AI & ML | C++

Ashot Vardanian

Intro

Who am I?

Ashot Vardanian, 23

Independent Software Developer
AI Researcher

Astrophysics Department,
St. Petersburg State Polytechnic University

Basic, Pascal, JS, C, Objective-C, Fortran, C++

linkedin.com/in/ashvardanian
fb.com/ashvardanian

Plan

1. Theory behind Neural Networks
2. Demo: building `neural::mlp` from scratch
3. State-of-the-art
    1. ANN models
    2. AI Hardware
    3. ML Libraries
4. Demo: `std::pow("AI", 3)`
5. Future of C++

Main theme of the talk:

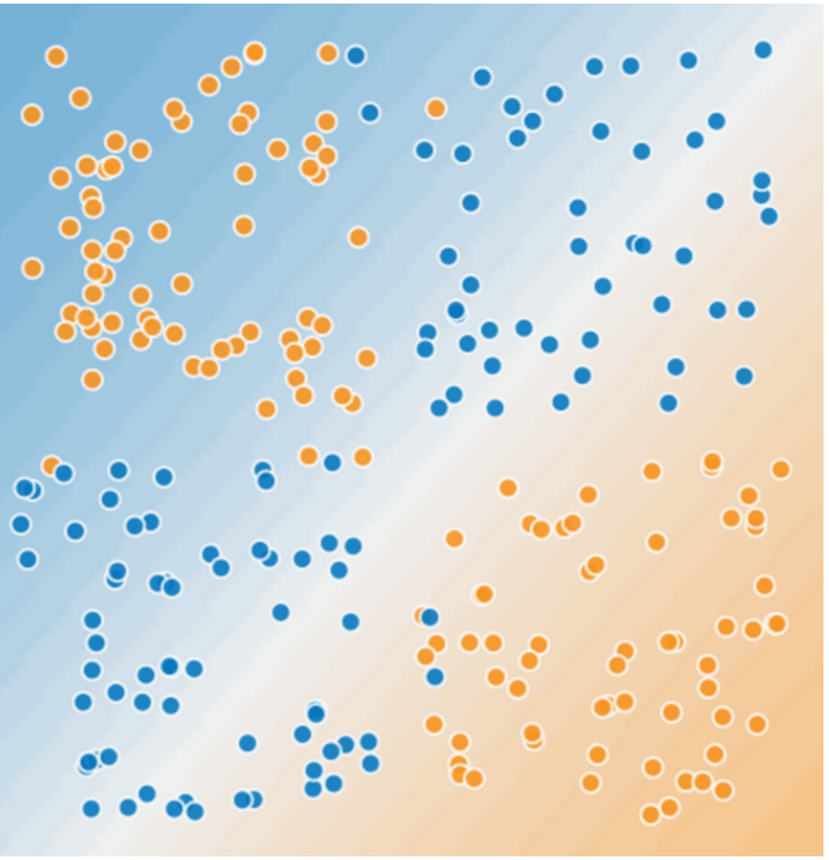
Neural Networks are easy to research and describe, but

Hard to implement and optimise for hardware!

Related Material

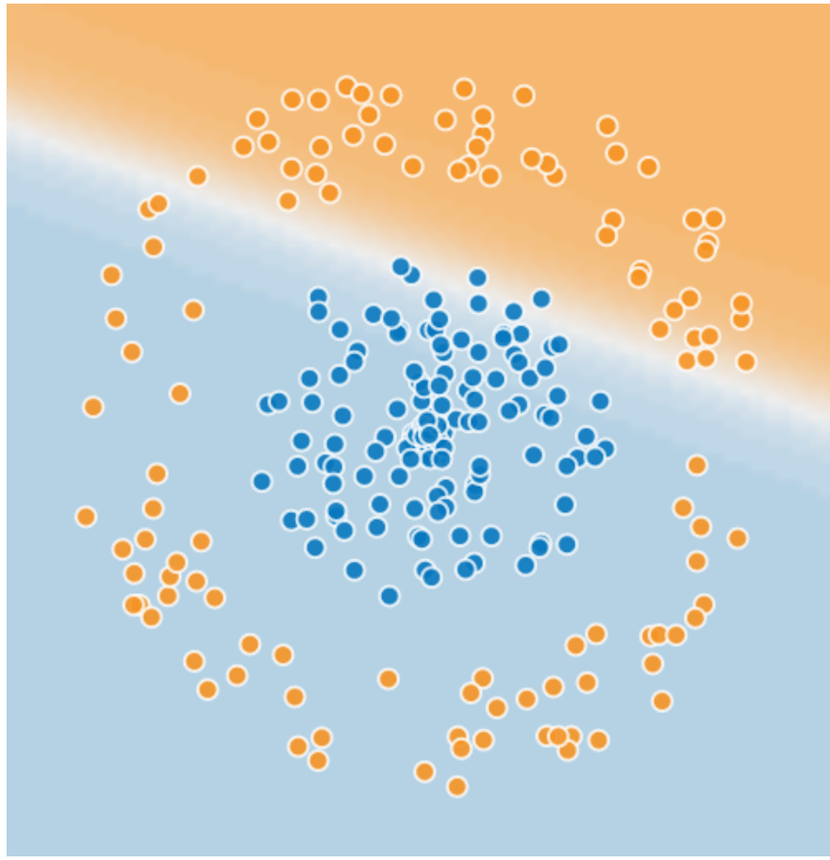
Theory

How can we split the points?



