tinyML® Research Symposium

Enabling Ultra-low Power Machine Learning at the Edge

March 27, 2023

www.tinyML.org

Training Neural Networks for Execution on Approximate Hardware

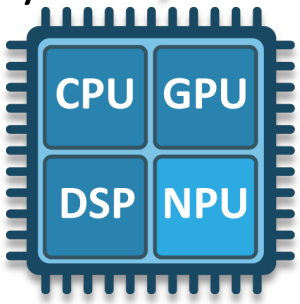
SPONSORED BY: DARPA, ONR

TIANMU LI, SHURUI LI, PUNEET GUPTA

Limits of Edge Machine Learning

Energy
Latency
Privacy

Edge Inference

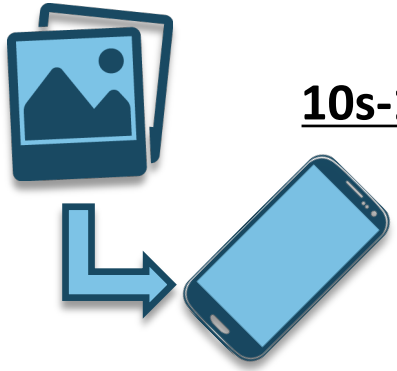


High Model Complexity

10s-100s of MBs

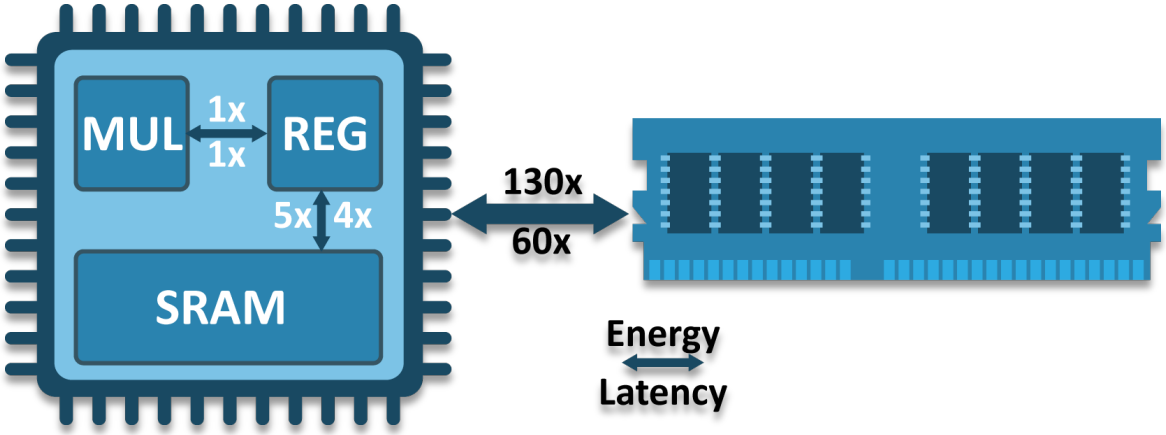
1s-10s GOPS

Non-Real Time Performance



10s-100s of ms

Memory constraints energy and latency



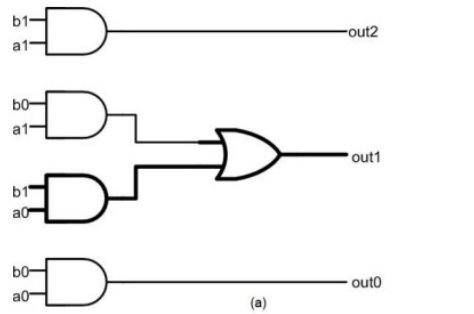
Possible Solution

Approximate Computing

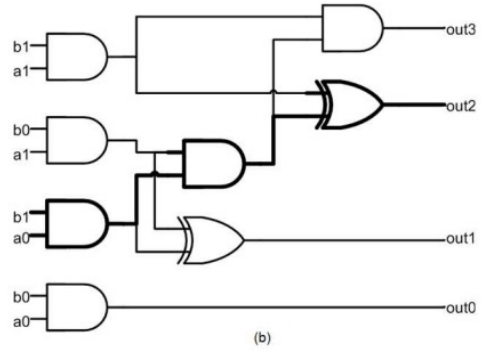
Images: Freepik.com

Benefits of Approximate Computing

- Approximate computing methods trade some accuracy for improved performance
- Approximate multiplication
  - Introduce error for specific input combinations
  - Simplified logic design
- Analog computing
  - Low-cost multiply-accumulate
  - Reduced memory movements
- Stochastic computing
  - Randomized bit streams
  - Single-gate multiplication and addition

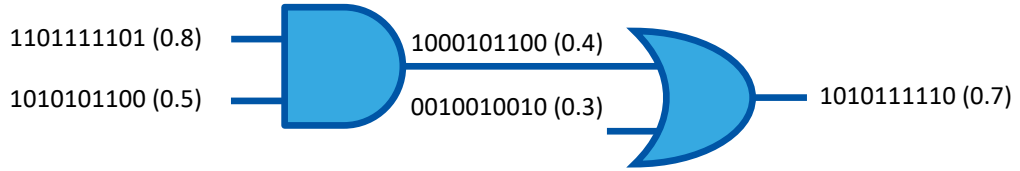


Approximate



Accurate

[1]



