration

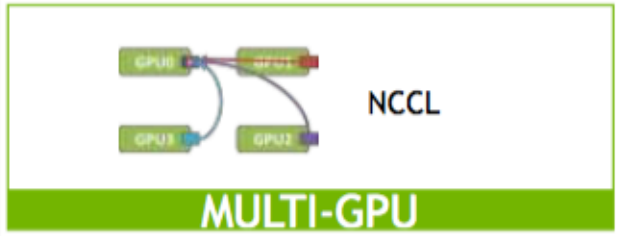












Source captured frin - Julie Bernauer – HPC Advisory Council Stanford Tutorial – 2017/02/07

Nvidia GPU Computing Applications

- ✓ A parallel computing platform that leverages NVIDIA GPUs
- ✓ Accessible through CUDA libraries, compiler directives, application programming interfaces, and extensions to several programming languages (**C/C++**, Fortran, and Python).

GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages

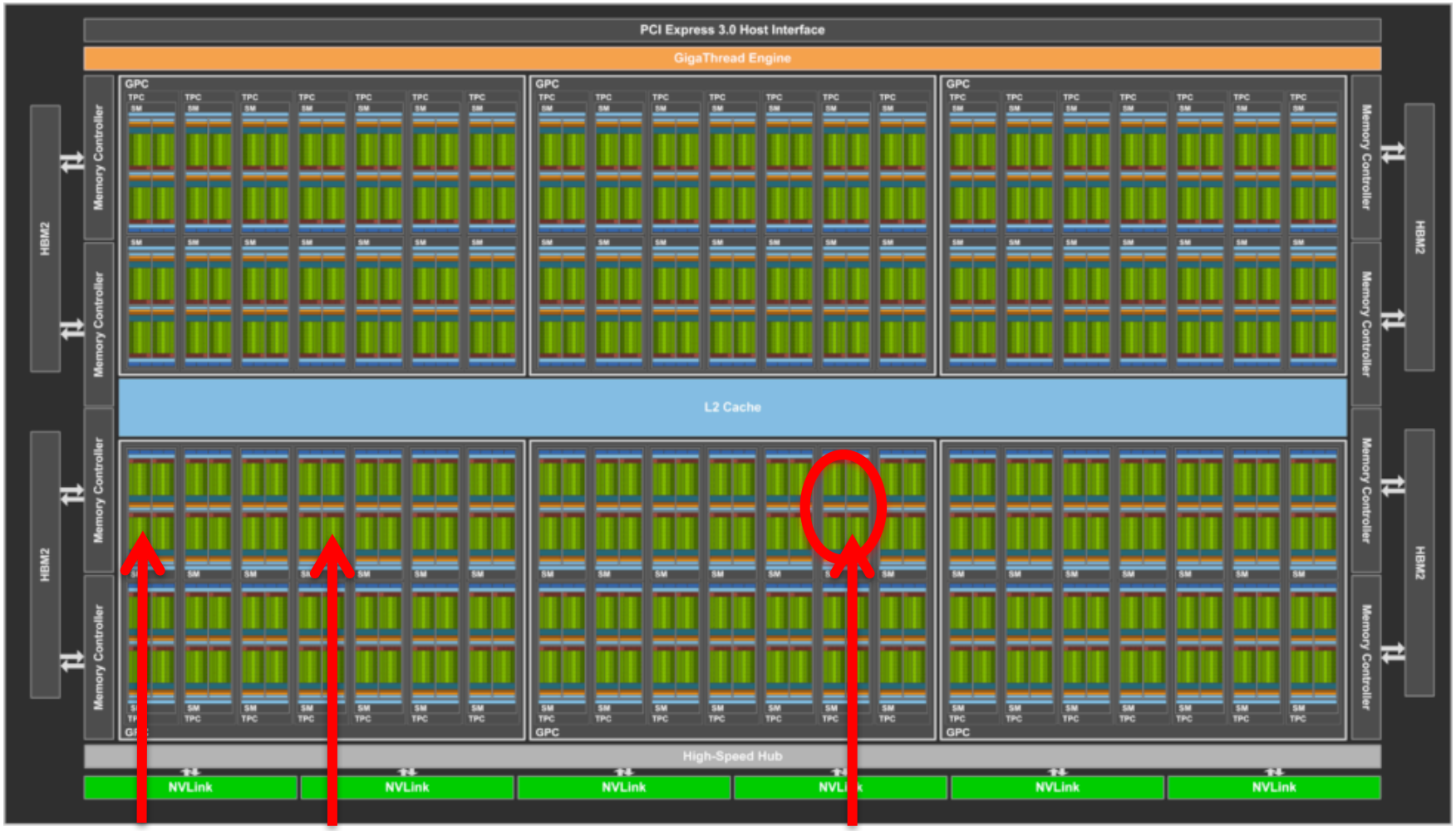
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
				