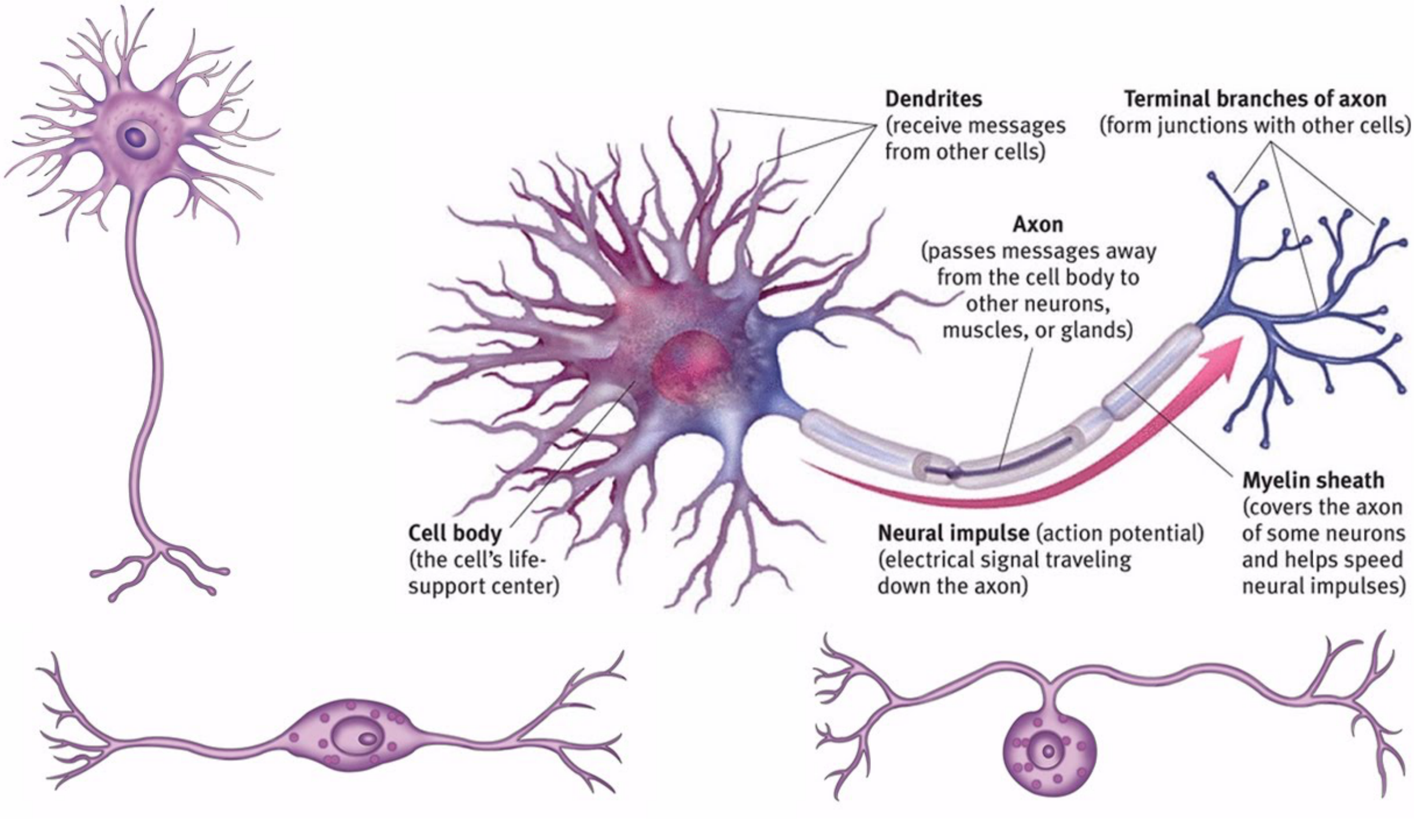
KD-Trees

Linear Models

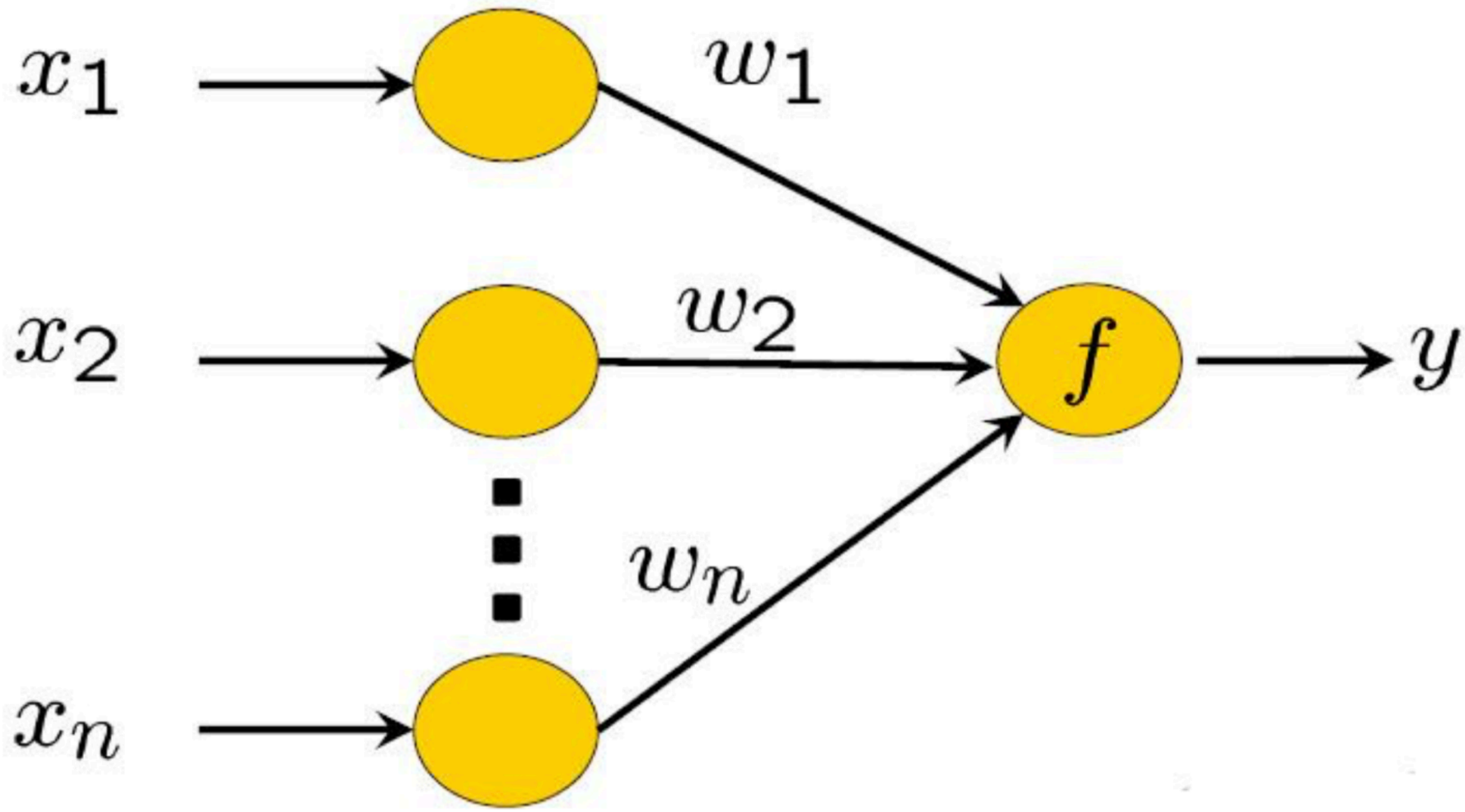


Non-Linear

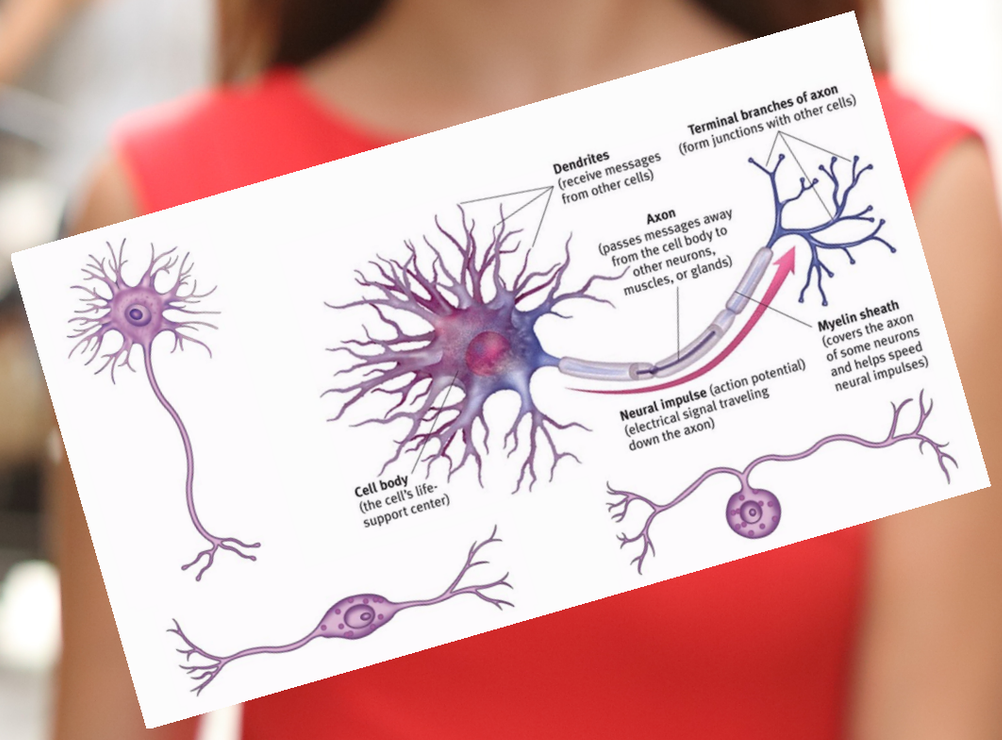
Inspiration

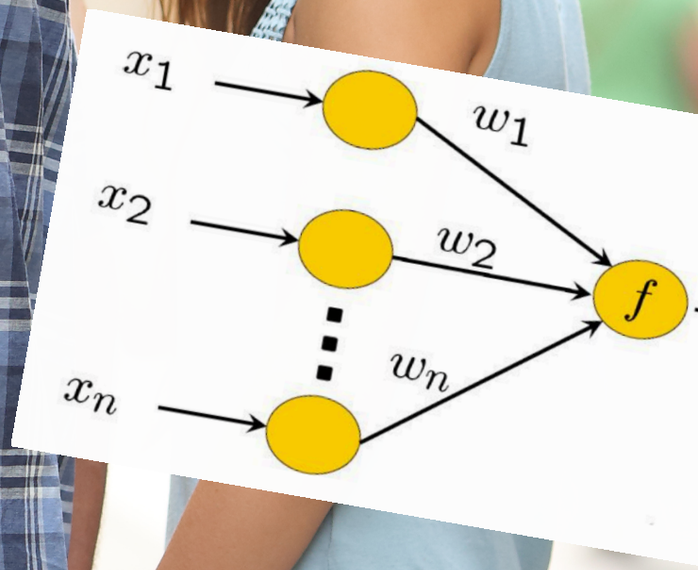


Incarnation



Expectation vs Reality



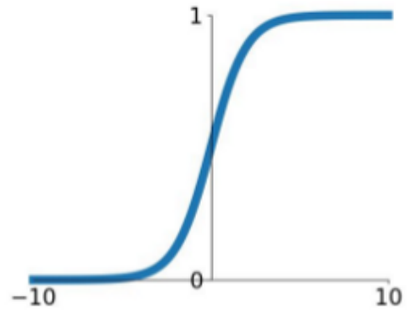


Activation Functions

The most common differentiable “non-linearities” in Neural Networks

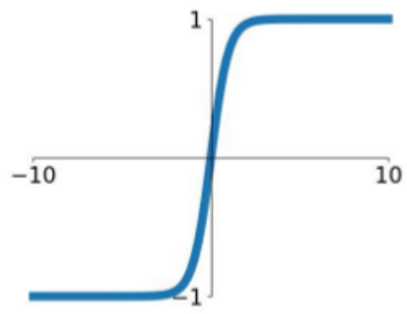
Sigmoid

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



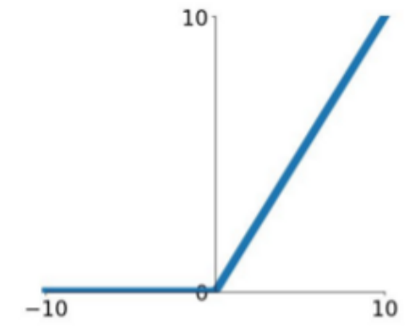
tanh

$$\tanh(x)$$



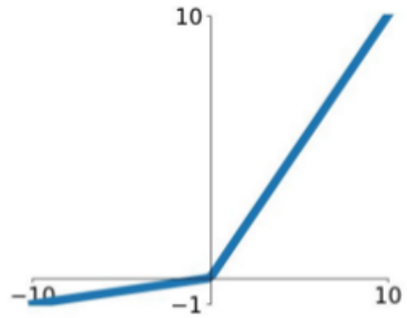
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

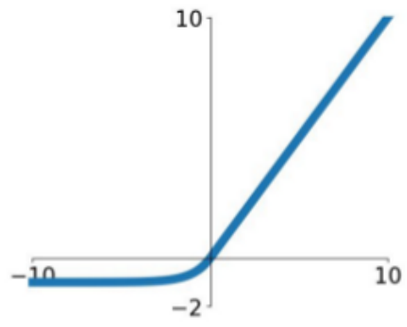


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Archean period
Single cell life

Progression

1943 - 1963

- First Neuron Model (1943), Pitts & McCulloch
- Perceptron (1958), Rosenblatt
- Perceptron convergence theorem (1962)
- **Back-propagation (1963), Bryson**

Degression

1963 - 1982

- Perceptron can’t even learn the XOR function
- **Nobody does back-propagation, so no MLP training**

Protozoic period
Eucaryotes

Progression

1983 - 1993

- Hopfield Nets (1982)
- Boltzmann Machines (1985)
- Good understanding of “non-linearities”, but still poor understanding of network topology, layers complexity

Degression

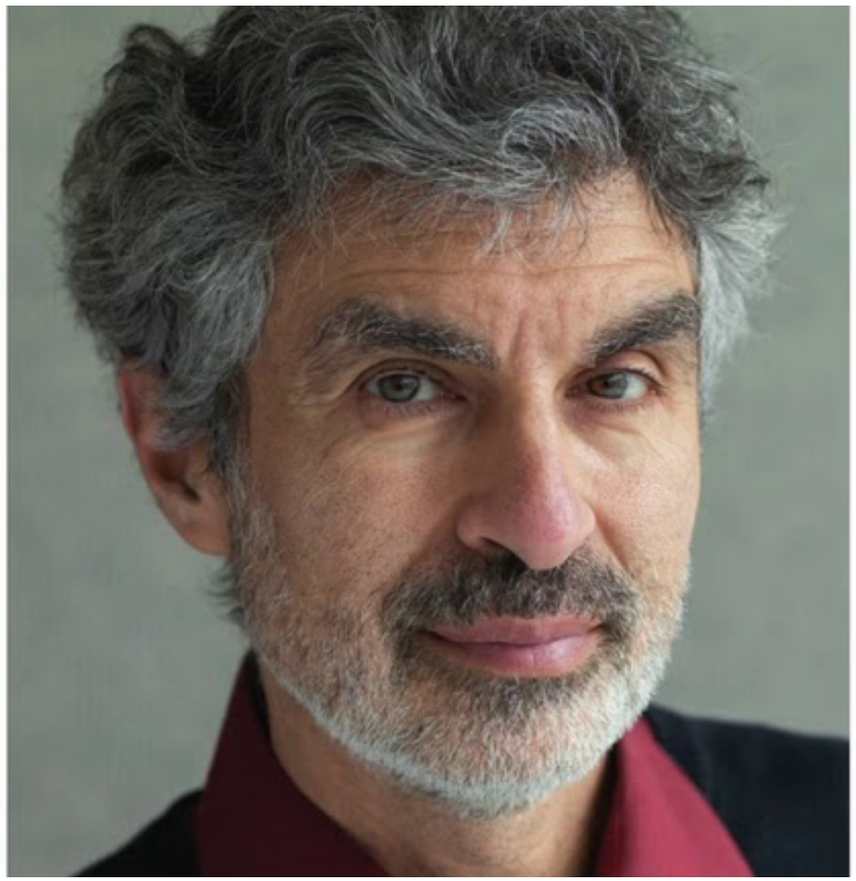
1993-2006

- SVM (1995) are killing ANNs
- **Training deeper networks consistently yields poor results**

- Deep convolutional neural networks (1998), Yann LeCun

Phanerozoic period
Complex Multicellular Life

- **Hardware** improvements empower larger models
- Deep Belief Networks (2006), Hinton
- Deep Autoencoder based networks (2007), Bengio

