Deep Learning Inference for the Edge

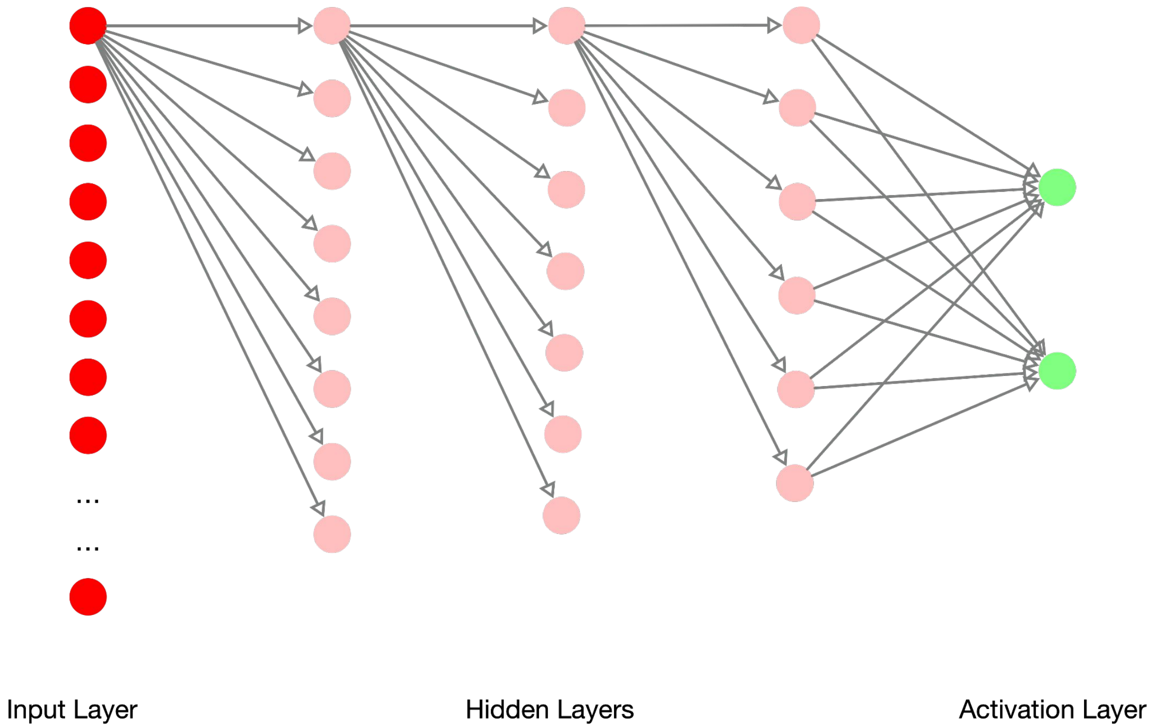
Sang Ryul “Eric” Pae and Joseph “Jay” Cordaro

9/3/2022

Outline

- DNN
- Edge Inference
- NN basics + Backpropagation
- Architectures for Edge Inference
- Benchmark Results
- Pros and Cons
- Future Work

Deep Learning



Deep Learning/DNN:

Neural Network with multiple hidden layers between input and output layers.

Deep Learning

- Deep Learning is a “Greedy Data” algorithm – lots of training data required
  - Outperforms other methods when lots of data is available
- Can learn optimal features and a classifier (or regressor) simultaneously
- Deep Neural Network (DNN) models are very difficult to interpret
- DNNs are tricky to train and easy to overfit (not covered here)
- DNNs trained using gradient descent with backpropagation
- Computationally expensive, but amenable to
  - hardware acceleration
  - frameworks
- Commercial-grade DNN models are deployed in the field and this remains an active research area as well.

Vocabulary

Tensor – Generalization of a matrix with an arbitrary number of indices.

Batch – number of samples in the training set per iteration

Batch Normalization [1] – normalize layer inputs during training. Allows faster learning rate

Epoch – one cycle of training the DNN with all the training data.

Parameter -- weights of the connections

Hyperparameter – training items like learning rate, number of epochs.
(confusing…Metaparameter would perhaps be better)

Expressive Power [2] – complexity of the function (decision or regression) DNN can implement. With proper training, a deeper network (with more hidden layers) can classify more complex input. **Initial layers of a DNN matter more**, and DNN has more expressive power when early weights are optimized. Increased expressive power comes with a downside and that is **overfitting**. Because if your network is too powerful it can overfit the data.

[1] https://arxiv.org/abs/1502.03167
[2] https://ganguli-gang.stanford.edu/pdf/17.ExpressivePower.pdf

Why Edge Inference?

Why not just do inference e.g. Key Word Spotting “Alexa” / “Hey Siri” in cloud on GPUs?

Latency

May need inference more quickly than a cloud compute model can provide

Power

Sending data over WiFi/Satellite/Cell network is very power intensive (battery life)

Scalability

If everyone uploaded raw data 24/7 to the cloud could the infrastructure handle it?

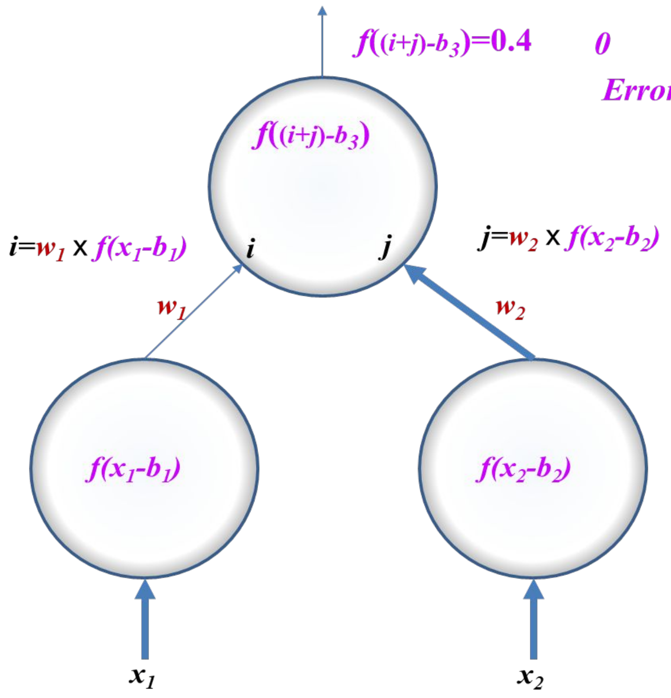
Access

Internet may not be available and a GPU may require too much power

Person detection vision, and KWS DNNs require 50K- 500K+ parameters for production models

Neural Networks

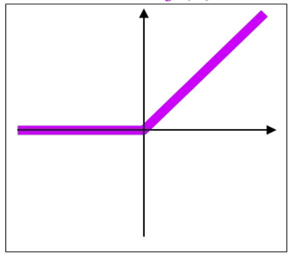
Artificial Neural Networks



0

Error=0.4-0

RELU: f(x)



dE/dw1 =?

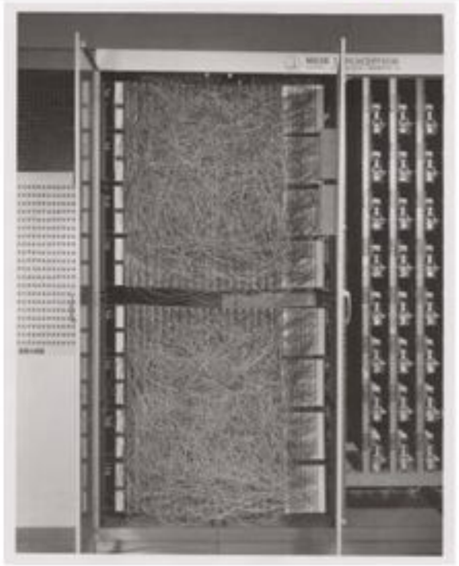
$x = -2, -1, 0, 5, 6$
$f(x-2)=0, 0, 0, 3, 4$

Back-propagation

https://youtu.be/tIeHLnjs5U8

Activation functions:
https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/#:~:text=5.-,ReLU,neurons%20at%20the%20same%20time.

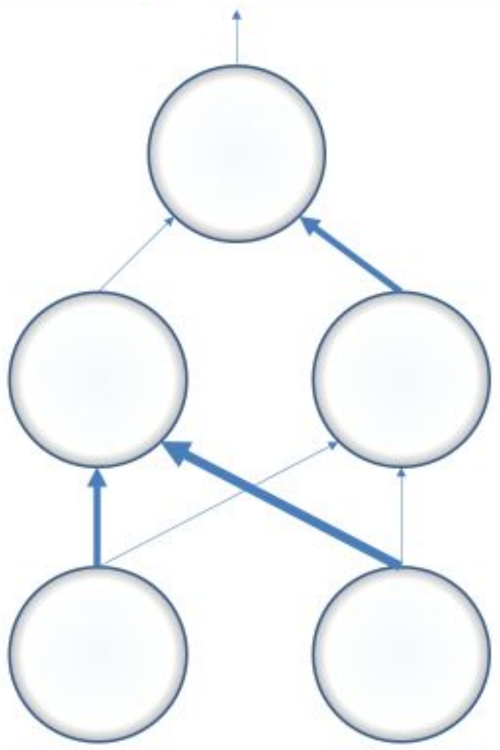
Perceptron(Mark I : single layer)