GPU architectures

nvidia-tesla-v100-gpu-details



Streaming Multiprocessors (SMs) **32 cores**

https://devblogs.nvidia.com/inside-volta/image3-3/

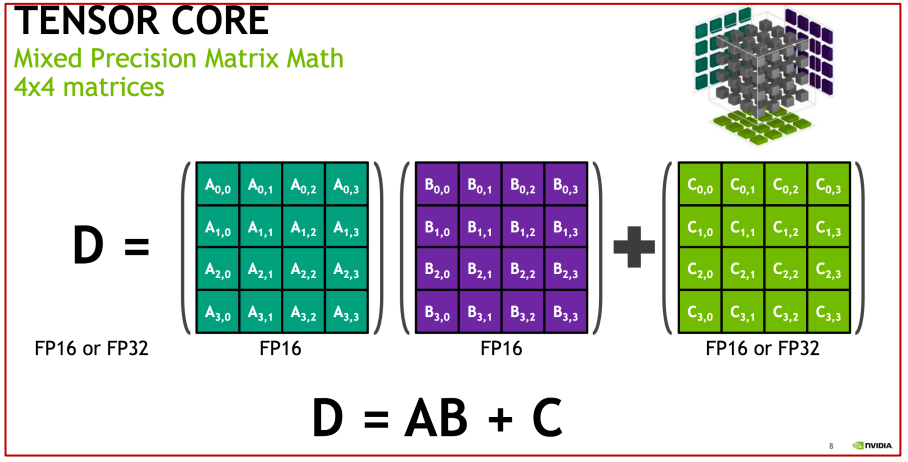
All About MM in SM V100, (A100, H100)

General Matrix-Matrix Multiplication (GEMM) operations are at the core of neural network training and inference, and are used to multiply large matrices of input data and weights in various layers. The GEMM operation computes the matrix product *D = A * B + C*, where *C* and *D* are *m*-by-*n* matrices, *A* is an *m*-by-*k* matrix, and *B* is a *k*-by-*n* matrix. The problem size of such GEMM operations running on Tensor Cores is defined by the matrix sizes, and typically denoted as *m*-by-*n*-by-*k*, (4x4x4) or (8x4x8).

FMA = Fused Multiple-add, compute 2 floating point

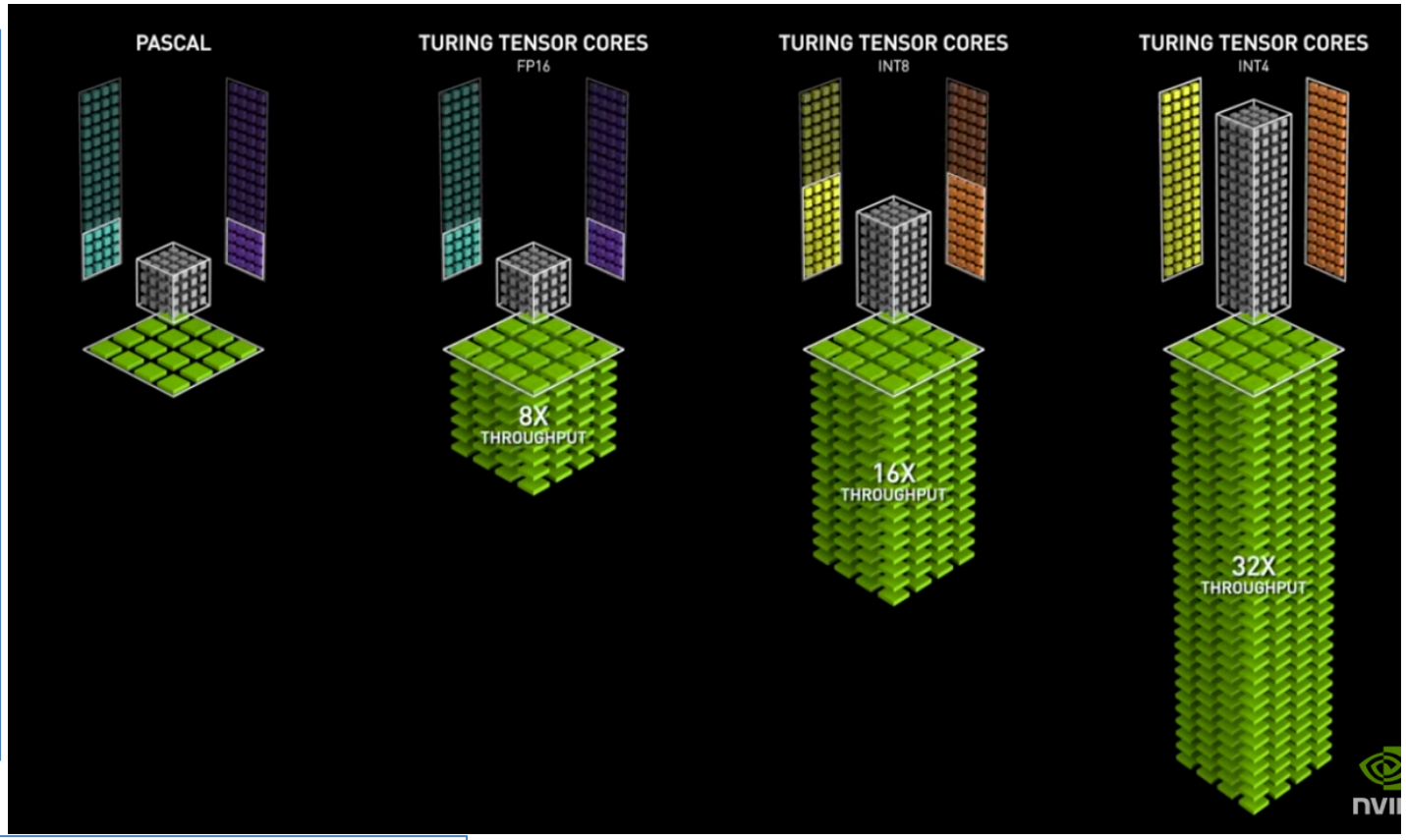
	V100	A100	A100 Sparsity[1]	A100 Speedup	A100 Speedup with Sparsity
A100 FP16 vs V100 FP16	31.4 TFLOPS	78 TFLOPS	NA	2.5x	NA
A100 FP16 TC vs V100 FP16 TC	125 TFLOPS	312 TFLOPS	624 TFLOPS	2.5x	5x
A100 BF16 TC vs V100 FP16 TC	125 TFLOPS	312 TFLOPS	624 TFLOPS	2.5x	5x
A100 FP32 vs V100 FP32	15.7 TFLOPS	19.5 TFLOPS	NA	1.25x	NA
A100 TF32 TC vs V100 FP32	15.7 TFLOPS	156 TFLOPS	312 TFLOPS	10x	20x
A100 FP64 vs V100 FP64	7.8 TFLOPS	9.7 TFLOPS	NA	1.25x	NA
A100 FP64 TC vs V100 FP64	7.8 TFLOPS	19.5 TFLOPS	NA	2.5x	NA
A100 INT8 TC vs V100 INT8	62 TOPS	624 TOPS	1248 TOPS	10x	20x
A100 INT4 TC	NA	1248 TOPS	2496 TOPS	NA	NA
A100 Binary TC	NA	4992 TOPS	NA	NA	NA