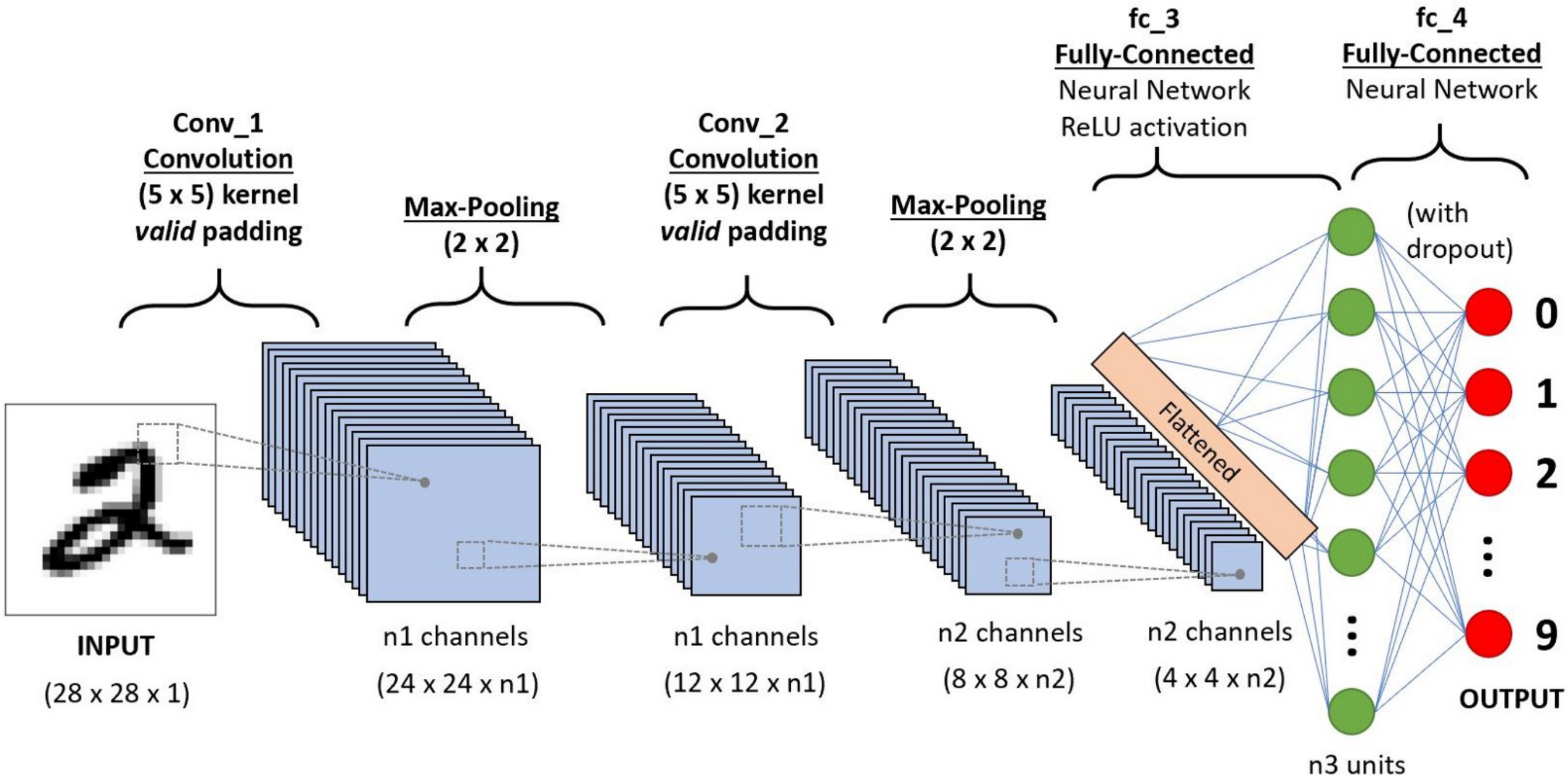

Simple Demo:

Building MLPs with STL

State of the Art:
Neural Networks

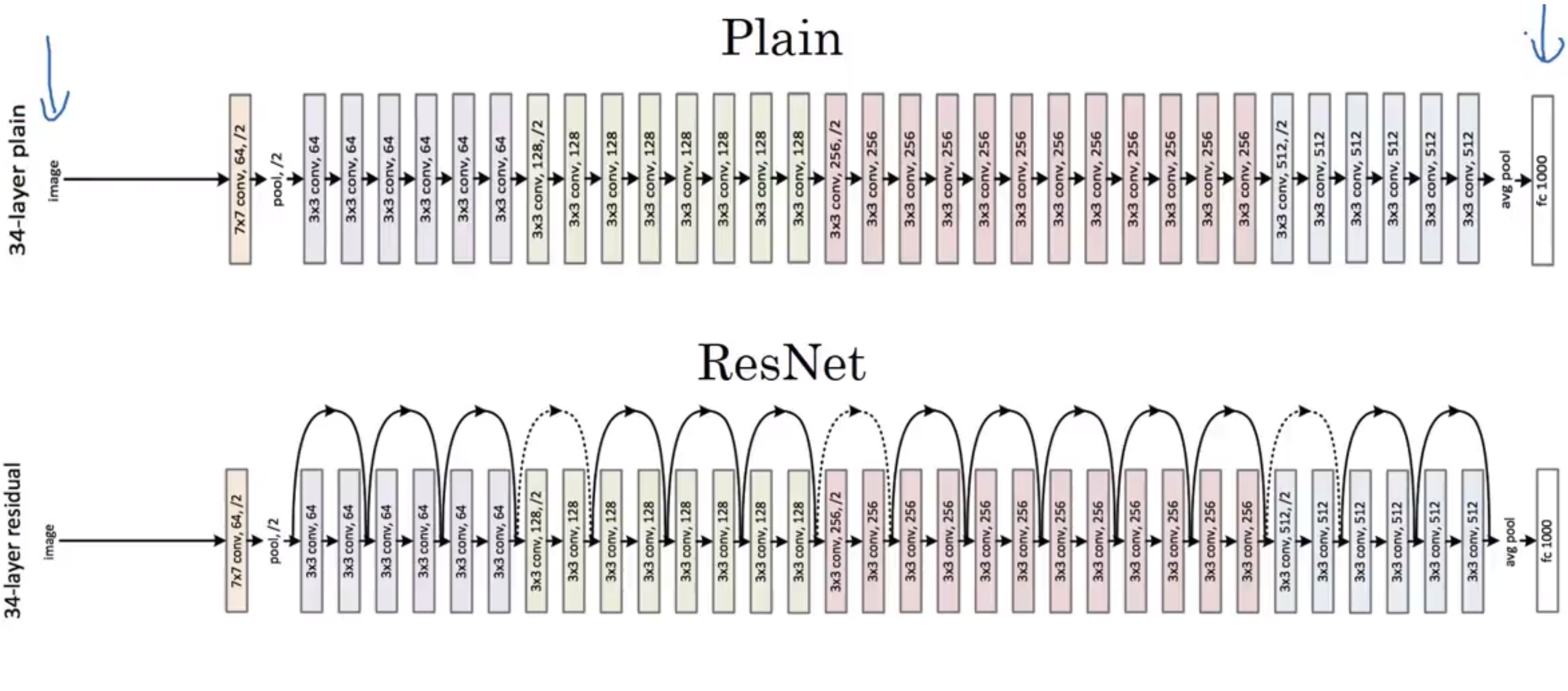
CNNs

A sequential model with convolutional layers



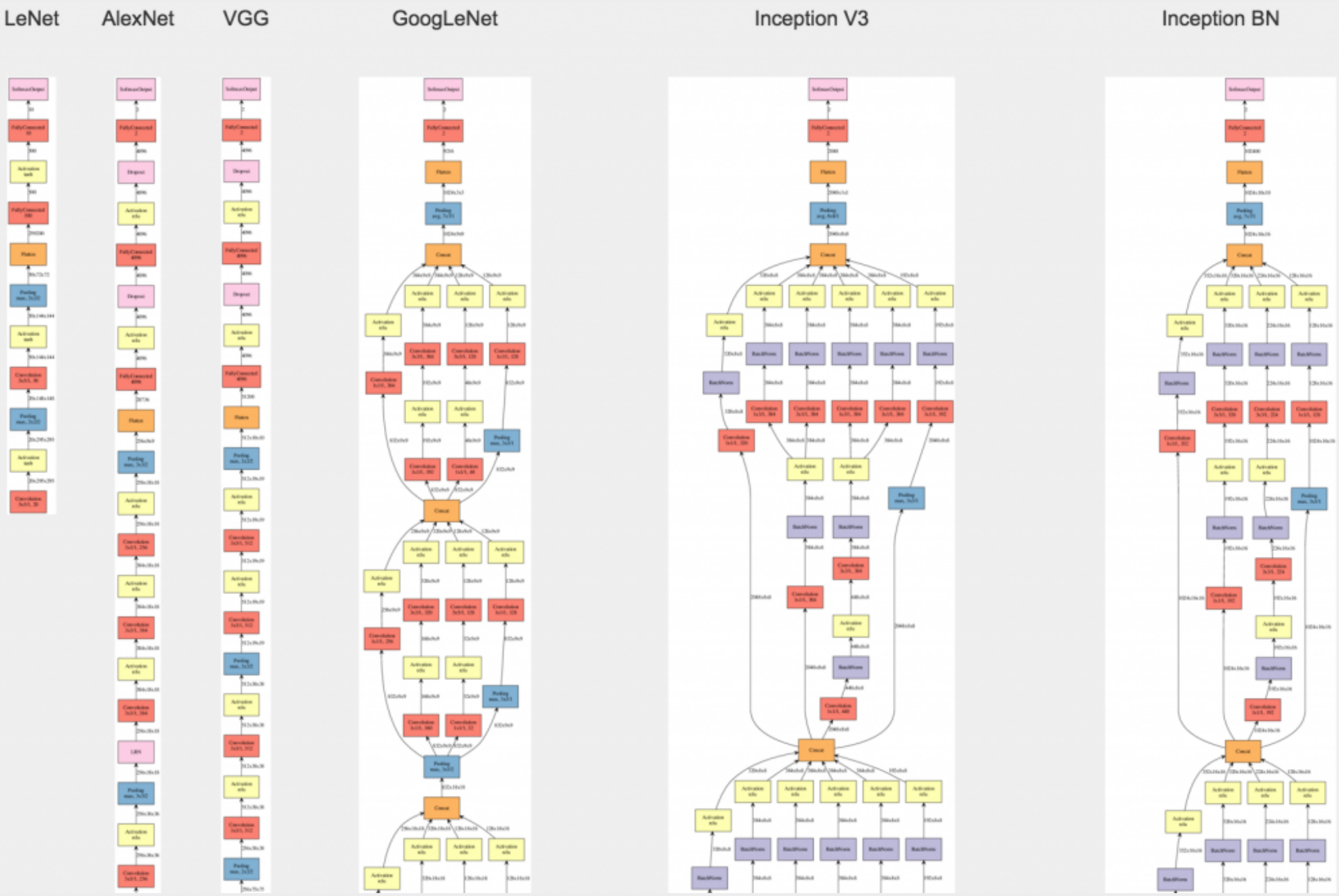
ResNet

A less sequential model with convolutional layers and skip-connections

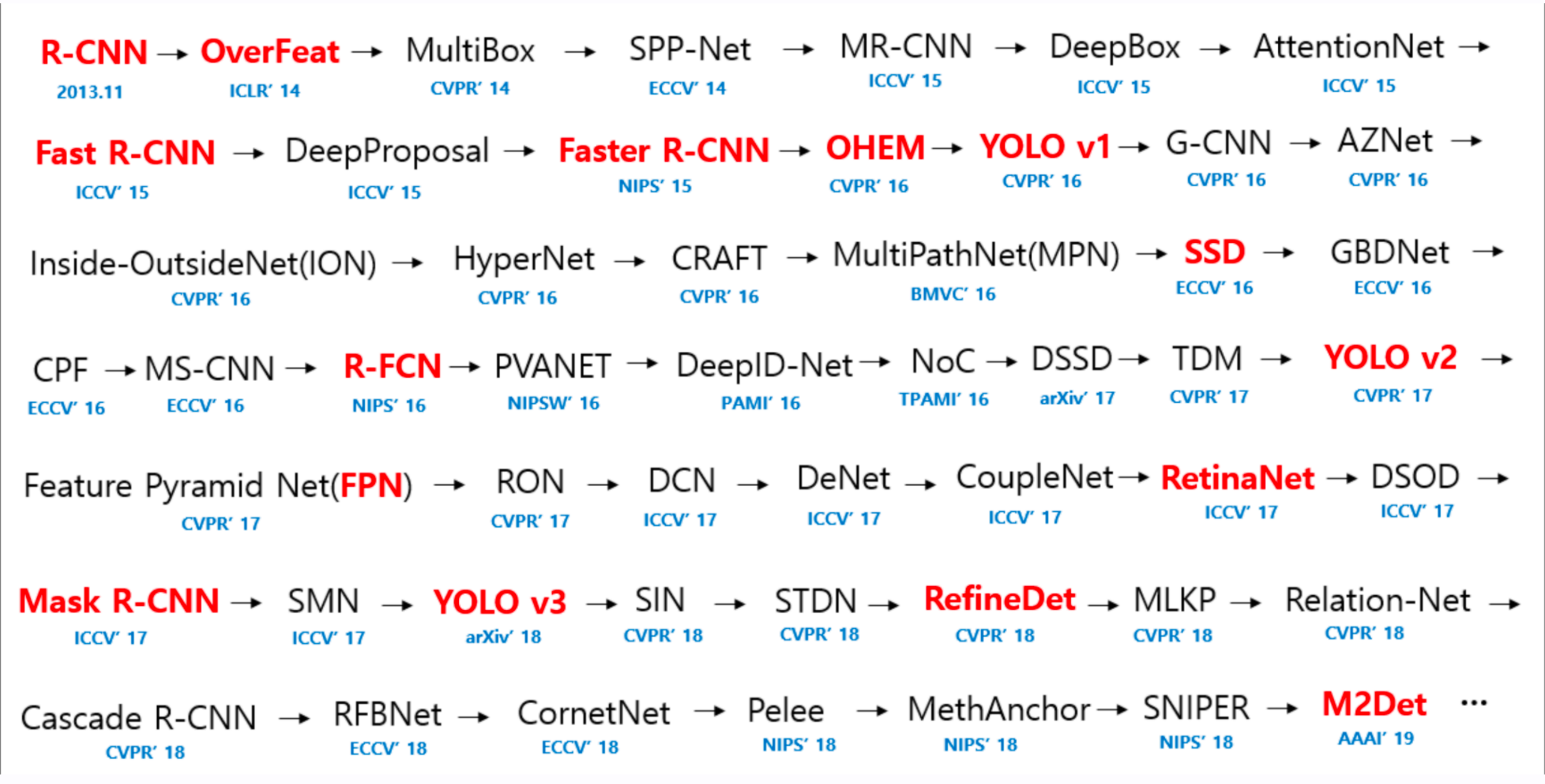


Complex Convolutional Networks

Directed acyclic computational graphs models used in production

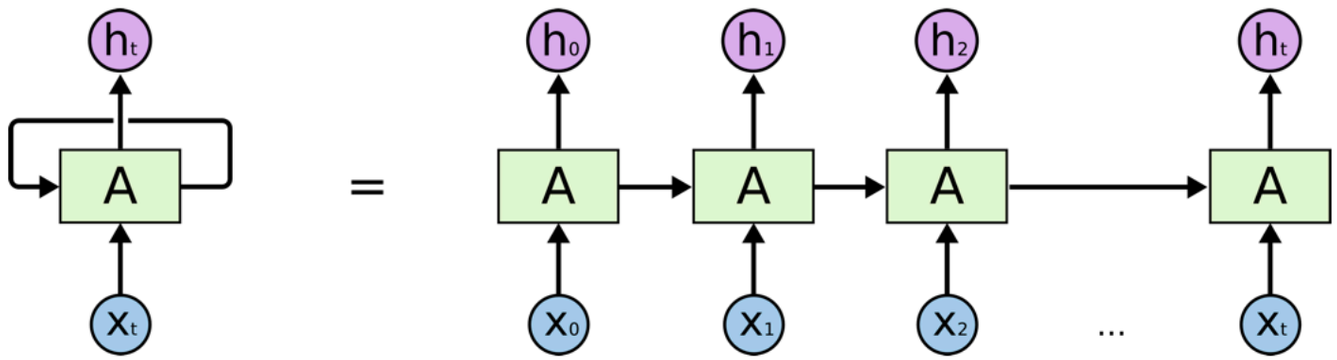


CNNs for Object Detection

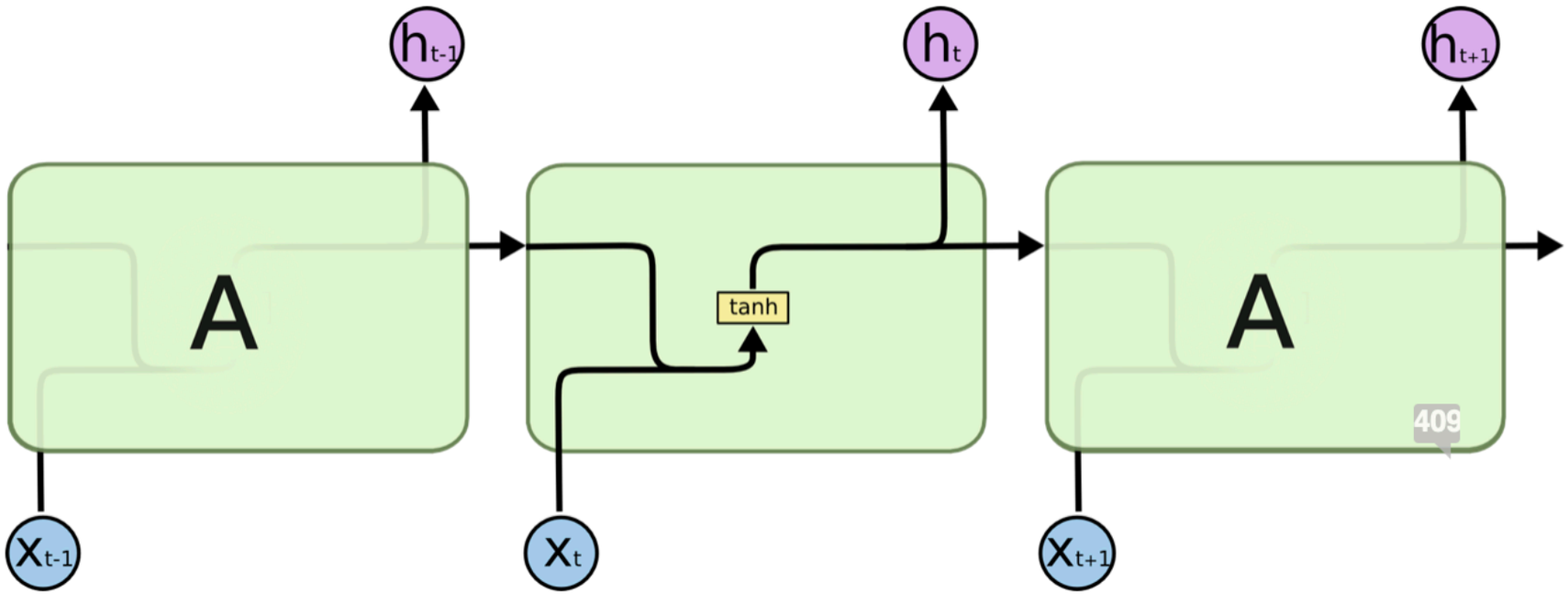


Recurrent Neural Networks

Back-propagation “thought-time” as a memory bottleneck and limitation for parallel execution



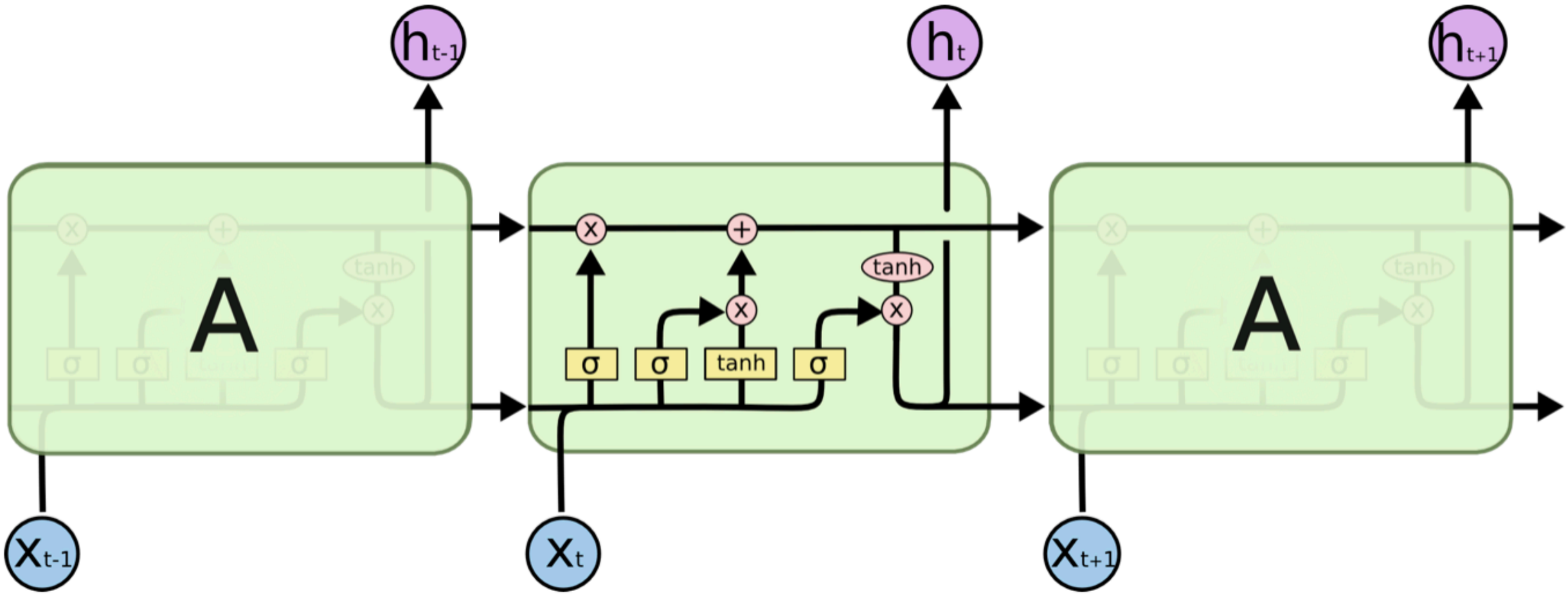
An unrolled recurrent neural network.



The repeating module in a standard RNN contains a single layer.

Long Short-Term Memory

Further increasing problem size
with “forget gate” & “modulation gate”



The repeating module in an LSTM contains four interacting layers.

Alpha Zero & Reinforcement

Anatomy of a world champion chess engine

Domain knowledge, extensions, heuristics in 2016 TCEC world champion *Stockfish*:

Board Representation:
Bitboards with Little-Endian Rank-File Mapping (LERF), Magic Bitboards, BMI2 - PEXT Bitboards, Piece-Lists,

Search:
Iterative Deepening, Aspiration Windows, Parallel Search using Threads, YBWC, Lazy SMP, Principal Variation Search.

Transposition Table:
Shared Hash Table, Depth-preferred Replacement Strategy, No PV-Node probing, Prefetch

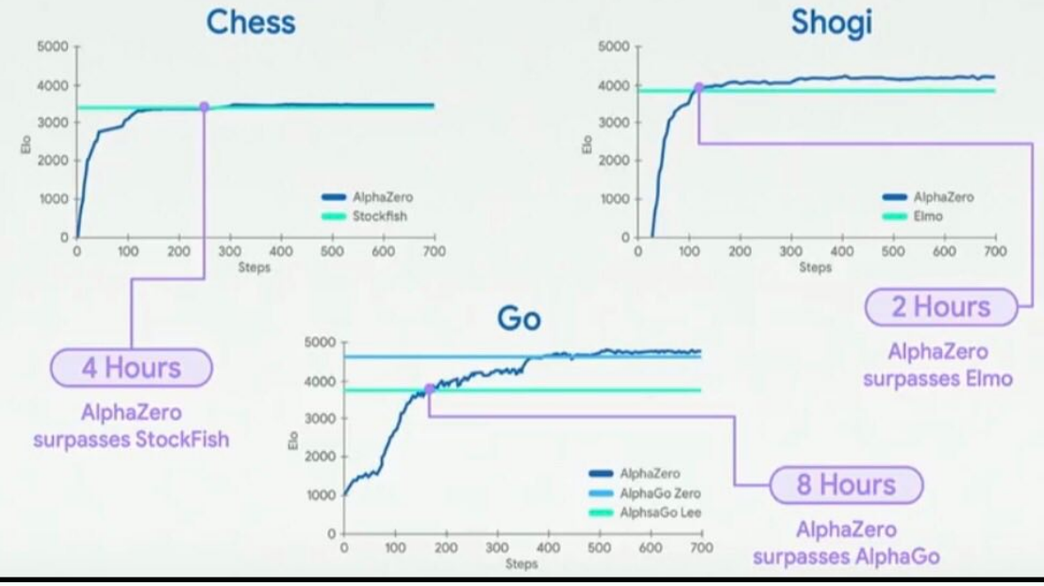
Move Ordering:
Countermove Heuristic, Counter Moves History, History Heuristic, Internal Iterative Deepening, Killer Heuristic, MVV/LVA, SEE,

Selectivity:
Check Extensions if SEE >= 0, Restricted Singular Extensions, Futility Pruning, Move Count Based Pruning, Null Move Pruning, Dynamic Depth Reduction based on depth and value, Static Null Move Pruning, Verification search at high depths, ProbCut, SEE Pruning, Late Move Reductions, Razoring, Quiescence Search