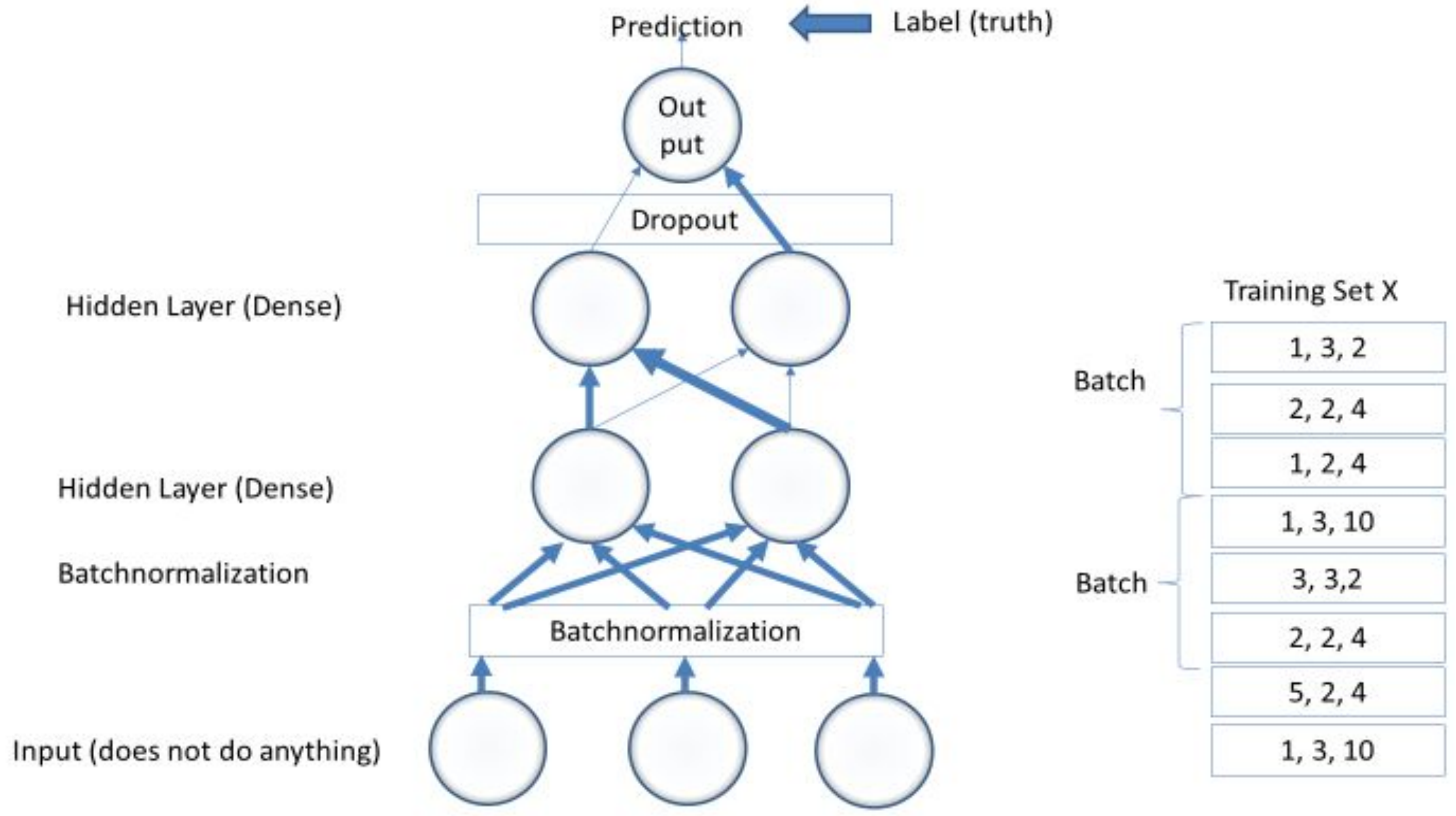
Mark I Perceptron machine, the first implementation of the perceptron algorithm. It was connected to a camera with 20×20 cadmium sulfide photocells to make a 400-pixel image. The main visible feature is a patch panel that set different combinations of input features. To the right, arrays of potentiometers that implemented the adaptive weights.

In a 1958 press conference organized by the US Navy, Rosenblatt made statements about the perceptron that caused a heated controversy among the fledgling AI community; based on Rosenblatt's statements, *The New York Times* reported the perceptron to be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." (Olazaran, 1996)