Tensor core is a new type of processing core that performs a type of specialized matrix math, suitable for deep learning and certain types of HPC. Tensor cores perform a fused multiply add, where two 4 x 4 FP16 matrices are multiplied and then the result added to a 4 x 4 FP16 or FP32 matrix. The result is a 4 x 4 FP16 or FP32 matrix; NVIDIA refers to tensor cores as performing mixed precision math, because the inputted matrices are in half precision but the product can be in full precision.



Tensor Core MM, V100, A100 Performance

- ✓ A100 : 1 SM has 4 TC,
- ✓ Each TC does 256 FP16 FMA unit
- ✓ Imply each TC can do a 8x4x8 (multiplying 8×4 matrix with 4×8 matrix) mixed-precision matrix multiplication per clock – (256)
- ✓ So 1 SM can do 4 x 256 = 1024 FP16 FMA or 2048 FP16 operations per clock cycle
- ✓ 108 x 2048 x 1.4 = 310 TFLOPS
- ✓ Same for FP32 → FP64



NVIDIA A100 Tensor Core GPU with its 108 SMs includes a total of 432 Tensor Cores delivers up to 312 TFLOPS of dense mixed-precision FP16/FP32 performance.

That equates to 2.5x the mixed-precision Tensor Core performance of the entire Tesla V100 GPU, and 20x V100's standard FP32 (FMA operations running on traditional FP32 CUDA cores) throughput.

FMA = Fused Multiple-add, compute 2 floating point

- ✓ V100 : 1 SM has 8 TC,
- ✓ Each TC does 64 FP16
- ✓ Imply each TC can do a 4x4x4 matrix – 64 FMA
- ✓ So 1 SM can do 8 x 64 = 512 FP16 FMA or 1024 FP16 operations per clock cycle
- ✓ 80SM x 1024 FP16 / C x 1.53 GHz = 125 TFLOPS

GPU Performance architectures

GPU Features	NVIDIA Tesla P100	NVIDIA Tesla V100	NVIDIA A100
GPU Codename	GP100	GV100	GA100
GPU Architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere
GPU Board Form Factor	SXM	SXM2	SXM4
SMs	56	80	108
TPCs	28	40	54
FP32 Cores / SM	64	64	64
FP32 Cores / GPU	3584	5120	6912
FP64 Cores / SM (excl. Tensor)	32	32	32
FP64 Cores / GPU (excl. Tensor)	1792	2560	3456
INT32 Cores / SM	NA	64	64
INT32 Cores / GPU	NA	5120	6912
Tensor Cores / SM	NA	8	4[2]
Tensor Cores / GPU	NA	640	432
GPU Boost Clock	1480 MHz	1530 MHz	1410 MHz
Peak FP16 Tensor TFLOPS with FP16 Accumulate[1]	NA	125	312/624[3]
Peak FP16 Tensor TFLOPS with FP32 Accumulate[1]	NA	125	312/624[3]
Peak BF16 Tensor TFLOPS with FP32 Accumulate[1]	NA	NA	312/624[3]
Peak TF32 Tensor TFLOPS[1]	NA	NA	156/312[3]
Peak FP64 Tensor TFLOPS[1]	NA	NA	19.5
Peak INT8 Tensor TOPS[1]	NA	NA	624/1248[3]
Peak INT4 Tensor TOPS[1]	NA	NA	1248/2496[3]
Peak FP16 TFLOPS[1] (non-Tensor)	21.2	31.4	78
Peak BF16 TFLOPS[1] (non-Tensor)	NA	NA	39
Peak FP32 TFLOPS[1] (non-Tensor)	10.6	15.7	19.5
Peak FP64 TFLOPS[1] (non-Tensor)	5.3	7.8	9.7
Peak INT32 TOPS[1,4]	NA	15.7	19.5
Texture Units	224	320	432
Memory Interface	4096-bit HBM2	4096-bit HBM2	5120-bit HBM2
Memory Size	16 GB	32 GB / 16 GB	40 GB
Memory Data Rate	703 MHz DDR	877.5 MHz DDR	1215 MHz DDR