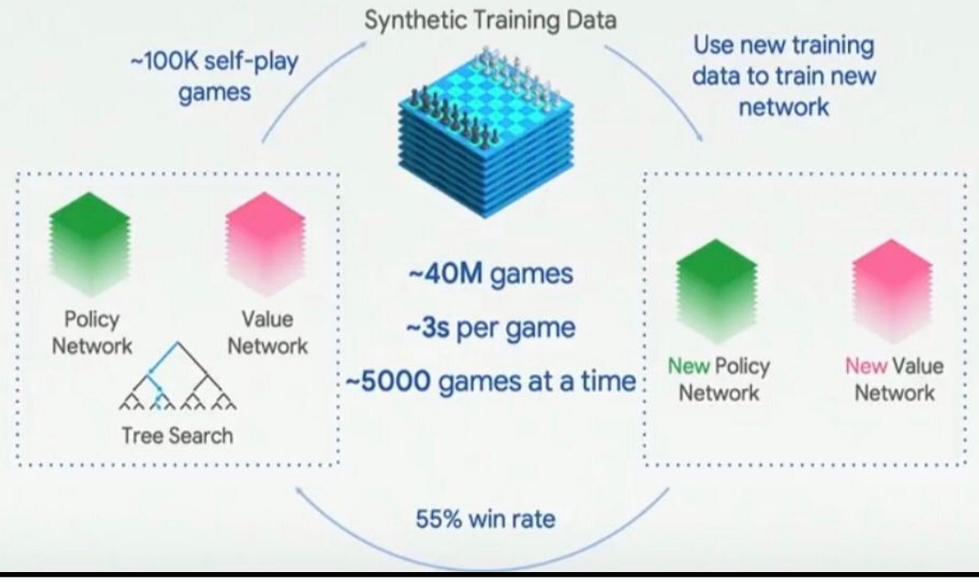
Evaluation:
Tapered Eval, Score Grain, Point Values Midgame: 198, 817, 836, 1270, 2521, Endgame: 258, 846, 857, 1278, 2558, Bishop Pair, Imbalance Tables, Material Hash Table, Piece-Square Tables, Trapped Pieces, Rooks on (Semi) Open Files, Outposts, Pawn Hash Table, Backward Pawn, Doubled Pawn, Isolated Pawn, Phalanx, Passed Pawn, Attacking King Zone, Pawn Shelter, Pawn Storm, Square Control, Evaluation Patterns,

Endgame Tablebases:
Syzygy TableBases

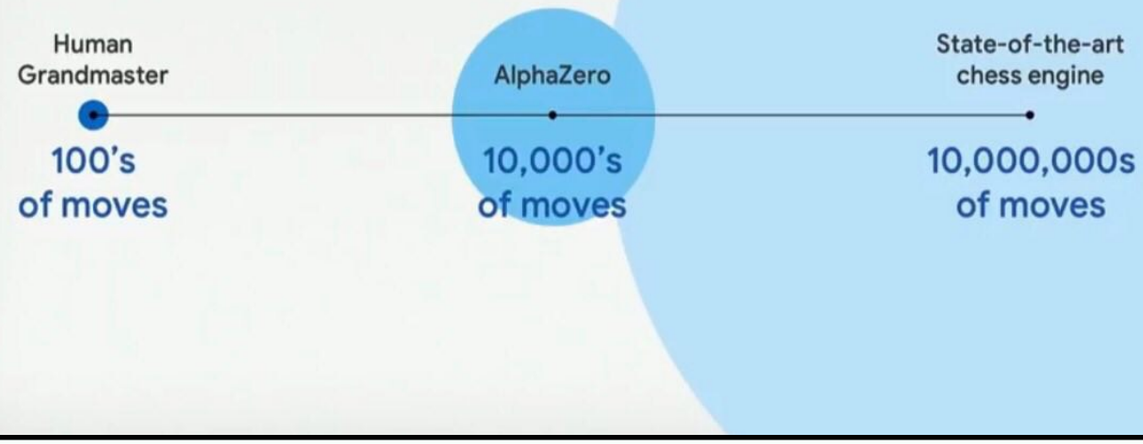
Same system can play any 2-player game



AlphaZero (AlphaGoZero)



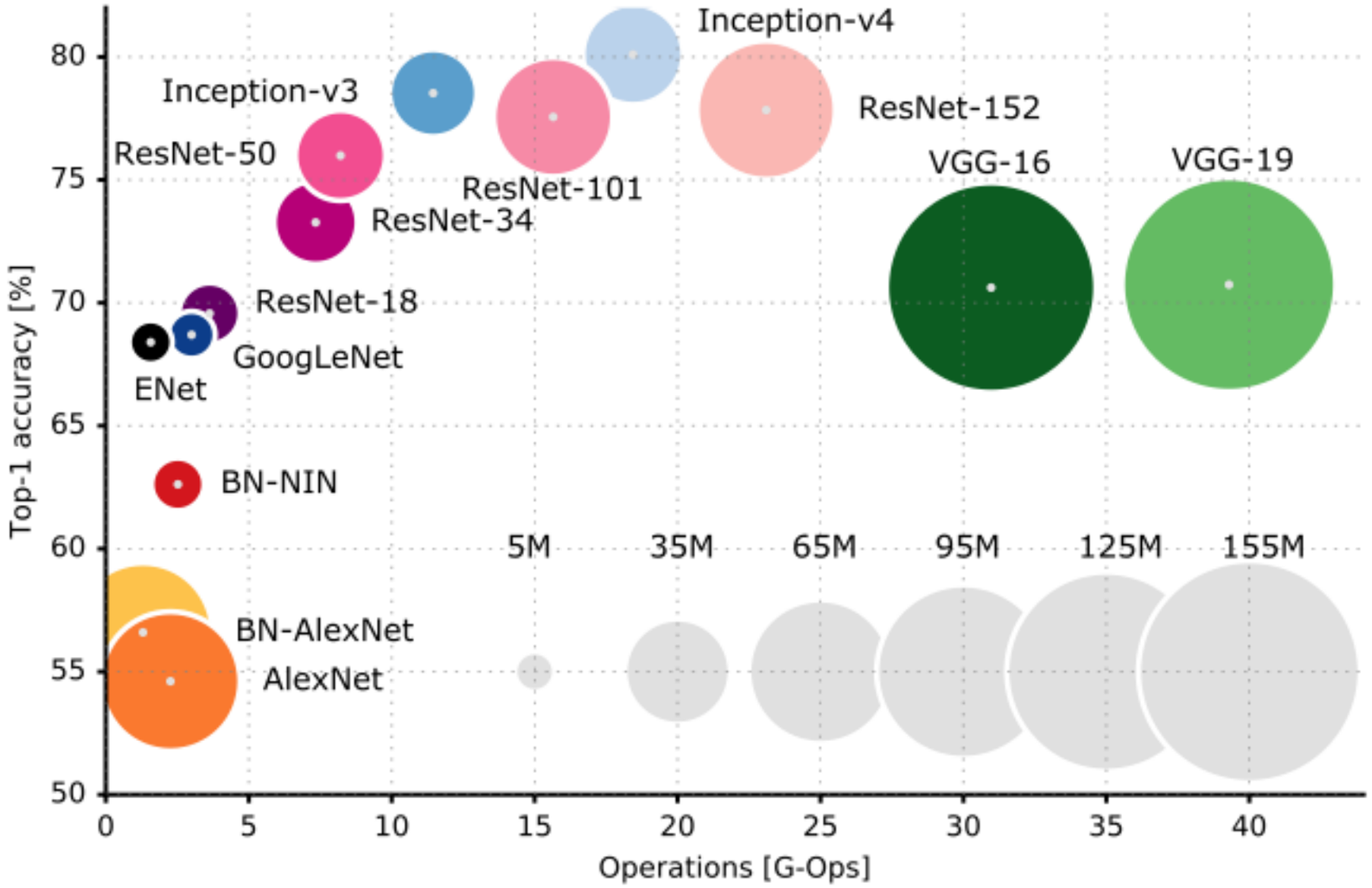
Amount of search per decision



Let’s look into some “ONNX” graphs

SotA Models Sizes

The deep models have enormous number of weights, which implies **long training time** and **potentially higher accuracy!**



Model Training Price

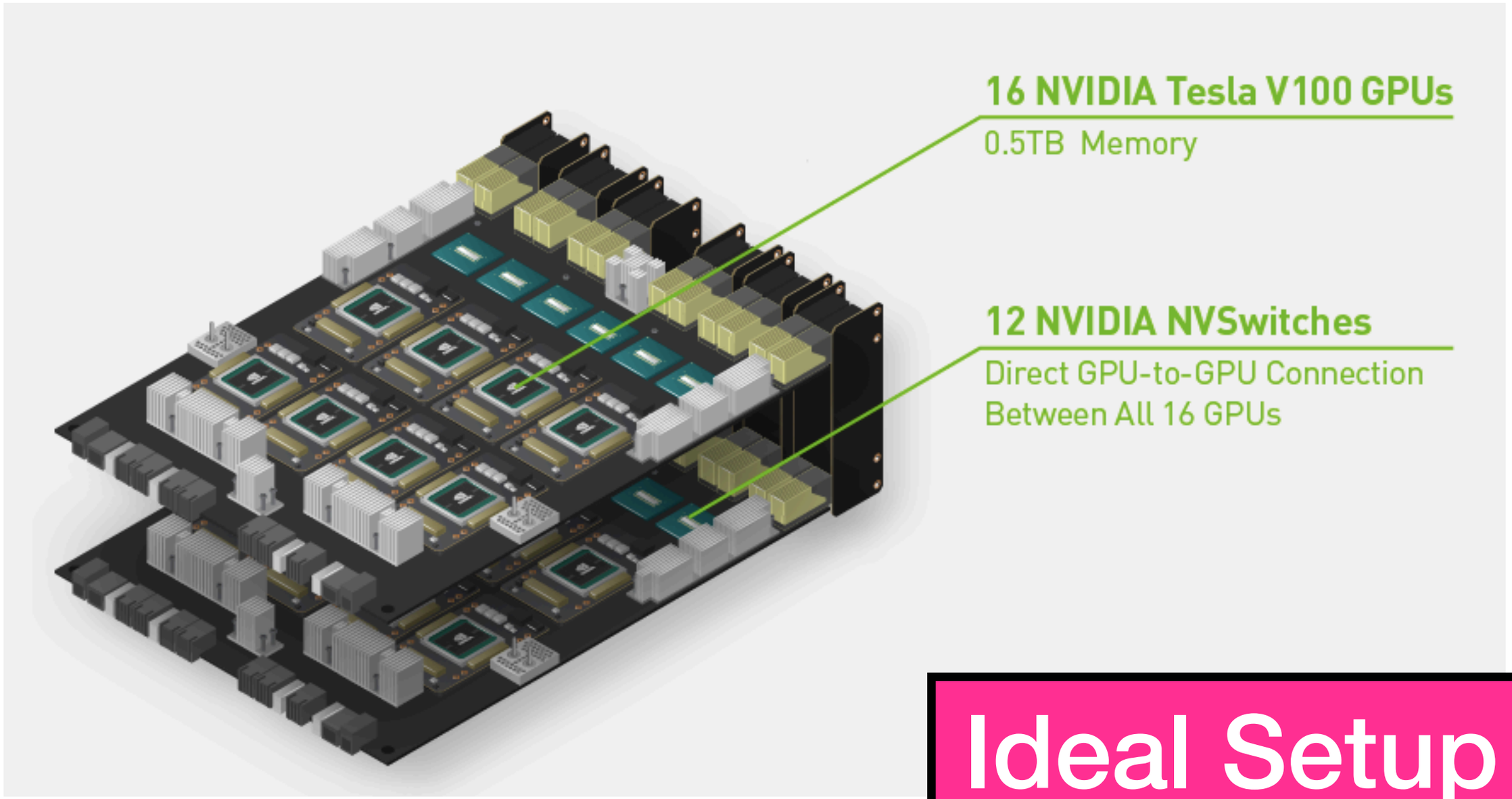
Inference of ANNs is relatively cheap, but training a single SotA model may cost over 100K USD in the cloud…

So the hardware must be optimized!