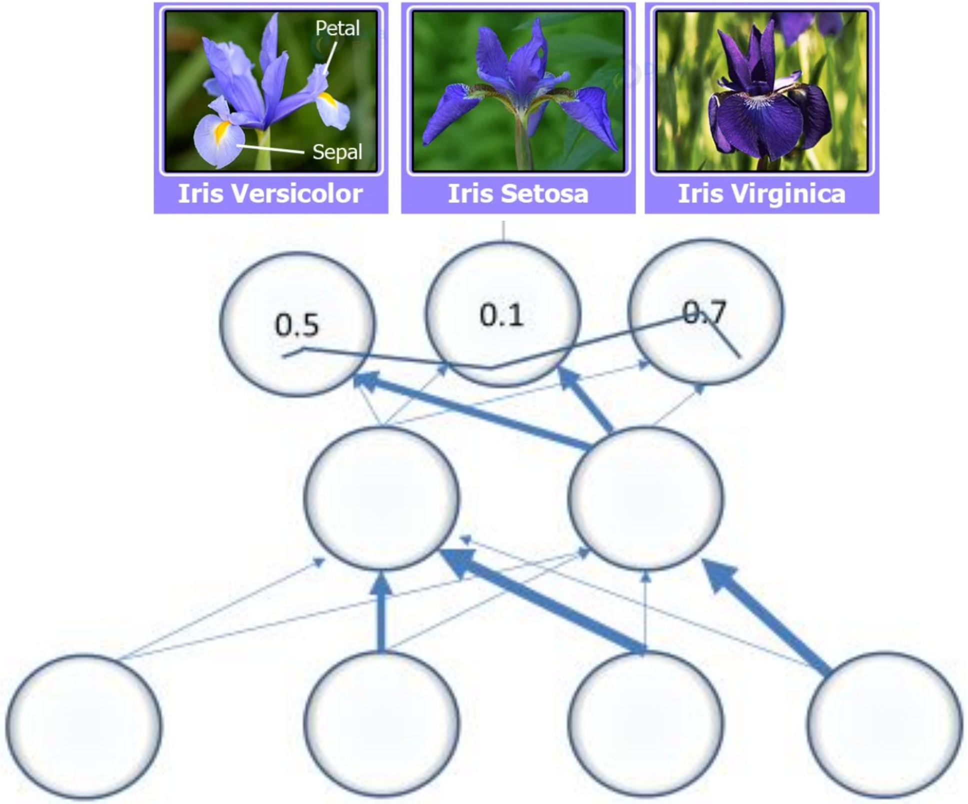
Multi-Layer Perceptrons



Multi-Layer Perceptrons



Iris Example



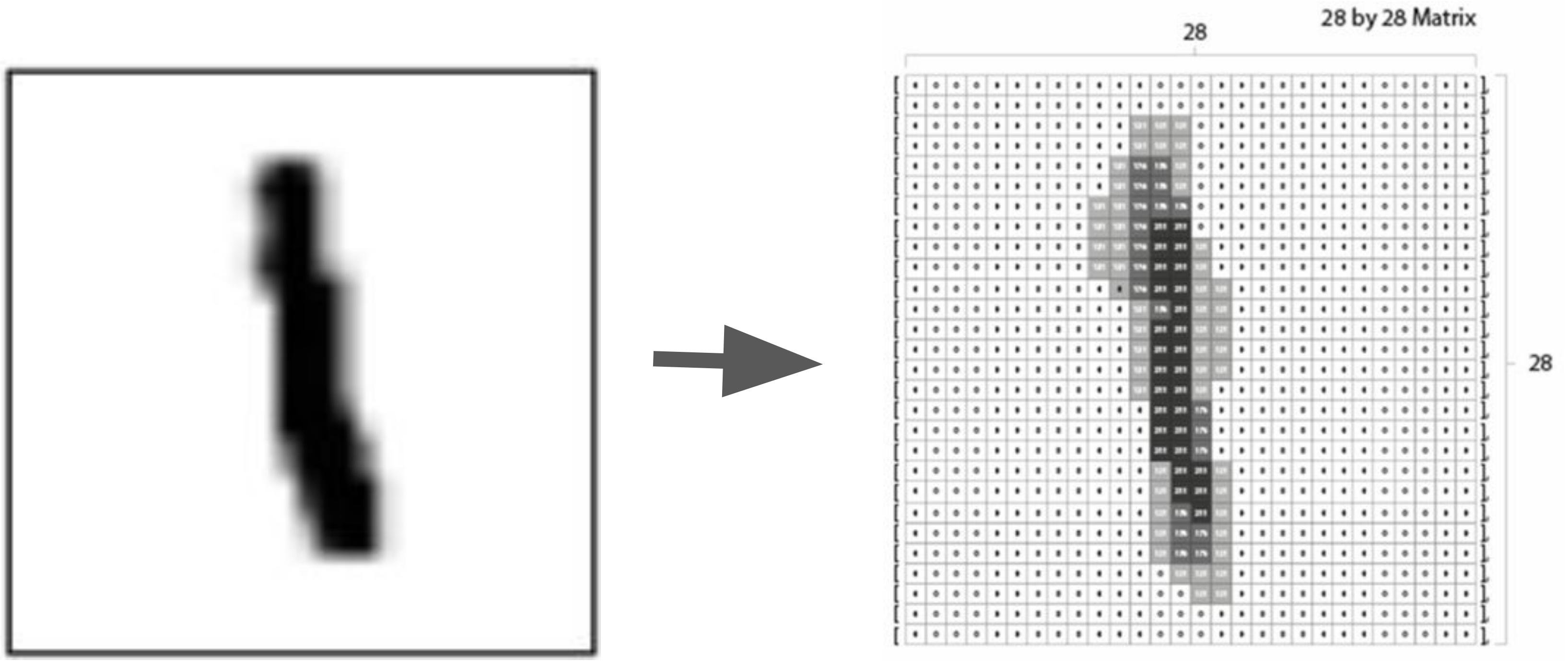
Modeling example: MNIST

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

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

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

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

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

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

model.evaluate(x_test,  y_test, verbose=2)
```

Tensorflow 2.0 for Edge TPU Programming
April 23th, 2020
Andrés L. Martínez
@davilagrau

https://colab.research.google.com/drive/1LFfW6cTUAsz9QB79OivXG5oQmzFJsWD0#scrollTo=NJWqEVrrJ7ZB

Flattening example



Number of Parameters in MNIST Modelling Example

		# parameters
Input to Dense Layer	28x28x128	100,352
Dense Layer to Dropout	.8x	
Dropout to 2nd Dense	128x10+10	1290
		101642

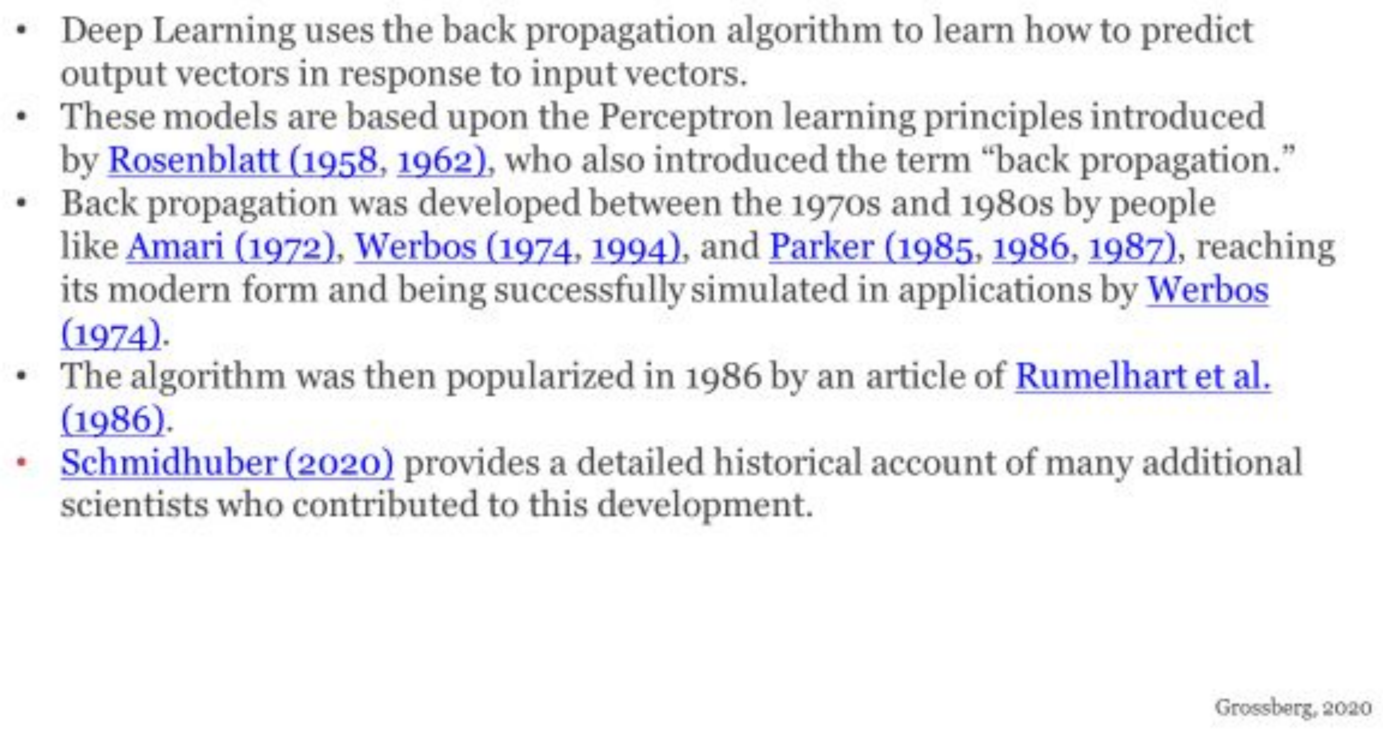
> 100k parameters just to categorize a digit as 0-9

Back-Propagation

(Derivative of the loss with respect to weights)   dl/dW

(Does it recursively backward)

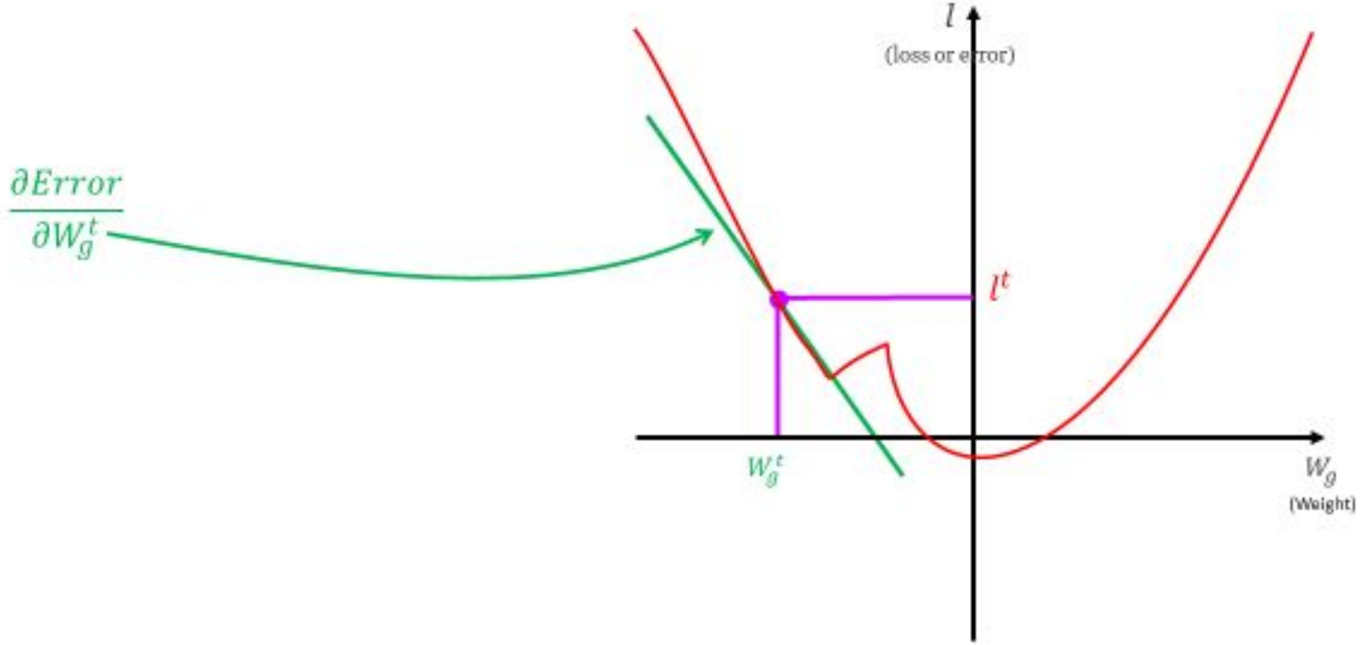
Back-Propagation



- Deep Learning uses the back propagation algorithm to learn how to predict output vectors in response to input vectors.
- These models are based upon the Perceptron learning principles introduced by Rosenblatt (1958, 1962), who also introduced the term "back propagation."
- Back propagation was developed between the 1970s and 1980s by people like Amari (1972), Werbos (1974, 1994), and Parker (1985, 1986, 1987), reaching its modern form and being successfully simulated in applications by Werbos (1974).
- The algorithm was then popularized in 1986 by an article of Rumelhart et al. (1986).
- Schmidhuber (2020) provides a detailed historical account of many additional scientists who contributed to this development.

Grossberg, 2020

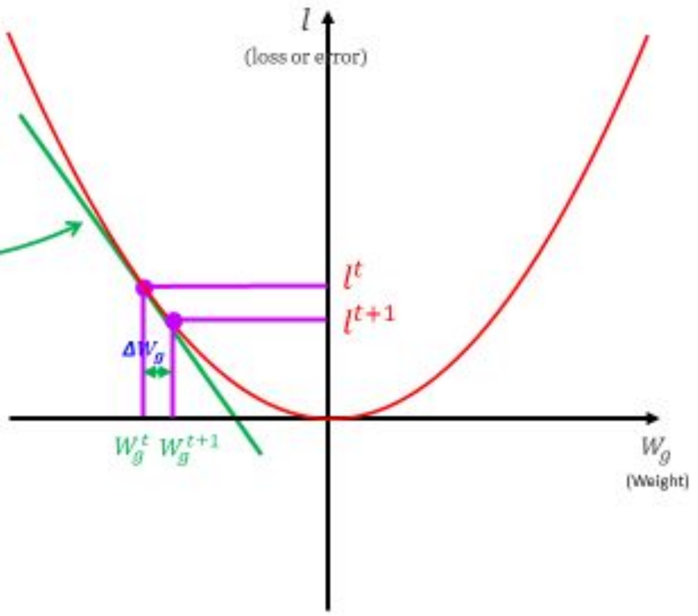
Gradient descent in Back-Propagation



https://youtu.be/tIeHLnjs5U8

Gradient descent in Back-Propagation

$$\frac{\partial l}{\partial W_g^t}$$



$$W_g^{t+1} \leftarrow W_g^t + \Delta W_g$$

$$\Delta W_g = -\eta \frac{\partial l}{\partial W_g^t}$$

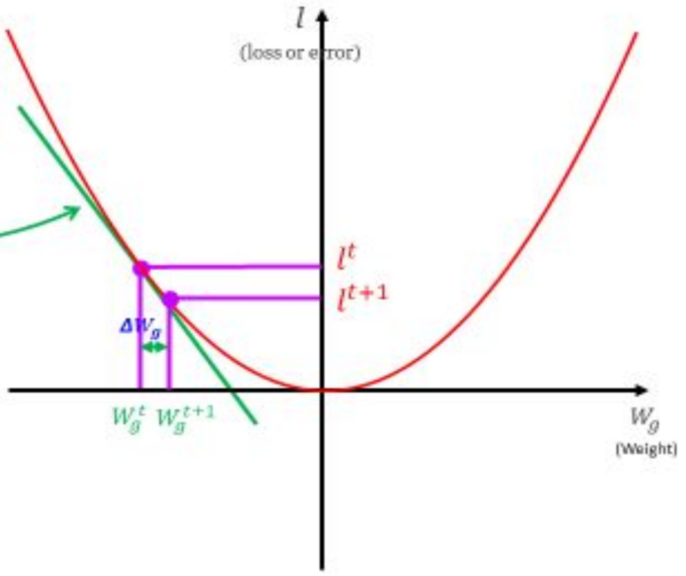
*Note: The sign and magnitude of the weight change (ΔW_g) depends on:

1) The steepness of the gradient
2) The sign of the gradient
3) Learning rate, η

https://youtu.be/tIeHLnjs5U8

Gradient descent in Back-Propagation

$$\frac{\partial l}{\partial W_g^t}$$



$$W_g^{t+1} \leftarrow W_g^t + \Delta W_g$$

$$\Delta W_g = -\eta \frac{\partial l}{\partial W_g^t}$$

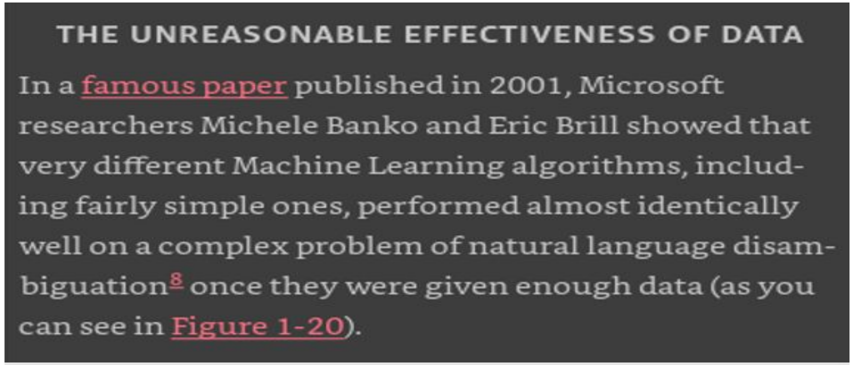
*Note: The sign and magnitude of the weight change (ΔW_g) depends on:

1) The steepness of the gradient
2) The sign of the gradient
3) Learning rate, η

Often times the gradient of loss (l) with respect to one weight (e.g., W_g) cannot be calculated directly.
So the chain-rule is used:

$$\frac{\partial l}{\partial W_g} = \frac{\partial z^t}{\partial W_g}\frac{\partial g^t}{\partial z^t}\frac{\partial s^t}{\partial g^t}\left(\frac{\partial h^t}{\partial s^t} + \frac{\partial h^t}{\partial s^{t-1}}\right)\frac{\partial l}{\partial h^t}$$

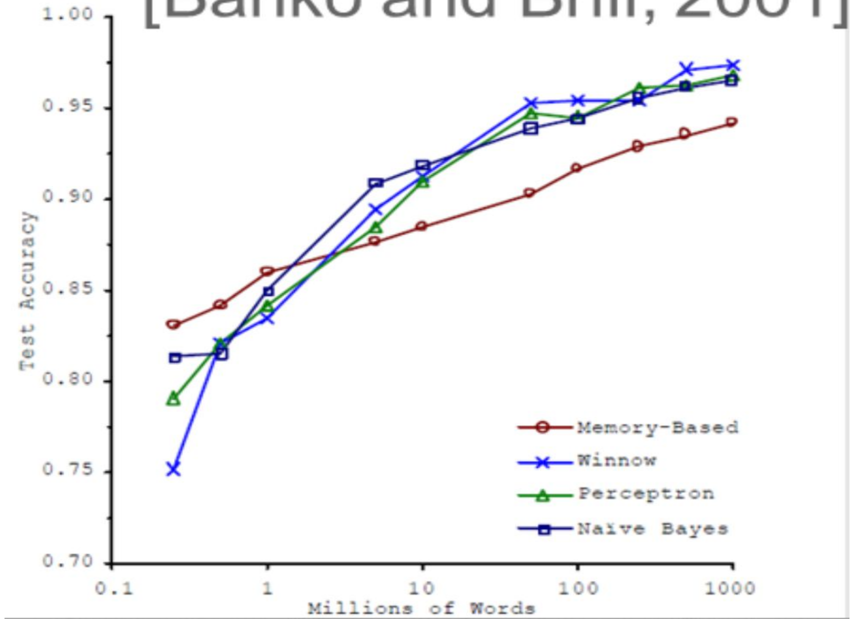
https://youtu.be/tIeHLnjs5U8

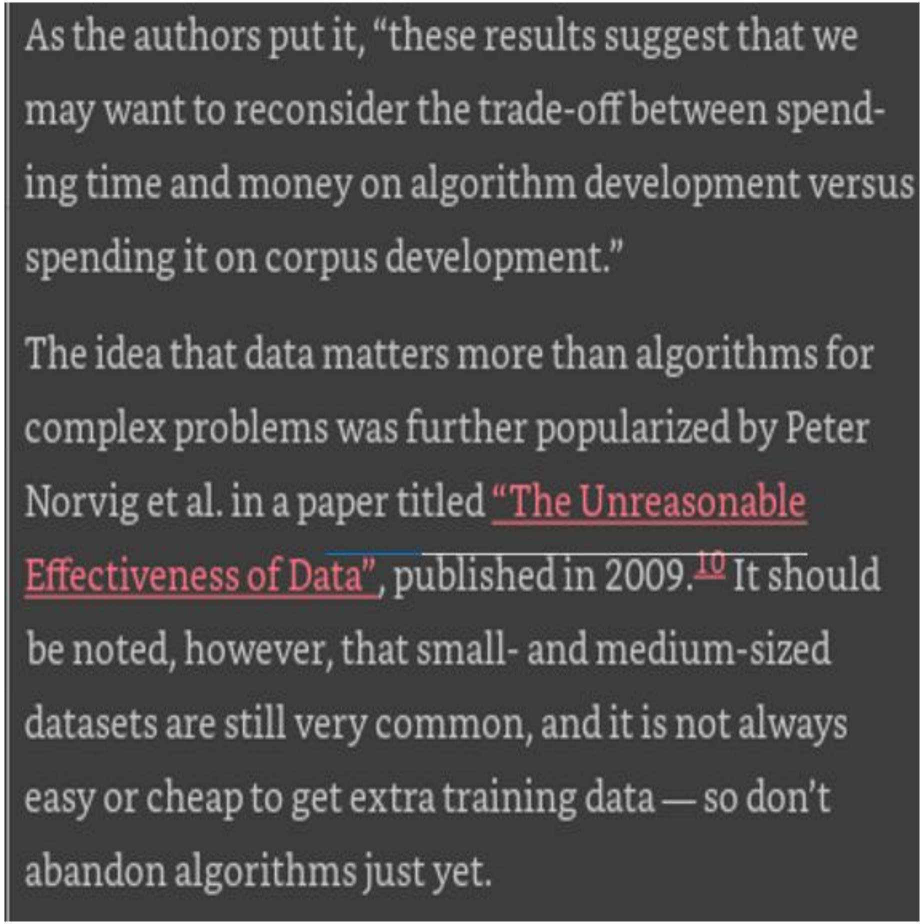


THE UNREASONABLE EFFECTIVENESS OF DATA

In a famous paper published in 2001, Microsoft researchers Michele Banko and Eric Brill showed that very different Machine Learning algorithms, including fairly simple ones, performed almost identically well on a complex problem of natural language disambiguation[8] once they were given enough data (as you can see in Figure 1-20).

[Banko and Brill, 2001]





As the authors put it, "these results suggest that we may want to reconsider the trade-off between spending time and money on algorithm development versus spending it on corpus development."

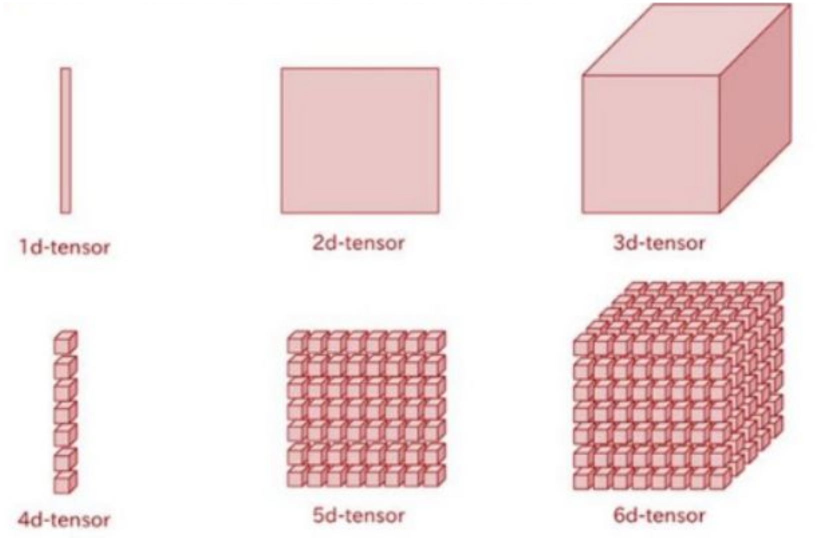
The idea that data matters more than algorithms for complex problems was further popularized by Peter Norvig et al. in a paper titled "The Unreasonable Effectiveness of Data", published in 2009.[10] It should be noted, however, that small- and medium-sized datasets are still very common, and it is not always easy or cheap to get extra training data — so don't abandon algorithms just yet.

http://technocalifornia.blogspot.com/2012/07/

What is a Tensor?

Generalization of a matrix with an arbitrary number of indices.

Tensors are simply **mathematical objects that can be used to describe physical properties, just like scalars and vectors**. In fact tensors are merely a generalisation of scalars and vectors; a scalar is a zero rank tensor, and a vector is a first rank tensor.



https://www.doitpoms.ac.uk/tlplib/tensors/what_is_tensor.php

What is TensorFlow?

- An end-to-end open source machine learning platform
- For research and production
- Distributed training and serving predictions
- Apache 2.0 license

CPU GPU TPU iOS JS Raspberry Pi

Why TensorFlow

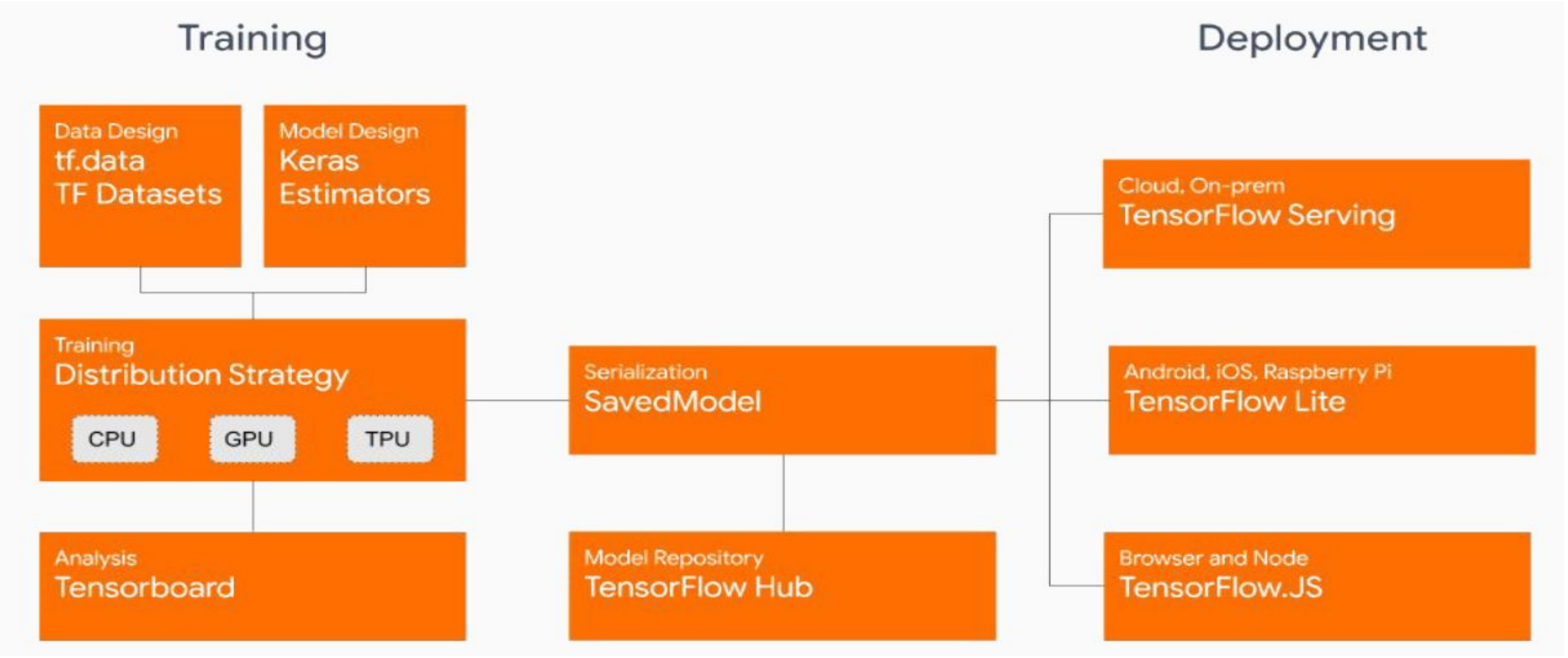
- Easy model building with Keras with eager execution

  : makes for immediate model interaction and easy debugging
- Robust ML production anywhere

  : easily train and deploy models in the cloud, on-premise, in the browser or on-device no matter what language you use.
- Powerful for research

  : a simple and flexible architecture to take new ideas from concept to code, to state-of-the-art models, and to publication faster.

TensorFlow core



TensorFlow Lite

- TensorFlow Lite is TensorFlow’s lightweight solution for mobile and embedded devices

- It enables on-device machine learning inference with low latency and a small binary size

- Low latency techniques: optimizing the kernels for mobile apps, pre-fused activations, and quantized kernels that allow smaller and faster (fixed-point math) models