Peak FP64[1]	9.7 TFLOPS
Peak FP64 Tensor Core[1]	19.5 TFLOPS
Peak FP32[1]	19.5 TFLOPS
Peak FP16[1]	78 TFLOPS
Peak BF16[1]	39 TFLOPS
Peak TF32 Tensor Core[1]	156 TFLOPS | 312 TFLOPS[2]
Peak FP16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak BF16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak INT8 Tensor Core[1]	624 TOPS | 1,248 TOPS[2]
Peak INT4 Tensor Core[1]	1,248 TOPS | 2,496 TOPS[2]

Memory Bandwidth	720 GB/sec	900 GB/sec	1555 GB/sec
L2 Cache Size	4096 KB	6144 KB	40960 KB
Shared Memory Size / SM	64 KB	Configurable up to 96 KB	Configurable up to 164 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	14336 KB	20480 KB	27648 KB
TDP	300 Watts	300 Watts	400 Watts
Transistors	15.3 billion	21.1 billion	54.2 billion
GPU Die Size	610 mm²	815 mm²	826 mm2
TSMC Manufacturing Process	16 nm FinFET+	12 nm FFN	7 nm N7

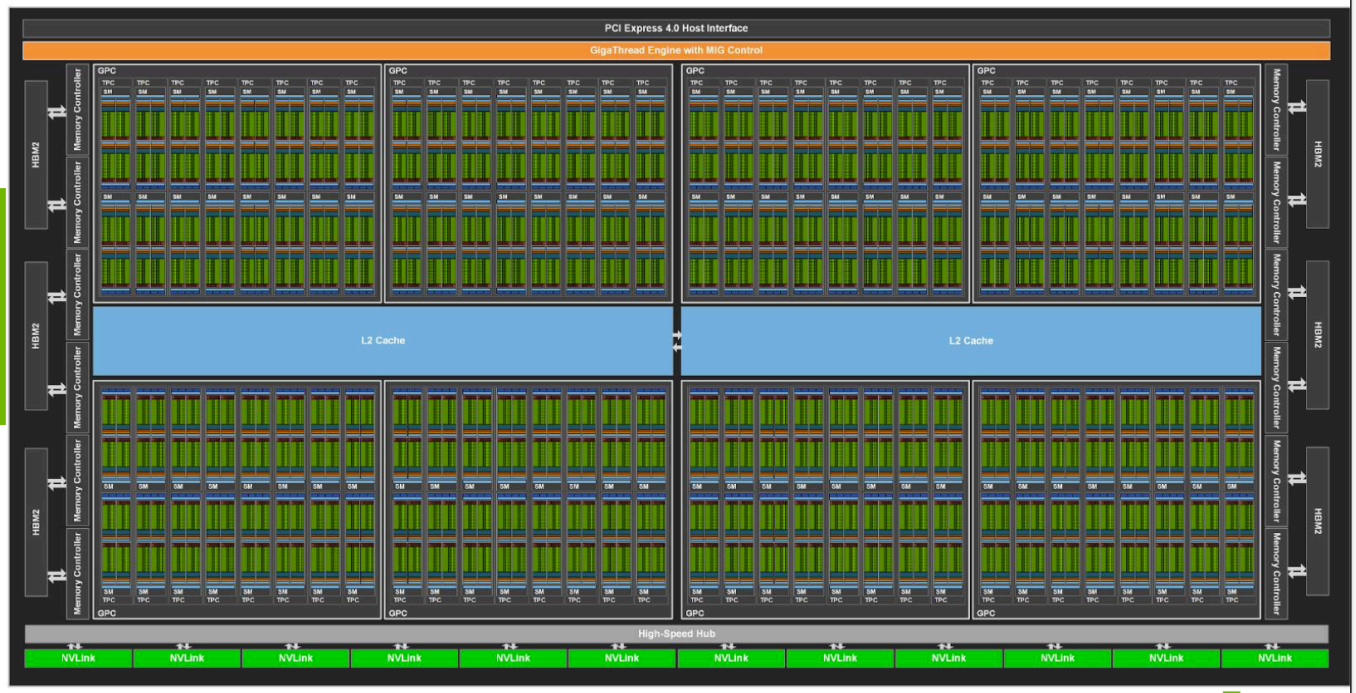
1. **Peak rates are based on GPU Boost Clock.**
2. **Four Tensor Cores in an A100 SM have 2x the raw FMA computational power of eight Tensor Cores in a GV100 SM.**
3. **Effective TOPS / TFLOPS using the new Sparsity Feature**
4. **TOPS = IMAD-based integer math**