Product (CPU/GPU)	GPUs	On Demand $/Hour	3 Year Reserve $/Hour	3 Year Cost 100% Utilization
p2.8xlarge (64 vCores/K80)	8	$7.20	$3.40	$89,352.00 to $189,216.00
p3.16xlarge (64 vCores/V100)	8	$24.48	$9.87	$259,383.60 to $643,334.40
On-Premises (2x22 Xeon/P100)	8	~ $2.59	-	~ $68,000.00
On-Premises (Xeon/V100)	8	~ $3.81	-	~ $100,000.00

State of the Art:
AI Hardware

Why are those instances so expensive?



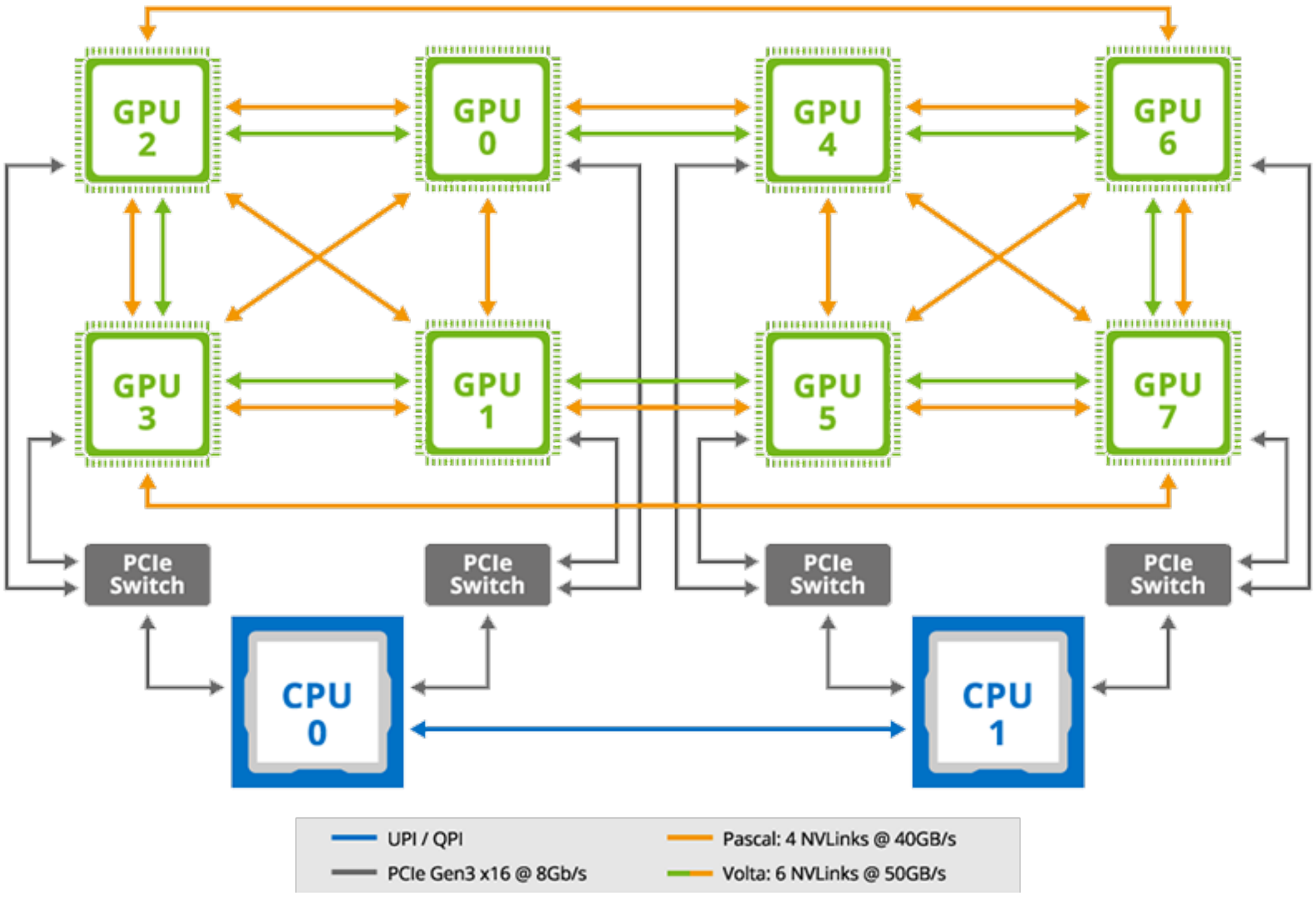
Ideal Setup

Good Setup


TITAN
TITAN
TITAN
TITAN
TITAN
TITAN
AIRFLOW DIRECTION
WARNING

Bad Setup

Proximity of memory buffers defines the performance limitations.
Every framework must track the **NUMA** layout to fully use **RDMA**.



Performance Metrics

	Intel Xeon Platinum 8180M	Nvidia Titan V	Google TPU v3
Year	2017	2017	2018
Memory	< 1.5 TB DDR4	12 GB HBM2	4x16 GB HBM
Memory Bandwidth	< 150 GB/s	650 GB/s	2.4 TB/s
TDP	205 W	250 W	~ 250 W
Fabrication Process	14 nm	12 nm	12-20 nm
Floating point Performance	< 10 TOPs	15 (120) TOPs	180 TOPs

State of the Art:
AI Libraries

Library Zoo

Spark MLlib

Lua

theano

scikit learn

K

dmlc XGBoost

Darknet

Microsoft Cognitive Toolkit

PyTorch

lib

mxnet

OpenCV

mlpack

ONNX

C++	Python
Manual Memory Control	Reference Counting for every variable
Compiled	Interpreted
Scope-limited Lifetime	Variables are accessible outside the loop
Statically-typed	Dynamically-typed
Floating point types are float and double	Floats are implementation specific, but normally have the capacity of `double`
Integer types are hardware defined	At least 16 bytes for zero integer, 24 bytes in most cases, 36 bytes or more for flexible
No proper modules	> 80 K easily pluggable modules
Complicated Optimised Code	Simple Slow Code

	C++		**Python**	
Tasks	Time	Memory	Time	Memory
Mandelbrot	2 s	26 Kb	260 s	52 Kb
N-Body	8 s	1 Kb	797 s	8 Kb
Spectral Norm	2 s	1 Kb	173	52 Kb
Fasta	1 s	2 Kb	63	683 Kb
Fannkuch Redux	10 s	1 Kb	472	50 Kb
Binary Trees	4 s	117 Kb	80	450 Kb
K-Nucleotide	4 s	156 Kb	73	188 Kb
Regex Redux	2 s	203 Kb	18	446 Kb
Rev Complement	2 s	996 Kb	18	1007 Kb
Pi Digits	2 s	4 Kb	3	10 Kb

Popular AI projects

	Tensorflow	PyTorch	MxNet	Keras	CNTK	Horovod
First Release	Nov, 2015	Jan, 2012	Apr, 2015	Mar, 2015	Aug, 2014	Aug, 2017
GitHub Stars	130 K	28 K	17 K	42 K	16 K	6 K
GitHub Watchers	8 K	1 K	1 K	2 K	1 K	280
Commits	57 K	18 K	9 K	5 K	16 K	300
Forks	75 K	7 K	6 K	16 K	4 K	1 K
Used by	41 K		400	27 K		58

Audience

	Tensorflow	PyTorch	MxNet	Keras	CNTK	Horovod
GitHub Repos	60 K	23 K	2 K	24 K	500	80
Arxiv Mentions	339	119	34	51	15	6
Google Trends	-3%	79%	2%	0%	-78%	42%
Interest Regions	China, Korea	China, Korea	China	Indonesia, China, Malaysia, Korea.	China, Indonesia, Korea	China, Korea

Codebases

	Tensorflow	PyTorch	MxNet	Keras	CNTK	Horovod
Last Commit ID	d38eaa19b3b656a4d4fd4ae388004a81a1c30094	c6255a57e4c15b88ded24195a9983a4aac401c22	f4598e743aa70e189db66e3d696f4cb52441d098	9d33a024e3893ec2a4a15601261f44725c6715d1	8ed70502bf1019e0ddeab8420d8babdd52bc7941	73d860f2396321761e0f5ef6fe934130afd69094
Total Files	10,778	4,715	2,856	252	1,727	161
Code Lines	2,251,532	710,449	406,488	46,618	331,206	17,557
Com. Lines	555,516	100,223	119,447	15,730	58,753	4,892
C++	53%	56%	35%	0%	59%	48%
Python	31%	26%	24%	93%	16%	44%
CUDA/ OpenCL	0%	13%	4%	0%	5%	0%

Let's take a look at some samples!

Using Keras for MLP

```
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))
```

```
model.summary()
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),
              metrics=['accuracy'])
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Using Keras for CNN

```
model = Sequential()
model.add(Conv2D(32, activation='relu',
                 kernel_size=(3,3),
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.summary()
model.compile(loss=losses.categorical_cross_entropy,
              optimizer=optimizers.Adadelta(),
              metrics=['accuracy'])
```

Using TF Python API for CNN

```
# Create some wrappers for simplicity
def conv2d(x, W, b, strides=1):
    # Conv2D wrapper, with bias and relu activation
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)


def maxpool2d(x, k=2):
    # MaxPool2D wrapper
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1],
                          padding='SAME')


# Create model
def conv_net(x, weights, biases, dropout):
    # MNIST data input is a 1-D vector of 784 features (28*28 pixels)
    # Reshape to match picture format [Height x Width x Channel]
    # Tensor input become 4-D: [Batch Size, Height, Width, Channel]
    x = tf.reshape(x, shape=[-1, 28, 28, 1])

    # Convolution Layer
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    # Max Pooling (down-sampling)
    conv1 = maxpool2d(conv1, k=2)

    # Convolution Layer
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    # Max Pooling (down-sampling)
    conv2 = maxpool2d(conv2, k=2)

    # Fully connected layer
    # Reshape conv2 output to fit fully connected layer input
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    # Apply Dropout
    fc1 = tf.nn.dropout(fc1, dropout)

    # Output, class prediction
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    return out
```

```
# Construct model
logits = conv_net(X, weights, biases, keep_prob)
prediction = tf.nn.softmax(logits)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)


# Evaluate model
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y, keep_prob: 0.8})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y,
                                                                 keep_prob: 1.0})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for 256 MNIST test images
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: mnist.test.images[:256],
                                      Y: mnist.test.labels[:256],
                                      keep_prob: 1.0}))
```

Using TF C++ for CNN

```
DataSet data_set(“/path/normalized_car_features.csv");
Tensor x_data(DataTypeToEnum<float>::v(),
              TensorShape { data_set.x().size()/3, 3 });
std::copy_n(data_set.x().begin(), data_set.x().size(),
            x_data.flat<float>().data());

Tensor y_data(DataTypeToEnum<float>::v(),
              TensorShape { data_set.y().size(), 1 });
std::copy_n(data_set.y().begin(), data_set.y().size(),
            y_data.flat<float>().data());
```

Using TF C++ for MLP (1/3)

```
/// This contains the graph.
Scope scope = Scope::NewRootScope();

// Every operation injects itself into the graph.
auto x = Placeholder(scope, DT_FLOAT);
auto y = Placeholder(scope, DT_FLOAT);

// Make weights matrices.
auto w1 = Variable(scope, {3, 3}, DT_FLOAT);
auto assign_w1 = Assign(scope, w1, RandomNormal(scope, {3, 3}, DT_FLOAT));

auto w2 = Variable(scope, {3, 2}, DT_FLOAT);
auto assign_w2 = Assign(scope, w2, RandomNormal(scope, {3, 2}, DT_FLOAT));

auto w3 = Variable(scope, {2, 1}, DT_FLOAT);
auto assign_w3 = Assign(scope, w3, RandomNormal(scope, {2, 1}, DT_FLOAT));