- TensorFlow Lite also supports hardware acceleration with the Android Neural Networks API

https://www.tensorflow.org/mobile/tflite/

What does TensorFlow Lite contain?

- a set of core operators, both quantized and float, which have been tuned for mobile platforms
  - pre-fused activations and biases to further enhance performance and quantized accuracy
  - using custom operations in models also supported
- a model file format, based on FlatBuffers
  - the primary difference is that FlatBuffers does not need a parsing/unpacking step to a secondary representation before you can access data
  - the code footprint of FlatBuffers is an order of magnitude smaller than protocol buffers
- a mobile-optimized interpreter,
  - key goals: keeping apps lean and fast.
  - a static graph ordering and a custom (less-dynamic) memory allocator to ensure minimal load, initialization, and execution latency
- an interface to Android NN API if available

https://www.tensorflow.org/mobile/tflite/

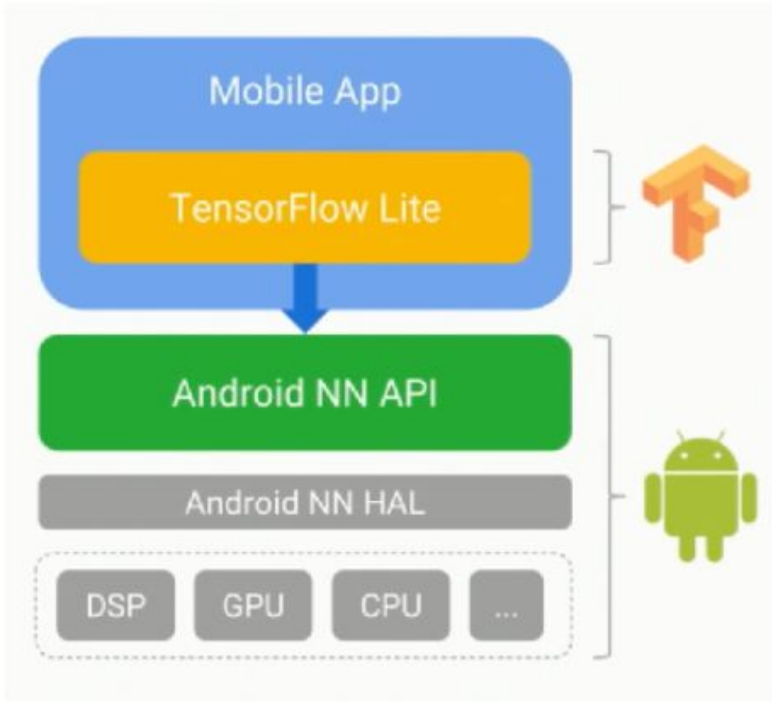
Why a new mobile-specific library?

- Innovation at the silicon layer is enabling new possibilities for hardware acceleration, and frameworks such as the Android Neural Networks API make it easy to leverage these
- Recent advances in real-time computer-vision and spoken language understanding have led to mobile-optimized benchmark models being open sourced (e.g. MobileNets, SqueezeNet)
- Widely-available smart appliances create new possibilities for on-device intelligence
- Interest in stronger user data privacy paradigms where user data does not need to leave the mobile device
- Ability to serve ‘offline’ use cases, where the device does not need to be connected to a network

https://www.tensorflow.org/mobile/tflite/

TesorFlow Lite and android NN in Google developer conf.

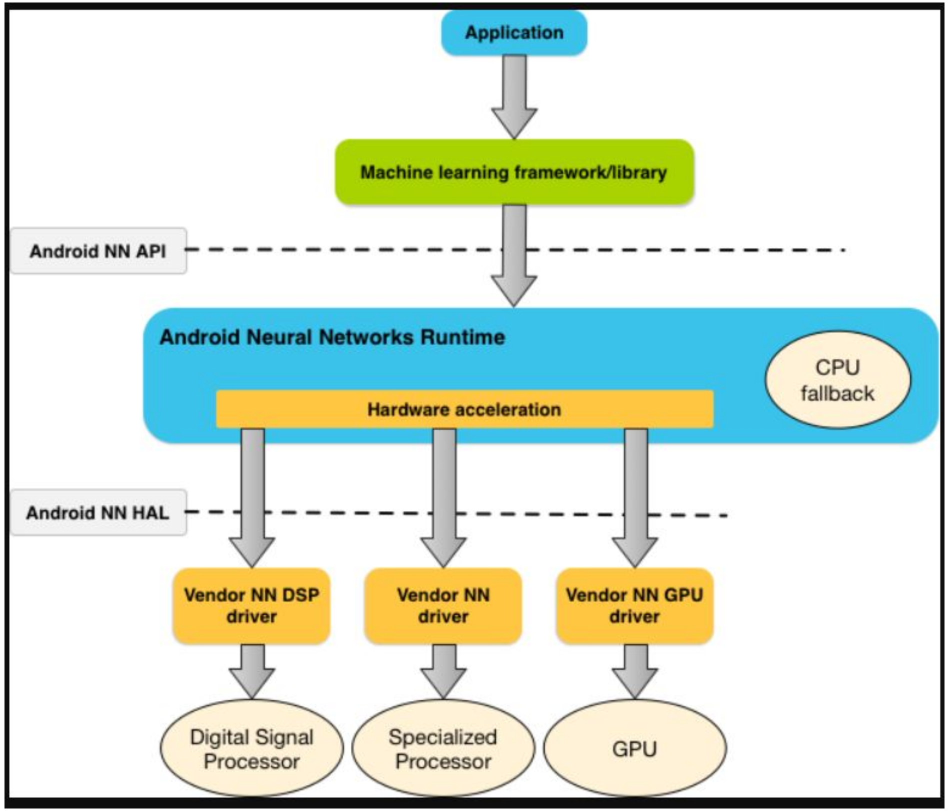
- TensorFlow runtime is get optimized for mobile and embedded applications
- Runs TensorFlow models on device
- Leverage Android NN API
- Released as open source



From Google I/O 2017 video

Actual Android NN API

- The Android Neural Networks API (NNAPI) is an Android C API designed for running computationally intensive operations for machine learning on mobile devices
- NNAPI is designed to provide a base layer of functionality for higher-level machine learning frameworks (such as TensorFlow Lite, Caffe2, or others) that build and train neural networks
- The API is available on all devices running Android 8.1 (API level 27) or higher.



https://developer.android.com/ndk/images/nnapi/nnapi_architecture.png

Example code of TFlite at a glance

- **model**: .tflite model
- **resolver**

: if no custom ops, builtin op resolve is enough

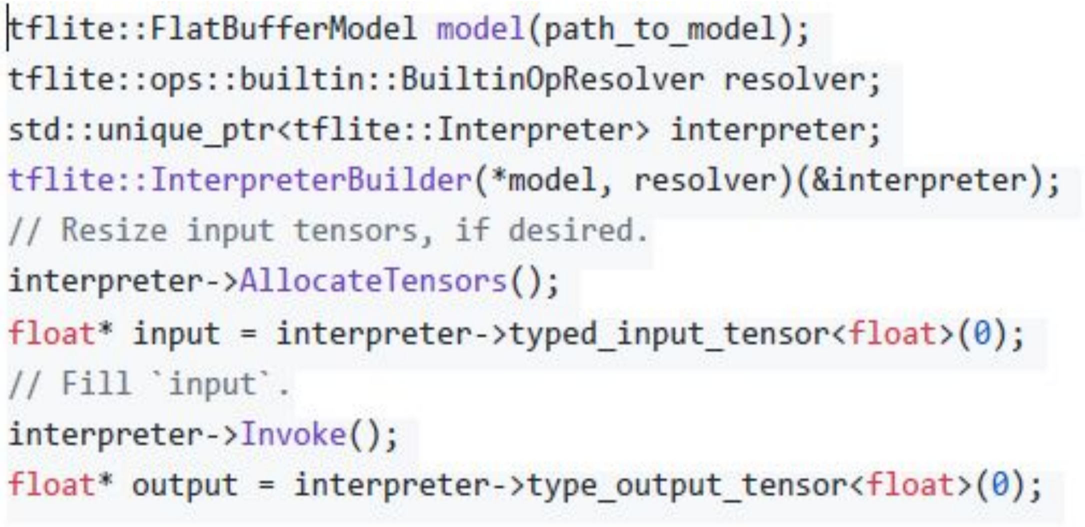
- **interpreter**

: need it to compute the graph

- interpreter->AllocateTensor()

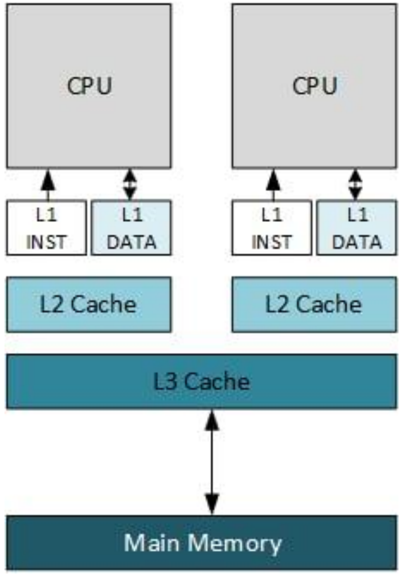
:allocate stuff for you, e.g., input tensor(s)

- fill the input
- interpreter->Invoke(): run the graph
- process the output