Ampere A100

54B transistors
826 mm²

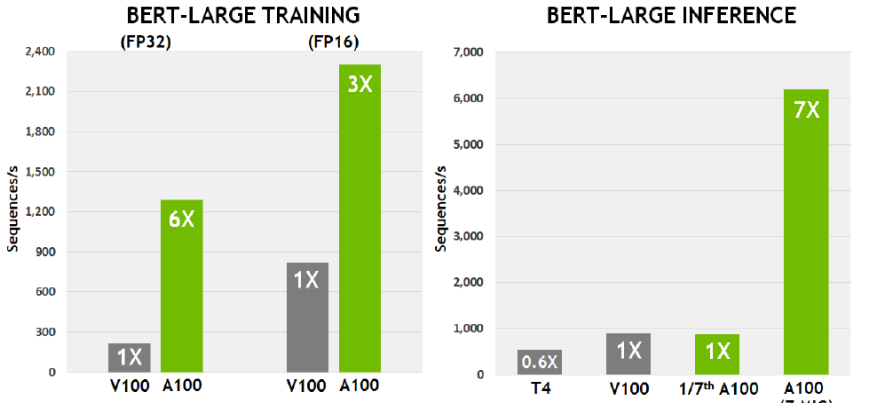
108 SM
6912 CUDA Cores
432 Tensor Cores

40 GB HBM2
1555 GB/s HBM2
600 GB/s NVLink

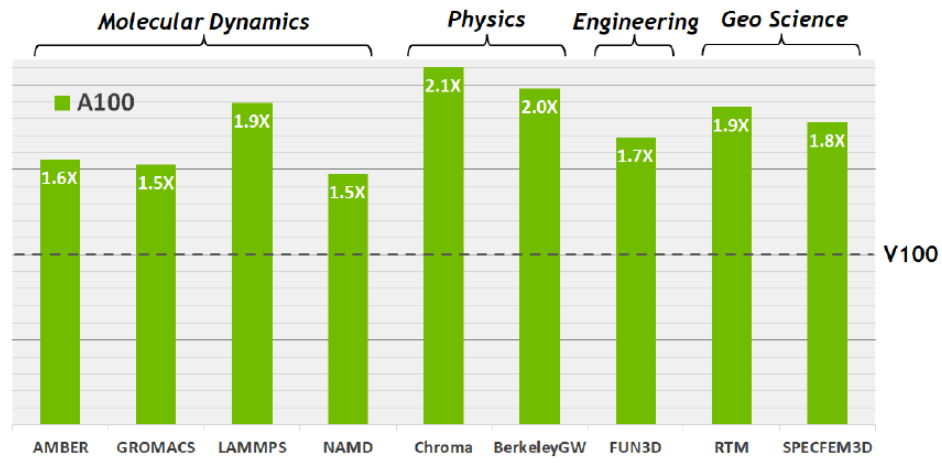


 NVIDIA.

UNIFIED AI ACCELERATION

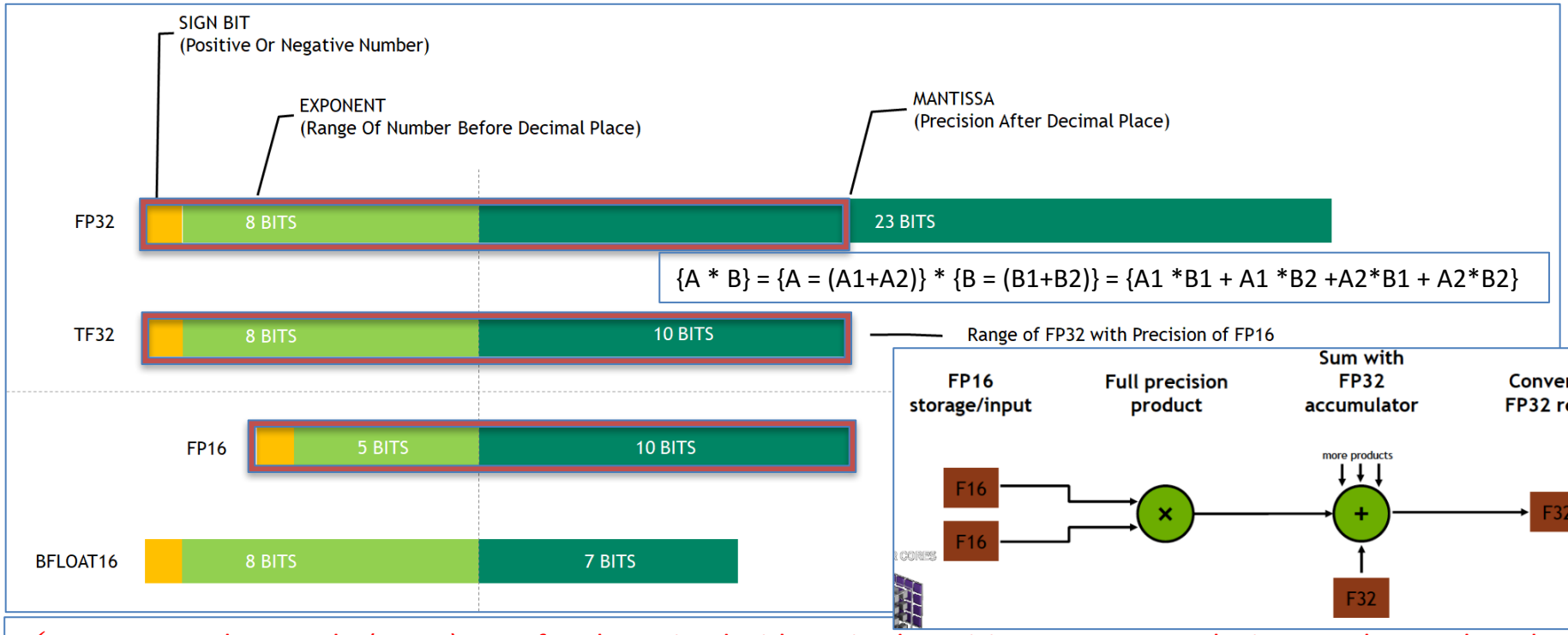


ACCELERATING HPC



https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

Numerical precisions supported by NVIDIA A100 for Deep Learning



- ✓ Deep neural networks (DNNs) can often be trained with a mixed precision strategy, employing mostly FP16 but also FP32 precision when necessary. This strategy results in a significant reduction in computation, memory, and memory bandwidth requirements while most often converging to the similar final accuracy.
- ✓ NVIDIA Tensor Cores are specialized arithmetic units on NVIDIA Volta and newer generation GPUs. They can carry out a complete matrix multiplication and accumulation operation (MMA) in a single clock cycle. On Volta and Turing, the inputs are two matrices of size 4×4 in FP16 format, while the accumulator is in FP32.
- ✓ The third-generation Tensor Cores on Ampere support a novel math mode: TF32 is a hybrid format defined to handle the work of FP32 with greater efficiency. Specifically, TF32 uses the same 10-bit mantissa as FP16 to ensure accuracy while sporting the same range as FP32, thanks to using an 8-bit exponent.
- ✓ A wider representable range matching FP32 eliminates the need of a loss-scaling operation when using TF32, thus simplifying the mixed precision training workflow.