// Make bias vectors.
auto b1 = Variable(scope, {1, 3}, DT_FLOAT);
auto assign_b1 = Assign(scope, b1, RandomNormal(scope, {1, 3}, DT_FLOAT));

auto b2 = Variable(scope, {1, 2}, DT_FLOAT);
auto assign_b2 = Assign(scope, b2, RandomNormal(scope, {1, 2}, DT_FLOAT));

auto b3 = Variable(scope, {1, 1}, DT_FLOAT);
auto assign_b3 = Assign(scope, b3, RandomNormal(scope, {1, 1}, DT_FLOAT));
```

Using TF C++ for MLP (2/3)

```
auto layer_1 = Tanh(scope, Add(scope, MatMul(scope, x, w1), b1));
auto layer_2 = Tanh(scope, Add(scope, MatMul(scope, layer_1, w2), b2));
auto layer_3 = Tanh(scope, Add(scope, MatMul(scope, layer_2, w3), b3));
```

```
auto regularization = AddN(scope, {
                           L2Loss(scope, w1),
                           L2Loss(scope, w2),
                           L2Loss(scope, w3)
                          });

auto loss = Add(scope,
                ReduceMean(scope, Square(scope, Sub(scope, layer_3, y)), {0, 1}),
                Mul(scope, Cast(scope, 0.01,  DT_FLOAT), regularization));
```

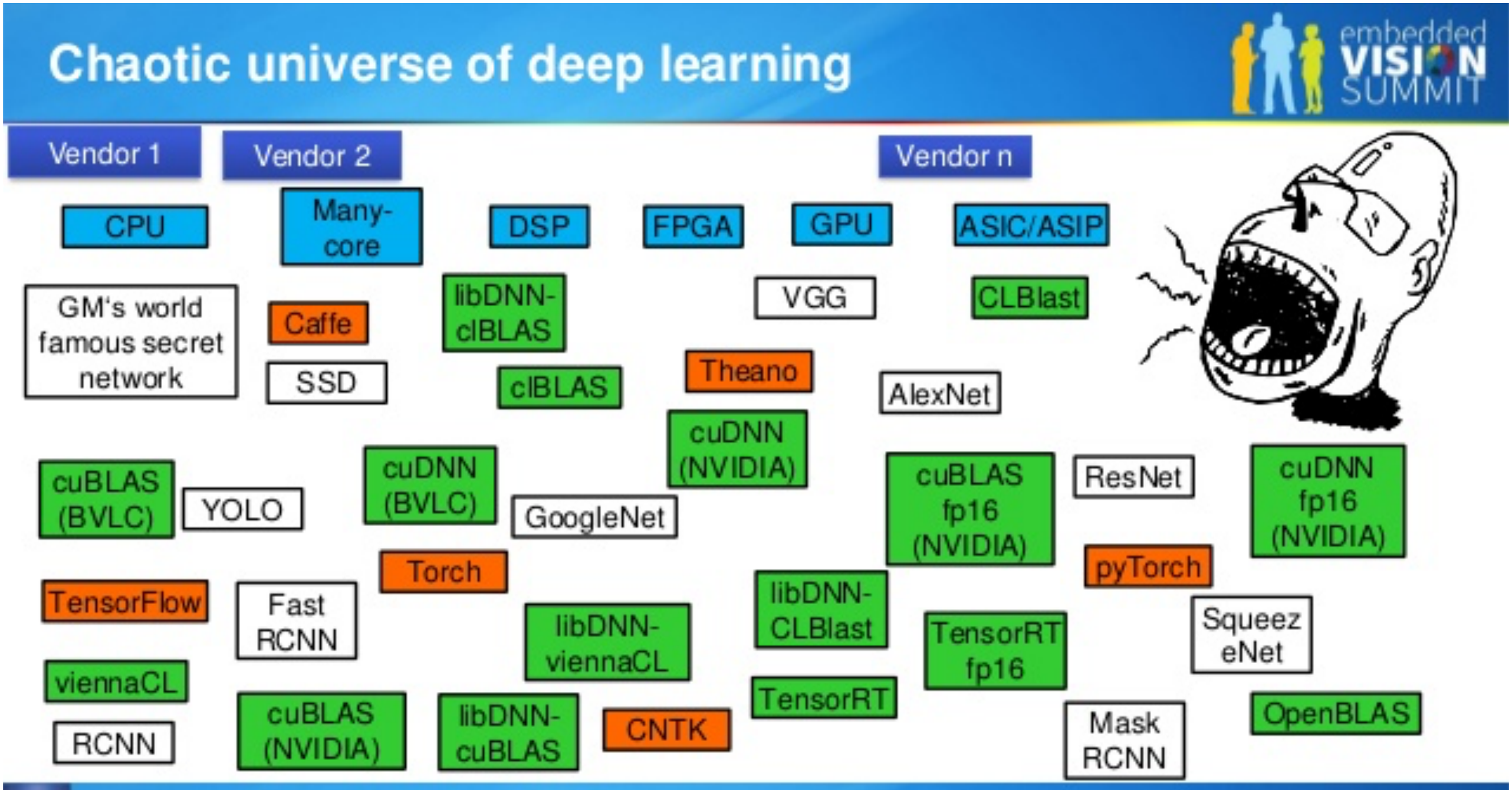
Using TF C++ for MLP (3/3)

```
// Backward pass:
std::vector<Output> grad_outputs;
AddSymbolicGradients(scope, { loss }, { w1, w2, w3, b1, b2, b3 }, &grad_outputs);

// Update the weights and bias using gradient descent.
auto apply_w1 = ApplyGradientDescent(scope, w1, Cast(scope, 0.01,  DT_FLOAT), {grad_outputs[0]});
auto apply_w2 = ApplyGradientDescent(scope, w2, Cast(scope, 0.01,  DT_FLOAT), {grad_outputs[1]});
auto apply_w3 = ApplyGradientDescent(scope, w3, Cast(scope, 0.01,  DT_FLOAT), {grad_outputs[2]});
auto apply_b1 = ApplyGradientDescent(scope, b1, Cast(scope, 0.01,  DT_FLOAT), {grad_outputs[3]});
auto apply_b2 = ApplyGradientDescent(scope, b2, Cast(scope, 0.01,  DT_FLOAT), {grad_outputs[4]});
auto apply_b3 = ApplyGradientDescent(scope, b3, Cast(scope, 0.01,  DT_FLOAT), {grad_outputs[5]});

ClientSession session(scope);
std::vector<Tensor> outputs;

// Init the weights and biases by running the assigns nodes.
session.Run({assign_w1, assign_w2, assign_w3, assign_b1, assign_b2, assign_b3}, nullptr);

// Training steps.
for (int i = 0; i < 5000; i ++) {
  session.Run({{x, x_data}, {y, y_data}}, {loss}, &outputs);
  session.Run({{x, x_data}, {y, y_data}}, {apply_w1, apply_w2, apply_w3, apply_b1, apply_b2,
apply_b3, layer_3}, nullptr);
}
```

Using PyTorch C++

```
/// Automatically differentiable, CPU/GPU-enabled arrays.
torch::Tensor;

/// Composable modules for ANNs.
torch::nn;

/// Optimization algorithms: SGD, Adam or RMSprop…
torch::optim;

/// Datasets, pipelines and multi-threaded, async data loader.
torch::data;

/// Serialization API for model checkpoints.
torch::serialize;

/// Glue to bind your C++ models into Python.
torch::python;