```
tflite::FlatBufferModel model(path_to_model);
tflite::ops::builtin::BuiltinOpResolver resolver;
std::unique_ptr<tflite::Interpreter> interpreter;
tflite::InterpreterBuilder(*model, resolver)(&interpreter);
// Resize input tensors, if desired.
interpreter->AllocateTensors();
float* input = interpreter->typed_input_tensor<float>(0);
// Fill `input`.
interpreter->Invoke();
float* output = interpreter->type_output_tensor<float>(0);
```

CPU for Deep Learning Inference



CPU

A typical Edge KWS application may have 2048 input layer, and 256 neurons in the 1st hidden layer – 500k MACs, 24 inferences/s

- Efficient code will require prefetching, small blocksizes for matrix multiplication to maximize the use of the local memory
  - Many access to larger caches and main memory required.
- Frameworks do allow compilation of the DNN model so it better fits into a traditional Von Neumann architecture.
- However, efficiency of 5-10% for inference on a CPU is common
- Could certainly do this on a RPi 4 or a Snapdragon…
- Can it be done at 140-600uW?

Question - Eric -

What is difference of TPU vs. DNN?

→ TPU is Google Tensor Processor Unit

DNN is deep neural network, a neural network with many hidden layers

Edge neural processor combines multiply accumulate units right next to memory (less capacitive and inductive load means smaller drivers, lower power required)

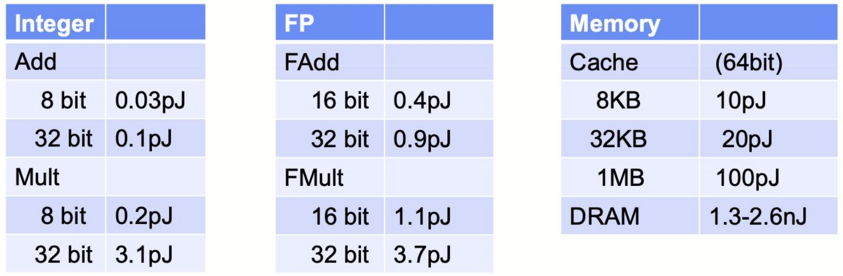
Computation flows from the input layer to the output in an array of processors with computation at the processors

Not good for training, only for inference. Just like TensorFlow lite, it's a deployment model but on HW

https://drive.google.com/file/d/1bOchLuPuyi7EUdODg16Y3jXW9I0d7Bib/view?usp=sharing
Seems Coral edge TPU or tensorflow lite also can do edge thing like DNN. is it correct? → It can, but it is comparable to the Jetson. It requires a framework to compile the model. It does not feature near/at memory compute…this is TPU ARCH: http://meseec.ce.rit.edu/551-projects/fall2017/3-4.pdf.
Look at results: https://mlcommons.org/en/inference-datacenter-20/
https://mlcommons.org/en/inference-edge-20/ QUALCOMM and NVIDIA kick Google's ass

Need For New Memory Architectures

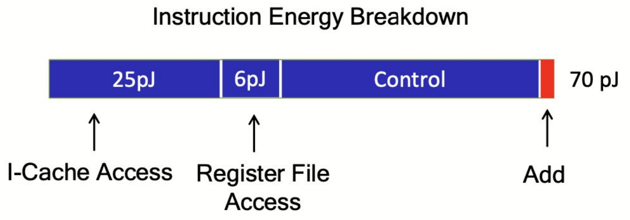
https://www.youtube.com/watch?v=JNQ7Eb5e7dc&t=90s



Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3.1pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1.1pJ
32 bit	3.7pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ



Computation

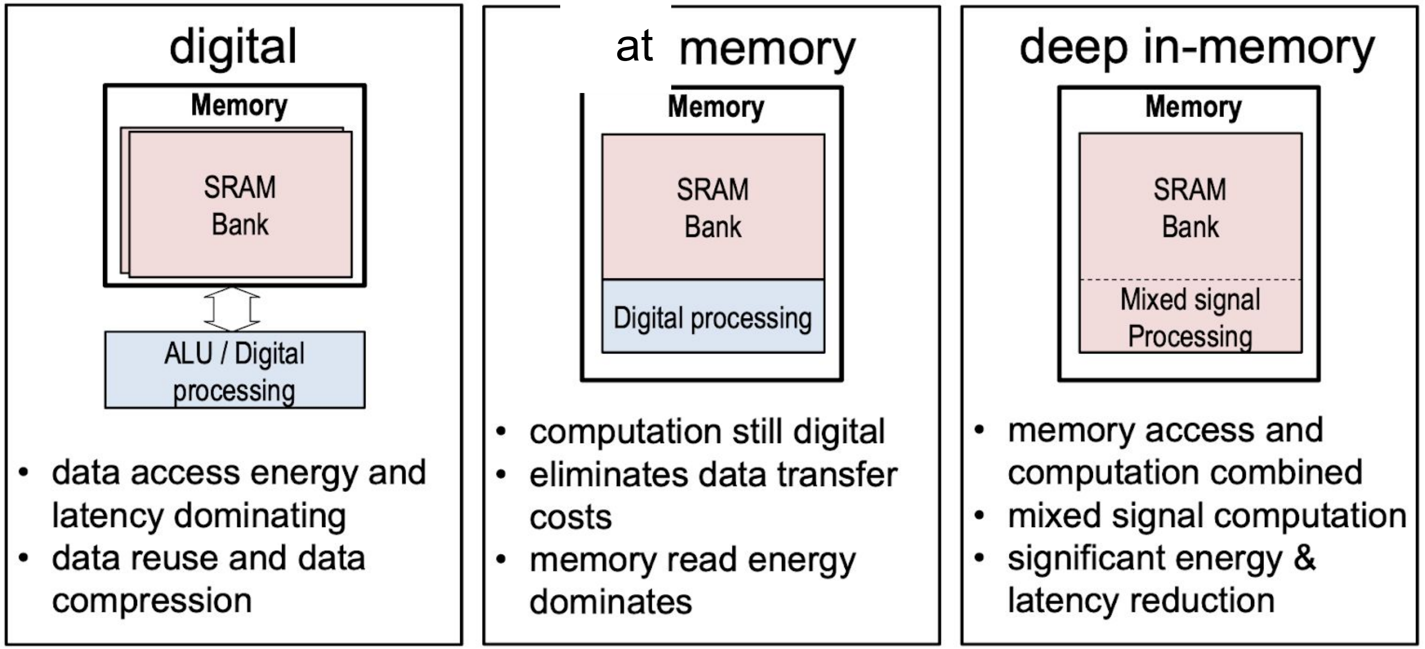
- Only a fraction of the energy required for memory accesses.
- putting memory as close to the the compute elements will save power!

Quantization

- Reduces computation energy consumption
- More importantly, reduces the memory access (and thus power) required for each parameter, or more parameters for same power

Mark Horwitz, Computing's Energy Problem (and what we can do about it) ISSCC 2014

At/Near Memory Compute



- data access energy and latency dominating
- data reuse and data compression

- computation still digital
- eliminates data transfer costs
- memory read energy dominates

- memory access and computation combined
- mixed signal computation
- significant energy & latency reduction

In-memory:
Analog
Variable results
Not directly scalable

near/at mem:
Digital
Repeatable
Scalable (for now)

From:
https://www3.nd.edu/~kogge/courses/cse40462-VLSI-fa18/www/Public/Lectures/compute-in-memory-architectures.pdf

At Memory Compute

Other Possibilities:

- GPU (Coral TPU, Jetson), DSP (Ethos, Greenwaves GAP9) or FPGA could be used to accelerate AI
- Another option is at or near memory compute architecture

Locate the compute (multiply accumulate) next to small segments of SRAM in a fabric allows more efficient processing.

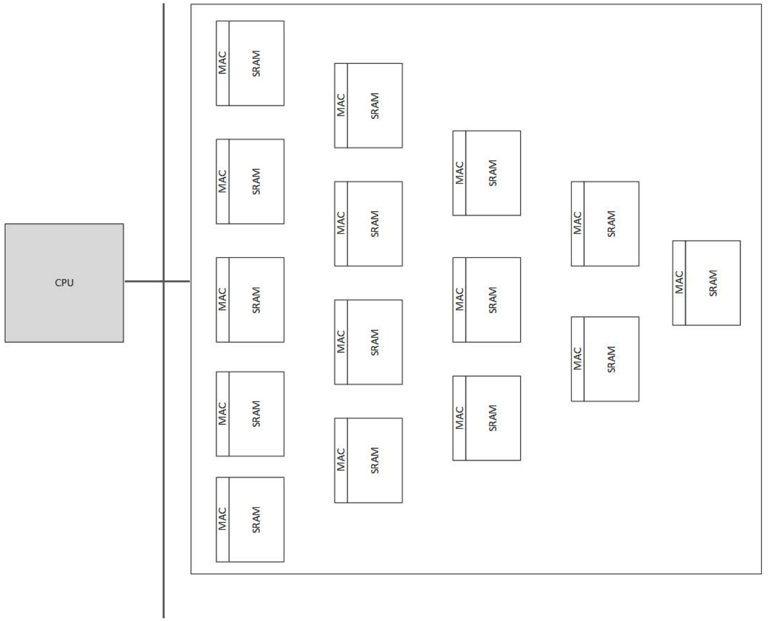
DNN weights stored in SRAM throughout the fabric

A CPU or DSP injects features into the input layer of the DNN.

Movement of data is minimized

Because the memory sizes are small and adjacent to compute, **< 1nJ** is required for each MAC.

Higher Efficiency than CPUs/DSPs for this workload:: 80+% vs 5%-15%

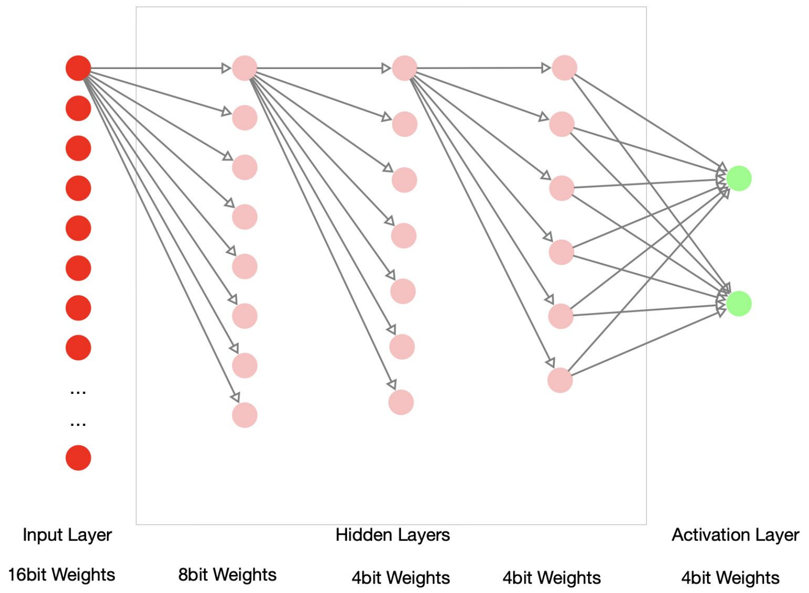


At-memory coprocessor

In Memory Compute

- Utilize a Non-volatile memory (typically Flash) with ADCs to do computations in the memory itself
- A lot of papers written on this topic!
- Mythic.AI founded a company based on this idea
- Syntiant tried two in-memory test chips and found that it was not productizable.
    - Analog variation requires calibration circuits which offset power and area advanages
    - dense layers map well, other layer types may not work at all (e.g. attention layers)
    -

Quantization



At-Memory Compute architecture allows variable precision quantization for different layers

More bits at the front, and highly quantized 4bit, 2bit, or even 1 bit weights at the furthest layers of the network.

More quantization means less movement of bits through the network.

Feature Extraction

Neural Networks need the number of features to be reduced, otherwise the number of parameters would be enormous! More expressive power for “free”.

Feature extraction is a way to increase its ‘trainability’ — it basically forces a physical structure into the DNN, giving a higher level representation of the input.

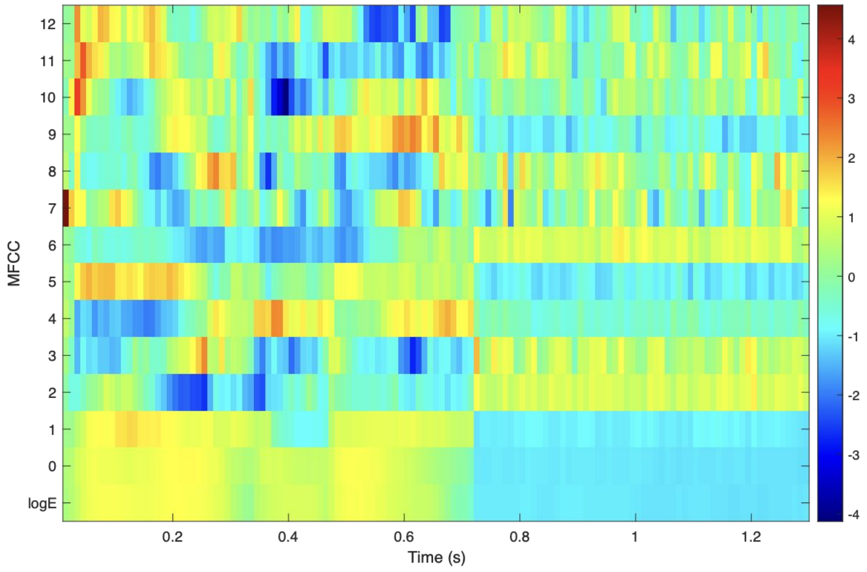
Sobel filter from our HW is a good example of feature extraction for vision

For Voice KWS,, two approaches are commonly used:

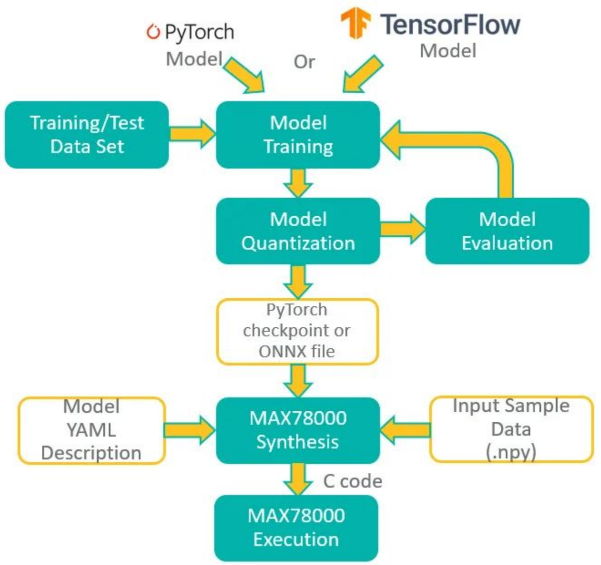
1. A Convolutional NN is used to pre-process and extract features from audio [1]
2. MFCC (Mel-frequency cepstral coefficient). A classic [2]

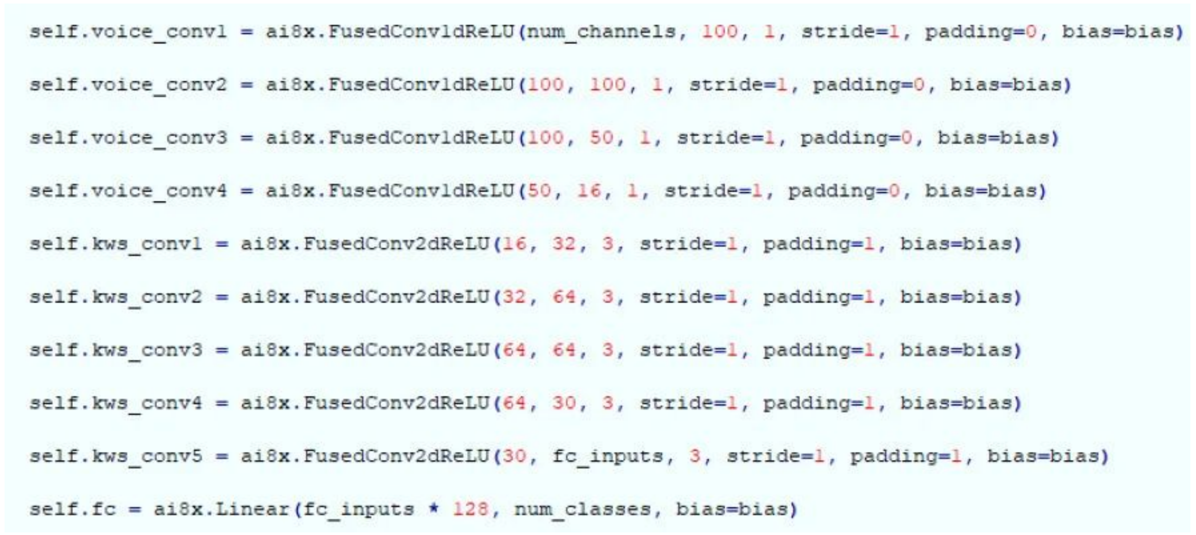
[1] https://www.maximintegrated.com/en/design/technical-documents/app-notes/7/7359.html
[2] M. J. Hunt, M. Lennig, and P. Mermelstein, “Experiments in syllable-based recognition of continuous speech,” Proceedings of the 1980 ICASSP, Denver, CO, pp. 880-883, 1980

Feature Input to DNN



Training / Deployment with at Memory Compute Accelerator