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7959640/pdf/peerj-cs-07-330.pdf

https://developer.nvidia.com/blog/accelerating-tensorflow-on-a100-gpus/

TF32 Single Precision - Tensor Core mode

TF32 is a Tensor Core mode, not a type

- ✓ Only convolutions and matrix multiplies convert inputs to TF32, all other operations remain completely FP32
- ✓ All storage in memory remains FP32
- ✓ Consequently, it's only exposed as a Tensor Core operation mode, contrast with fp16/bfloat16 types that provide: storage, various math operators, etc

Operation:

- ✓ Read FP32 inputs from memory, round inputs to TF32 prior to Tensor Core operation
- ✓ Multiply inputs without loss of precision, accumulate products in FP32, write FP32 product to memory

SAMPLING OF NETWORKS

Classification Tasks

Architecture	Network	Top-1 Accuracy	
		FP32	TF32
ResNet	RN18	70.43	70.58
	RN32	74.03	74.08
	RN50	76.78	76.73
	RN101	77.57	77.57
ResNext	RNX50	77.51	77.62
	RNX101	79.10	79.30
WideResNet	WRN50	77.99	78.11
	WRN101	78.61	78.62
DenseNet	DN121	75.57	75.57
	DN169	76.75	76.69
VGG	V11-BN	68.47	68.44
	V16-BN	71.54	71.51
	V19-BN	72.54	72.68
	V19	71.75	71.60
GoogleNet	InceptionV3	77.20	77.34
	Xception	79.09	79.31
Dilated RN	DRN A 50	78.24	78.16
ShuffleNet	V2-X1	68.62	68.87
	V2-X2	73.02	72.88
MNASNet	V1.0	71.62	71.49
SqueezeNet	V1_1	60.90	60.85
MobileNet	MN-V2	71.64	71.76
Stacked UNet	SUN64	69.53	69.62
EfficientNet	B0	76.79	76.72

Dataset is ISLVRC 2012

Detection & Segmentation Tasks

Architecture	Network	Metric	Model Accuracy	
			FP32	TF32
Faster RCNN	RN50 FPN 1X	mAP	37.81	37.95
	RN101 FPN 3X	mAP	40.04	40.19
	RN50 FPN 3X	mAP	42.05	42.14
Mask RCNN	TorchVision	mAP	37.89	37.89
		mIOU	34.65	34.69
	RN50 FPN 1X	mAP	38.45	38.63
		mIOU	35.16	35.25
	RN50 FPN 3X	mAP	41.04	40.93
		mIOU	37.15	37.23
	RN101 FPN 3X	mAP	42.99	43.08
		mIOU	38.72	38.73
Retina Net	RN50 FPN 1X	mAP	36.46	36.49
	RN50 FPN 3X	mAP	38.04	38.19
	RN101 FPN 3X	mAP	39.75	39.82
RPN	RN50 FPN 1X	mAP	58.02	58.11
Single-Shot Detector (SSD)	RN18	mAP	19.13	19.18
	RN50	mAP	24.91	24.85

Dataset is MS COCO 2017

Language Tasks

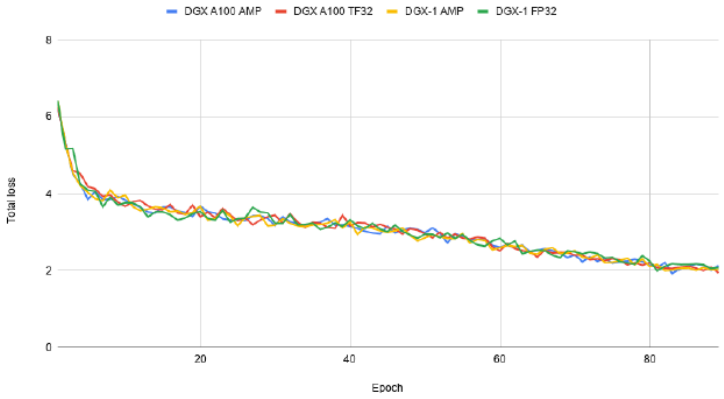
Architecture	Network	Dataset	Metric	Model Accuracy	
				FP32	TF32
Transformer	Vaswani Base	WMT	BLEU	27.18	27.10
	Vaswani Large	WMT	BLEU	28.63	28.62
	Levenshtein	WMT	Loss	6.16	6.16
Convolutional	Light Conv Base	WMT	BLEU	28.55	28.74
	Light Conv Large	WMT	BLEU	30.10	30.20
	Dynamic Conv Base	WMT	BLEU	28.34	28.42
	Dynamic Conv Large	WMT	BLEU	30.10	30.31
	FairSeq Conv	WMT	BLEU	24.83	24.86
Recurrent	GNMT	WMT	BLEU	24.53	24.80
Convolutional	Fairseq Dauphin	WikiText	PPL	35.89	35.80
Transformer	XL Standard	WikiText	PPL	22.89	22.80
BERT	Base Pre-train	Wikipedia	LM Loss	1.34	1.34
	Base Downstream	SQUAD v1	F1	87.95	87.66
		SQUAD v2	F1	76.68	75.67

No hyperparameter changes
Differences in accuracy are within typical bounds of run-to-run variation (different random seeds, etc.)