/// Pure C++ access to the TorchScript JIT compiler.
torch::jit;
```

The front-ends of all major libraries look the same, but is that all?
The complexity arises in the back-end part!



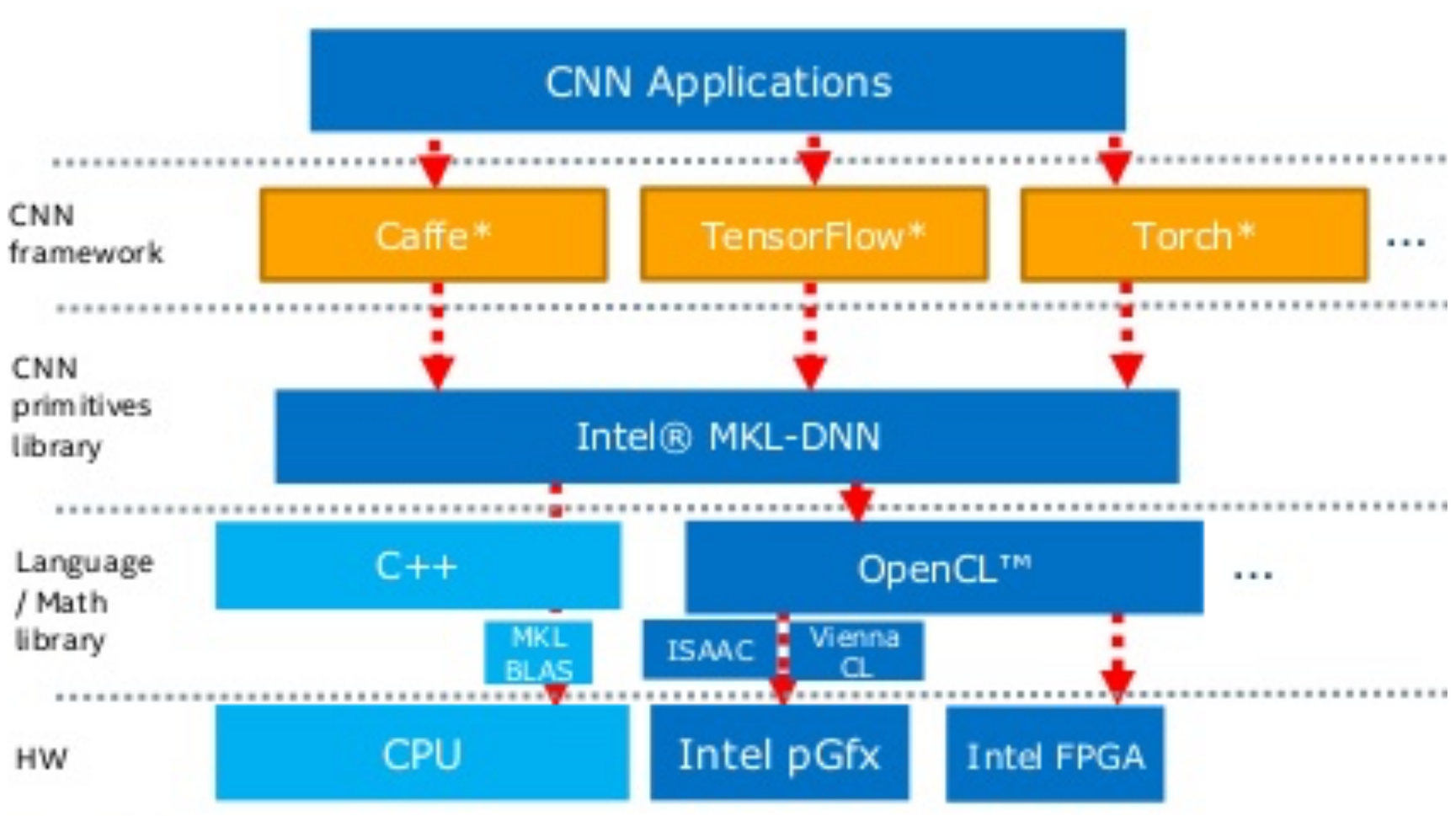
Deep Learning Stack

AI frameworks span across all those layers and often incorporate **multiple libraries at every layer** for maximum deployment flexibility!

Deep Learning	Keras, TF, cuDNN, MKL-DNN
Computational Graphs*	Eigen, TF, VexCL, ArrayFire
Math	Eigen, MKL, VexCL, cuBLAS, ArrayFire, Boost.Compute
Parallelism	Intel TBB, OpenML, OpenACC
Language & Extensions*	CUDA, OpenCL, OpenML, OpenACC
Compilers*	LLVM, TVM, GCC

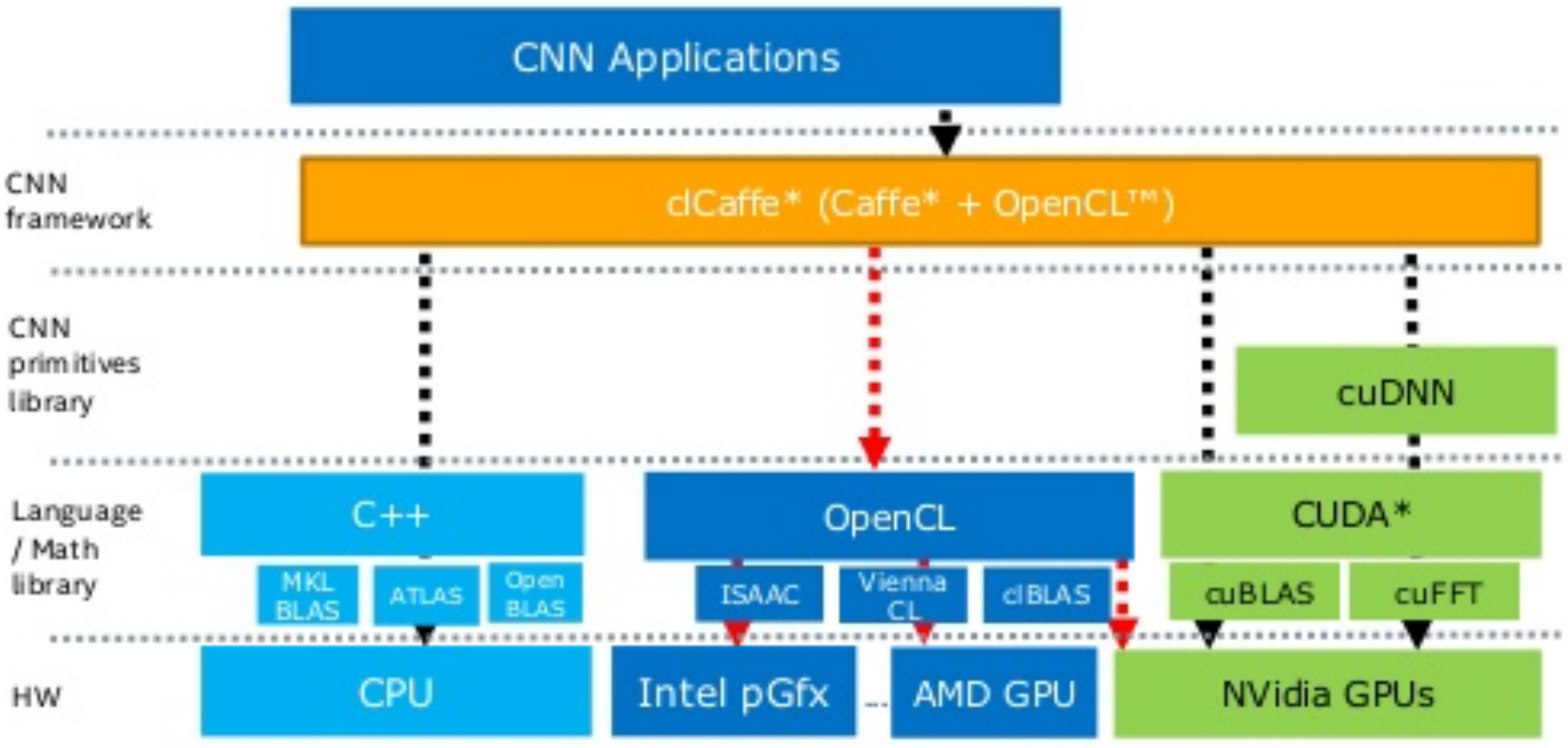
Intel Software Stack

Intel provides multiple libraries optimised for their CPUs.
They have different names, but their functionality is often similar.
MKL, MKL-DNN, BLAS, TBB.



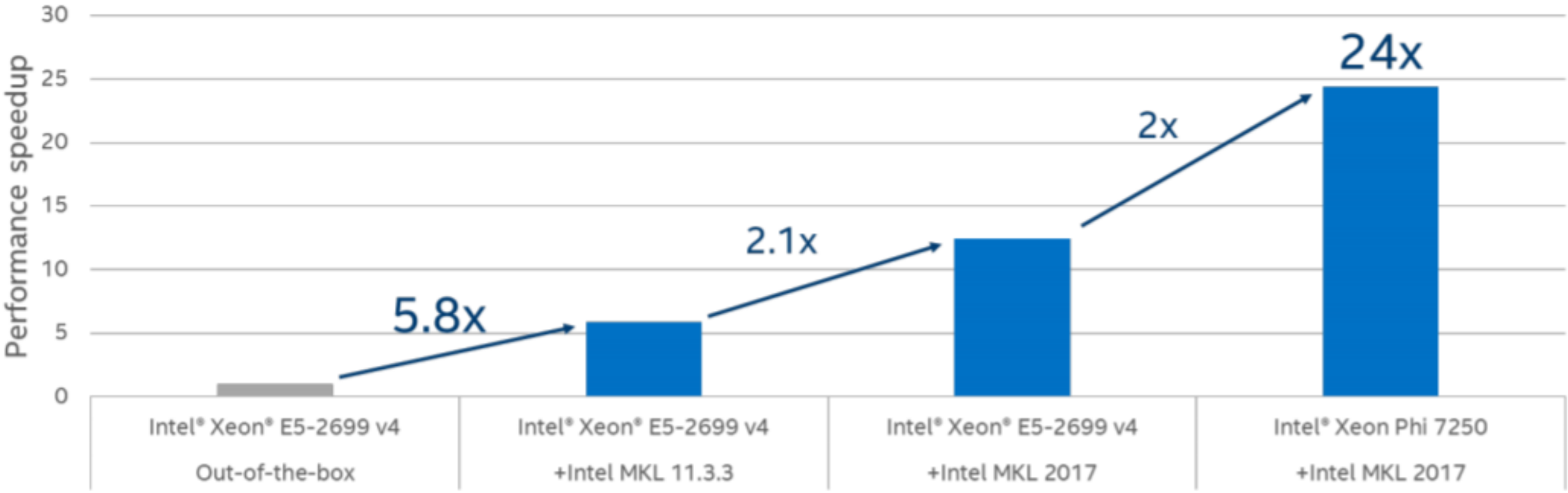
Intel GPU Stack

Intel supported OpenCL as primary language for their integrated GPUs.
ViennaCL, clBLAS, Vulkan.



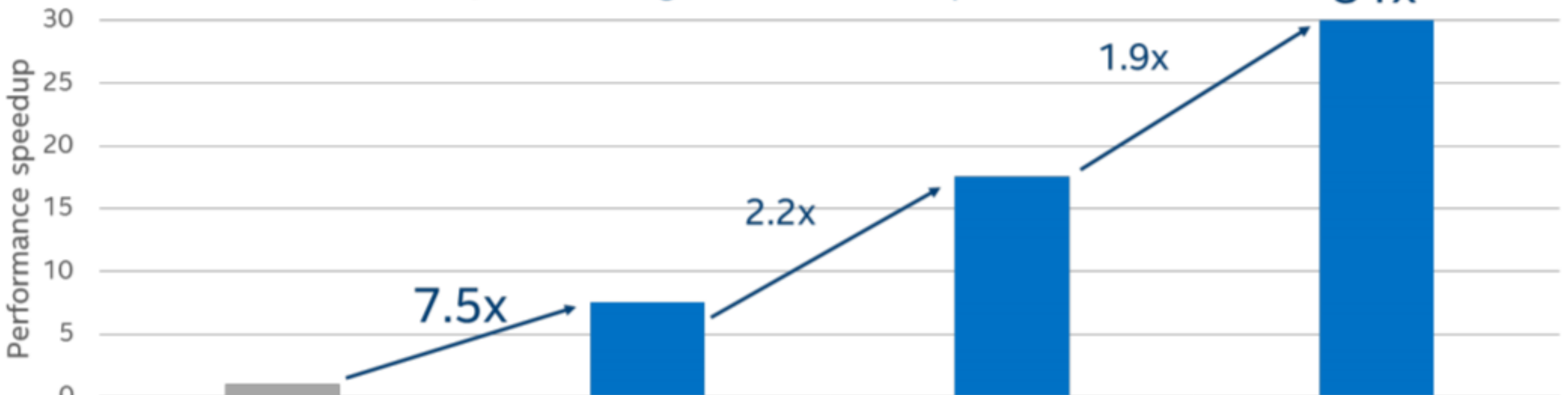
Improved Deep Neural Network training performance using Intel® Math Kernel Library (Intel® MKL)

Caffe/AlexNet single node training performance



Improved Deep Neural Network inference performance using Intel® Math Kernel Library (Intel® MKL)

Caffe/AlexNet single node inference performance



Nvidia Software Stack

1. Math:
    1. **cuBLAS**: Linear Algebra
    2. cuSPARSE: Sparse MAtrix Operations
    3. **cuDNN**: Deep Learning Primitives
2. Device-Specific:
    1. **Automatic Mixed Precision**: Tensor Cores Support on Volta
    2. OpticalFlowSDK: Optical Flow for Video Inference on Turing
3. Networking:
    1. **NCCL**: Multi-GPU Communication
4. Pre/Post-processing:
    1. **TensorRT**: Deep Learning Inference Engine
    2. DALI: Input Data Processing
    3. DeepStreamSDK: Deep Learning for Video Analytics
    4. AI-Assisted Annotation SDK: AI enabled Annotation for Medical Imaging
    5. Transfer-Learning Toolkit

Front End

Graph Optimizer

Tensor Optimizer

VTA JIT Runtime

Compiler and
Code Generation

VTA ISA

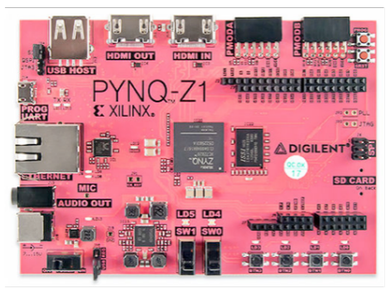
VTA Micro-Architecture

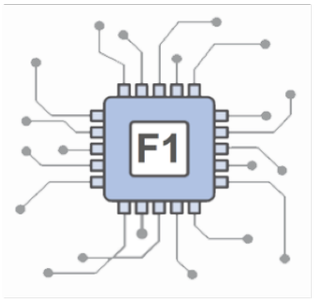
Hardware
Design

Edge FPGA

Cloud FPGA (in progress)

Simulator





Hardware
Deployment

Complex Demo:

PyTorch,Clang,Git

Future of C++

Proposal P1019.
Executors with STL Algo

```
auto sum = std::reduce(std::execution::par, data);

std::execution::static_thread_pool pool(4);
sum = std::reduce(par.on(pool.executor()), data);

my_gpu_executor my_executor;
sum = std::reduce(par.on(my_executor), data);
```

Proposal P0009.
mdspan: K-Dimensional Range

```
vector<float> own(X * Y);
mdspan<float, dimensions<dyn, dyn>> ref(data.data(), X, Y);

expect(ref.rank() == 2);
expect(ref.size() == X * Y);
expect(ref.extent(0) == X);
expect(ref.extent(1) == Y);
expect(ref.is_regular() == true);
expect(ref.stride(0) == 1);
expect(ref.stride(1) == 1);
expect(ref.span() == X * Y);

for (int i = 0; i < ref.extent(0); i ++) {
   for (int j = 0; j < ref.extent(1); j ++) {
      ref(i, j) = rand();
   }
}
```

Proposal P1436.
Affinity in Executors

```
// Current platform-dependant versions:
// Solaris: pbind()
// Linux: sched_setaffinity()
// Windows: SetThreadAffinityMask()

enum class bulk_execution_affinity {
   none, balanced, scatter, compact
};

executor exec;
auto exec_subs = execution::require(exec, execution::bulk,
                                    bulk_execution_affinity.scatter);
exec_subs.bulk_execute([](std::size_t i, shared s) {
   func(i);
}, 8, shared_factory);
```

Proposal P0798.
Monadic std::optional

```
std::optional<image> get_einstein_pic(image const & img) {
    return crop_to_face(img)
           .and_then(add_bow_tie)
           .and_then(make_eyes_sparkle)
           .transform(add_glasses)
           .or_else([] {
               return [=] { std::cout << "Failed!"; };
           });
}
```

More proposals for numerics

- P1368: Multiplication and division of fixed-point numbers
- P0943: Support C atomics in C++
- P1438: A Rational Number Library for C++

Contacts

Ashot Vardanian

linkedin.com/in/ashvardanian
fb.com/ashvardanian