```
self.voice_conv1 = ai8x.FusedConv1dReLU(num_channels, 100, 1, stride=1, padding=0, bias=bias)

self.voice_conv2 = ai8x.FusedConv1dReLU(100, 100, 1, stride=1, padding=0, bias=bias)

self.voice_conv3 = ai8x.FusedConv1dReLU(100, 50, 1, stride=1, padding=0, bias=bias)

self.voice_conv4 = ai8x.FusedConv1dReLU(50, 16, 1, stride=1, padding=0, bias=bias)

self.kws_conv1 = ai8x.FusedConv2dReLU(16, 32, 3, stride=1, padding=1, bias=bias)

self.kws_conv2 = ai8x.FusedConv2dReLU(32, 64, 3, stride=1, padding=1, bias=bias)

self.kws_conv3 = ai8x.FusedConv2dReLU(64, 64, 3, stride=1, padding=1, bias=bias)

self.kws_conv4 = ai8x.FusedConv2dReLU(64, 30, 3, stride=1, padding=1, bias=bias)

self.kws_conv5 = ai8x.FusedConv2dReLU(30, fc_inputs, 3, stride=1, padding=1, bias=bias)

self.fc = ai8x.Linear(fc_inputs * 128, num_classes, bias=bias)
```

https://www.maximintegrated.com/en/design/technical-documents/app-notes/7/7359.html

TinyMLperf – A Benchmark for Edge Compute

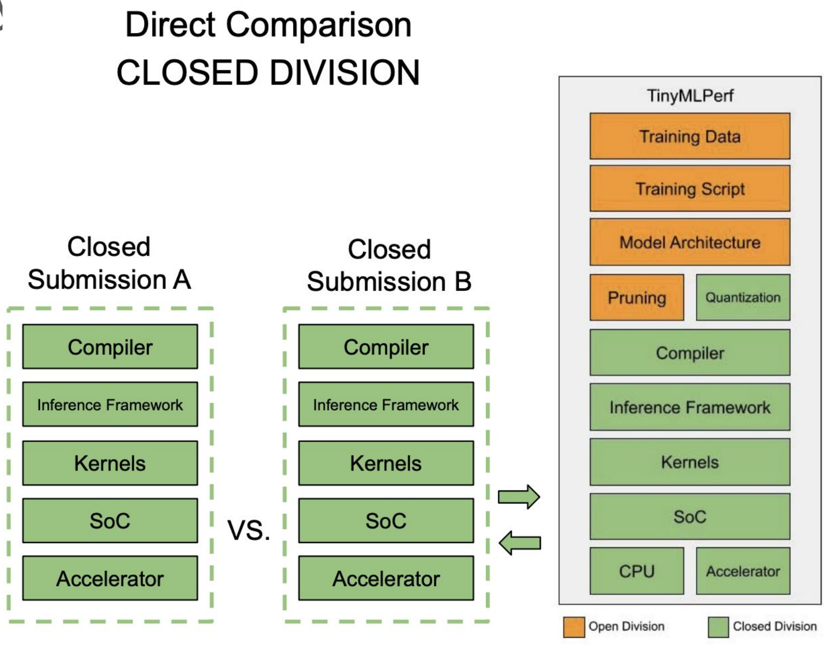
A benchmark for comparing different edge inference solutions [paper],

measuring Accuracy, Latency, and Energy

A vector of features are fed in directly to computing unit (CPU/NPU) with a reference model (52K parameter CNN in KWS case)

Output classification is measured against reference input, latency is measured by start/stop

Compare at-memory compute vs. CPU-based inference



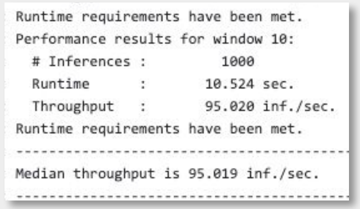
Measuring Benchmark Metrics (TinyMLperf)

Latency

Data loading + warmup excluded

Evaluated on loop of inferences for accuracy

Data-dependent execution path?