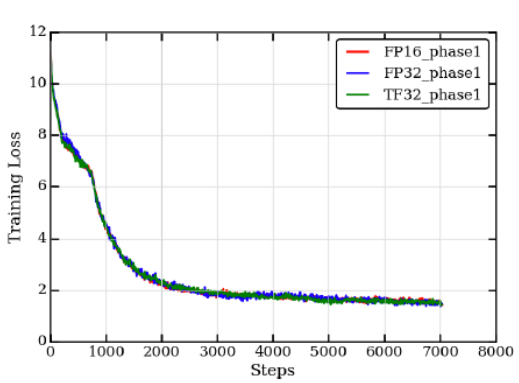
13 NVIDIA

Training Neural Networks with Tensor Cores - Dusan Stosic, NVIDIA
https://www.youtube.com/watch?v=jF4-_ZK_tyc

DL Performance of TF32 Single Precision mode

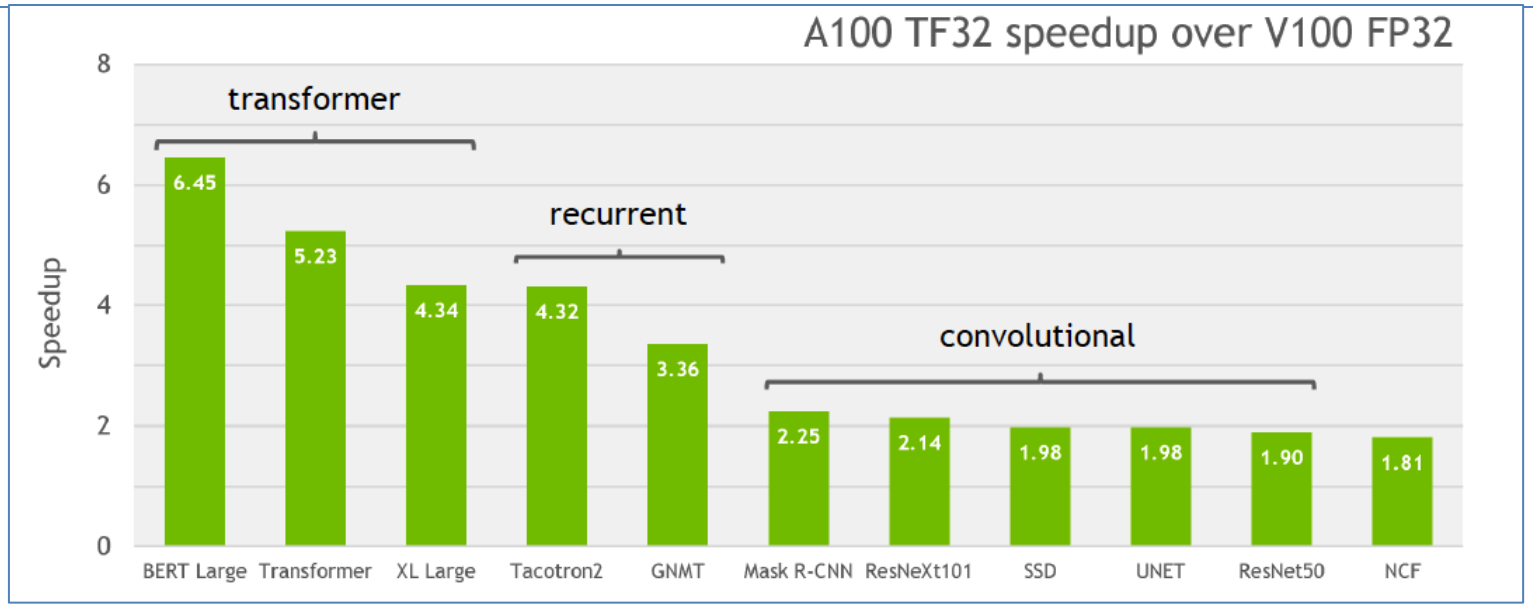


Model can be found at:
https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Classification/ConvNets/resnet50v1.5
Data collected on NVIDIA A100
Results can be reproduced with Tensorflow 1.15 in NGC container tensorflow:20.06-tf1-py3



Model can be found at:
https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/BERT
Data collected on NVIDIA A100
Performance can be reproduced with PyTorch 1.6 in NGC pytorch:20.06-py3 container

Results are easily reproducible using NGC containers and Deep Learning Examples

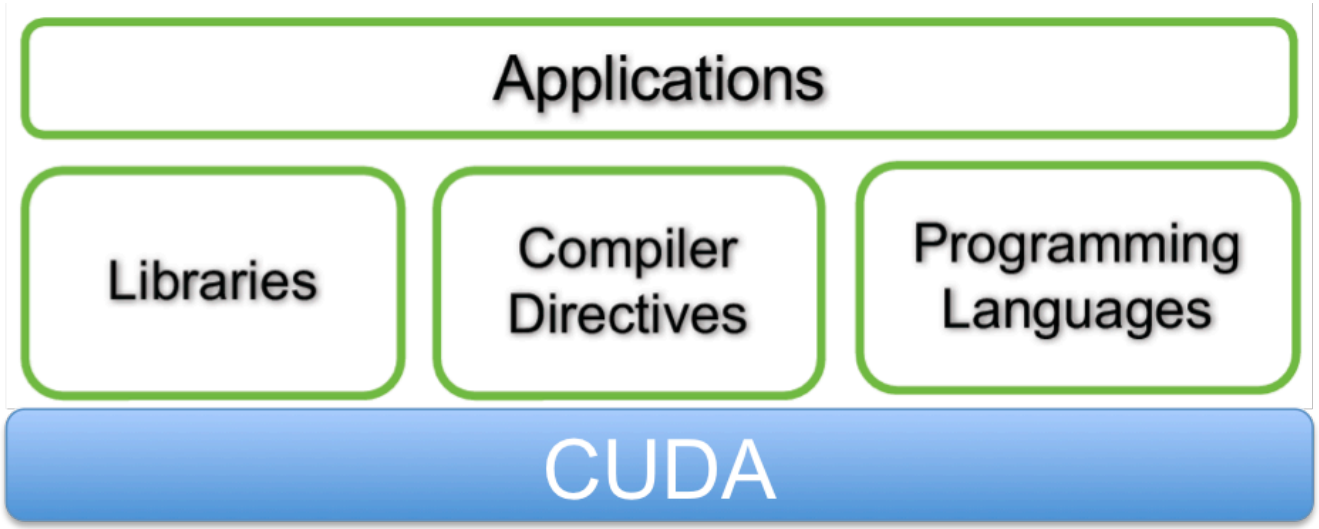


Training Neural Networks with Tensor Cores - Dusan Stosic, NVIDIA

https://www.youtube.com/watch?v=jF4-_ZK_tyc

Introduction to CUDA libraries

CUDA Libraries to accelerate the application development



Three Ways to Accelerate Applications by CUDA

- ✓ Ease to use: Using libraries enables GPU acceleration without in depth knowledge of GPU programming
- ✓ “Drop-in”: Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- ✓ Quality: Libraries offer high-quality implementations of functions encountered in a broad range of applications

From Now to Beyond

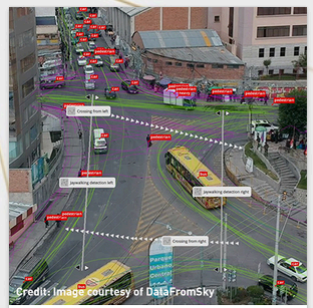
From predict to build..
From modeling to generating
From control to autonomous
From composition to integration
From predict to build…

From Now to Beyond

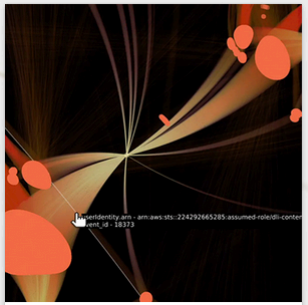
Keynote

Accelerated Computing and Dev Tools

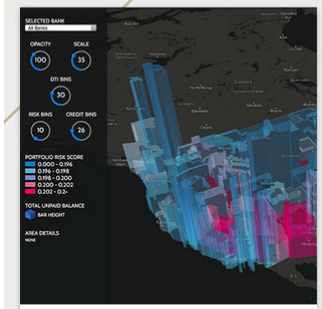
Autonomous Vehicles



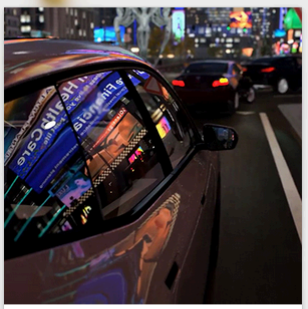
Computer Vision/ Video Analytics



Cybersecurity

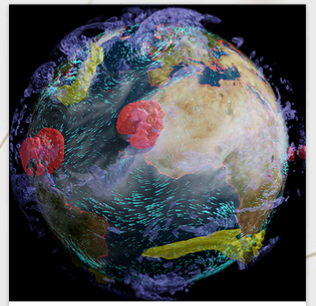


Data Science

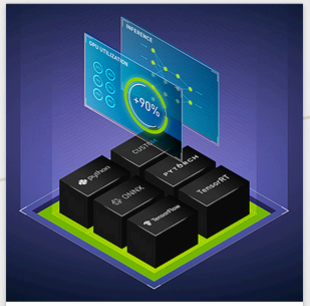


Game Development

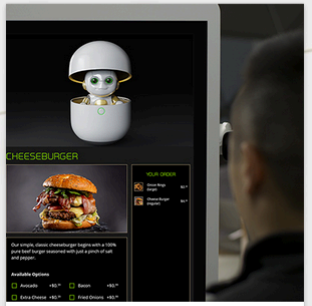
Graphics, Design Collaboration, and Digital Twins



HPC



Inference



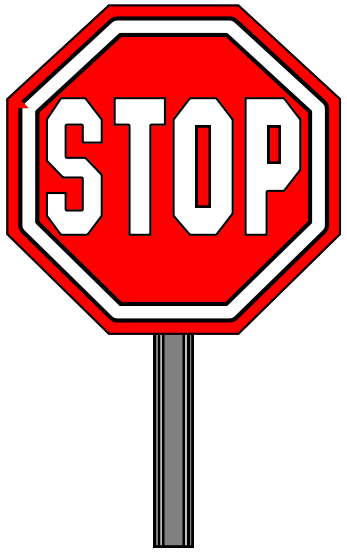
Recommender System

Robotics

Speech AI / NLP

Keynote Movie
www.nvidia.com/gtc/keynote

The End



- The End!