```
Runtime requirements have been met.
Performance results for window 10:
  # Inferences :         1000
  Runtime      :         10.524 sec.
  Throughput   :         95.020 inf./sec.
Runtime requirements have been met.
-----------------------------------------
Median throughput is 95.019 inf./sec.
-----------------------------------------
```

Accuracy

Evaluate on larger dataset

Top-1 accuracy & **AUC**

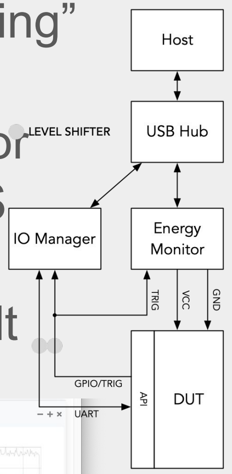
CLOSED: meet threshold
vs.
OPEN: part of the metrics

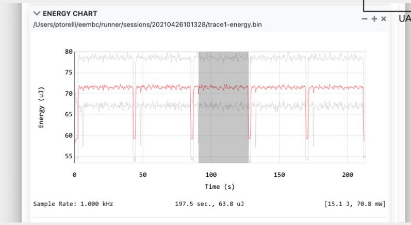
Energy

No “cherry-picking”

Power Monitor setup <100$

Median result





```
Sample Rate: 1.000 kHz          197.5 sec., 63.8 uJ          [15.1 J, 70.8 mW]
```

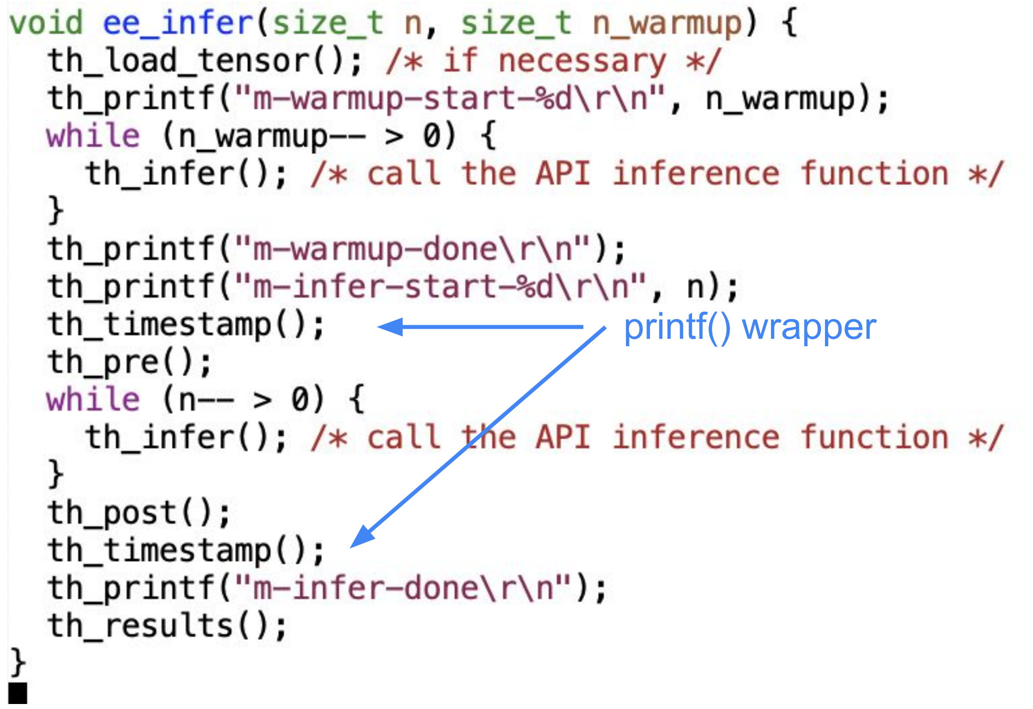
b.

Latency / Throughput (how it's measured in TinyML Perf)

Use MCU internal timer, wrapped around a loop of inferences.

Load tensor and run a few warmup cycles outside of timing.

Run >10s of inferences to amoritize timestamps.



```
void ee_infer(size_t n, size_t n_warmup) {
  th_load_tensor(); /* if necessary */
  th_printf("m-warmup-start-%d\r\n", n_warmup);
  while (n_warmup-- > 0) {
    th_infer(); /* call the API inference function */
  }
  th_printf("m-warmup-done\r\n");
  th_printf("m-infer-start-%d\r\n", n);
  th_timestamp();
  th_pre();
  while (n-- > 0) {
    th_infer(); /* call the API inference function */
  }
  th_post();
  th_timestamp();
  th_printf("m-infer-done\r\n");
  th_results();
}
```

https://mlcommons.org/en/inference-tiny-07/

						Keyword Spotting	
						Google Speech Commands	
						DSCNN	
						90% (top 1)	
Submitter	**System**	**Device**	**Processor(s)**	**Accelerator(s)**	**Software**	Latency in ms	Energy in uJ
Andes	ADP-XC7K160/410 FPGA	AE350 Platform SoC with AndesCore™ processor	AndesCore™ D25F, 5-stage, single-issue, RV32 IMACP (1)	RISC-V P Extension	TensorFlowLite for Microcontrollers, Andes NN Library	79.85	
Andes	ADP-XC7K160/410 FPGA	AE350 Platform SoC with AndesCore™ processor	AndesCore™ D45, 8-stage, dual-issue, RV32 IMACP (1)	RISC-V P Extension	TensorFlowLite for Microcontrollers, Andes NN Library	68.31	
Andes	Xilinx VCU118 FPGA	AE350 Platform SoC with AndesCore™ processor	AndesCore™ NX27V, 5-stage, single-issue, RV64 IMACV, RVV(VLEN, SIMD, BIU)=(128, 128, 128) (1)	RISC-V V Extension	TensorFlowLite for Microcontrollers, Andes NN Library		
Plumerai	NUCLEO-L4R5ZI	STM32L4R5ZIT6U	Arm® Cortex®-M4		Plumerai inference engine	73.5	
Plumerai	CY8CPROTO_062_4343w	PSoC 62 MCU	Arm® Cortex®-M4		Plumerai inference engine	63.6	
Plumerai	DISCO-F746NG	STM32F746NGH6	Arm® Cortex®-M7		Plumerai inference engine	19.5	
Renesas	EK-RA6M4	RA6M4	Arm Cortex-M33 w/FPU(1)		TensorFlowLite for Microcontrollers	50.57	3796.96
Renesas	RX65N-Cloud-Kit	RX65N	Renesas RXv2		TensorFlowLite for Microcontrollers	81.85	4422.68
STMicroelectronics	NUCLEO-L4R5ZI	STM32L4R5ZIT6U	Arm® Cortex®-M4		X-CUBE-AI v7.1.0	97.65	4635.68
STMicroelectronics	NUCLEO-U575ZI-Q	STM32U575ZIT6Q	Arm® Cortex			54.81	1482.4
STMicroelectronics	NUCLEO-H7A3ZI-Q	STM32H7A3ZIT6Q	ARM® Cortex®-M7		X-CUBE-AI v7.1.0	22.29	3713.19
Syntiant	syntiant_9120_1v1_98mhz	NDP120	M0 + HiFi	Syntiant Core 2	Syntiant TDK	1.8	49.59
Syntiant	syntiant_9120_0v9_30mhz	NDP120	M0 + HiFi	Syntiant Core 2	Syntiant TDK	4.3	35.29
Silicon Labs	xG24-DK2601B	EFR32MG24	Cortex-M33(1)	Silicon Labs MVP(1)	TensorFlowLite for Microcontrollers, CMSIS-NN, Silicon Labs Gecko SDK	63.09	611.49

At memory compute: 28x less latency + 76x less energy

Pros and Cons – At Memory Compute

At Memory Compute Neural Accelerator

Pros:

- Less latency
- Much Lower power during inferencing
- Easy to fit model into embedded application –no compilation
- Easier to apply heterogenous quantization in model
- More efficient than CPU, CPU w/SIMD or DSP

Cons:

- High leakage (large SRAM array)
  - High power even when idling
- Large Die Area (for memory)
- Size of DNN limited by size of on-chip SRAM
- Smaller process nodes have even higher leakage – balance for leakage and size
- Doesn’t take advantage of sparsity of tensor

Future Directions – Near/At/In-Memory Compute

- Replace SRAM with a fast non-volatile memory such as Resistive RAM
  - higher density
  - Less leakage
- Mix in-memory compute for some dense layers with at-memory compute layers… take advantage of both

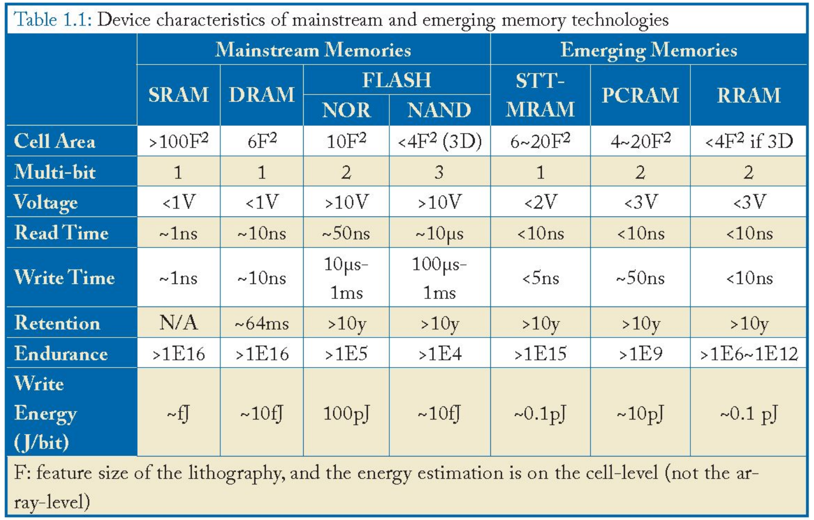


Table 1.1: Device characteristics of mainstream and emerging memory technologies

	Mainstream Memories				Emerging Memories		
			FLASH				
	SRAM	DRAM	NOR	NAND	STT-MRAM	PCRAM	RRAM
Cell Area	>100F^2	6F^2	10F^2	<4F^2 (3D)	6~20F^2	4~20F^2	<4F^2 if 3D
Multi-bit	1	1	2	3	1	2	2
Voltage	<1V	<1V	>10V	>10V	<2V	<3V	<3V
Read Time	~1ns	~10ns	~50ns	~10µs	<10ns	<10ns	<10ns
Write Time	~1ns	~10ns	10µs–1ms	100µs–1ms	<5ns	~50ns	<10ns
Retention	N/A	~64ms	>10y	>10y	>10y	>10y	>10y
Endurance	>1E16	>1E16	>1E5	>1E4	>1E15	>1E9	>1E6~1E12
Write Energy (J/bit)	~fJ	~10fJ	100pJ	~10fJ	~0.1pJ	~10pJ	~0.1 pJ

F: feature size of the lithography, and the energy estimation is on the cell-level (not the array-level)

From: Resistive Random Access Memory (RRAM), 2016, Shimeng Yu

Conclusion

TensorFlow Lite framework on an embedded CPUs/DSPs, as well as dedicated neural processors can both be used for edge inference tasks.

- DNNs are trained in the cloud using frameworks like TensorFlow which use backpropagation for training.
- Edge DNNs can run on microcontrollers using TensorFlow Lite or dedicated neural processing hardware
- Benchmark shows at-memory neural processors outperform CPUs and DSPs running TensorFlow Lite in terms of latency and inference energy
    - Can save energy or use a much larger network for better performance
- TensorFlow Lite on embedded CPUs do not require additional hardware
    - if edge inference is infrequent, or the model is not big, may meet system requirements
    - May have less leakage than dedicated neural processor –