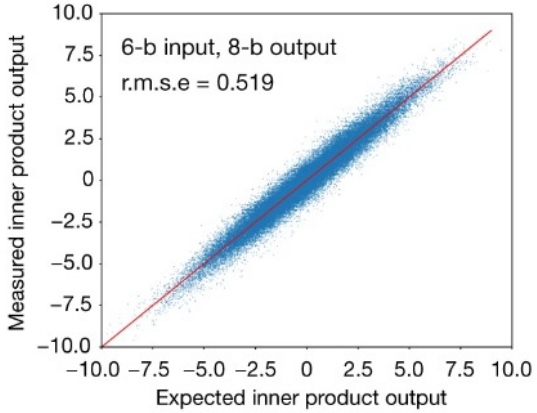
[1] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture," in *Proc. IEEE/ACM International Conference on VLSI Design*, 2011

Training for Approximate Computing

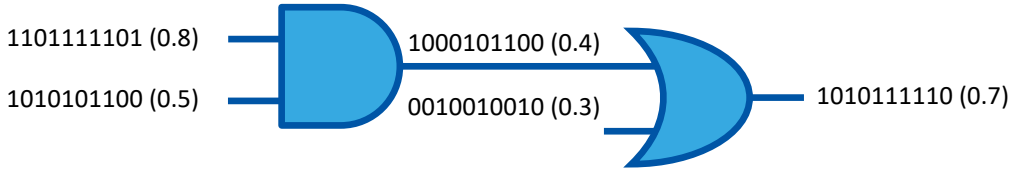
- Approximate computing introduces error
- Approximate multiplication
  - Error for certain input combinations
  - Uneven error curve
- Analog computing
  - Physical variations
  - Analog-to-digital converter limits accumulation range and precision
- Stochastic computing
  - Random bit stream error
  - Non-linear accumulation

A_1A_0 \ B_1B_0	00	01	11	10
00	000	000	000	000
01	000	001	011	010
11	000	011	111	110
10	000	010	110	100

[1]



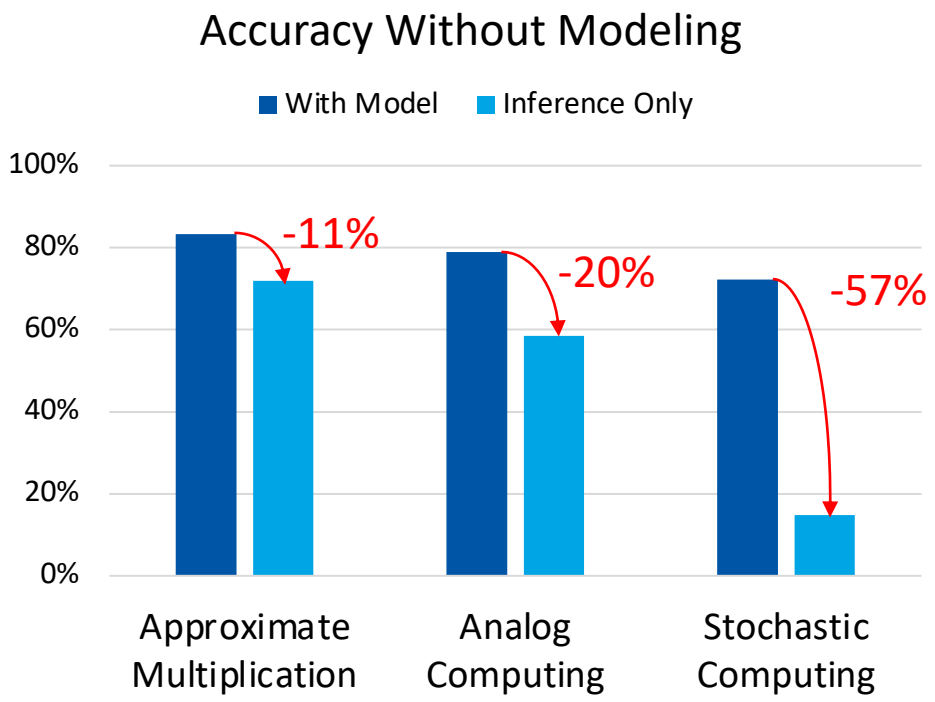
[2]



[1] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture," in *Proc. IEEE/ACM International Conference on VLSI Design*, 2011
[2] Wan, W., Kubendran, R., Schaefer, C. et al., "A compute-in-memory chip based on resistive random-access memory", *Nature 608, 504–512,* 2022

Training for Approximate Computing

- Approximate computing errors are not modeled in floating/fixed-point training
  - Inaccurate representation
  - Random multiplication error
  - Non-linear addition
- Approximate computation needs to be modeled during training
  - Inferencing using a fixed-point model reduces accuracy
  - Approximate computing needs to be modeled in **both** forward and backward pass
  - Not modeling approximate computing drops accuracy by **11-57%**



Forward Pass Modeling

- Accurate modeling of approximate computing is expensive
  - CPU/GPUs do not have native operators for approximate computation
  - Requires software simulation

Emulation Cost Estimation

	Multiplication	Addition
Floating Point	0.5 (fused)	0.5 (fused)
Stochastic Computing	64 (unrolled) 2 (packed)	64 (unrolled) 2 (packed)
Approximate Multiplication	86	1
Analog Computing	1	9

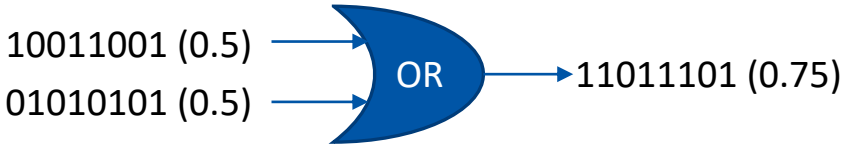
```
int mult_evo(int a, int b) {
  int wa[7];
  int wb[7];
  int y = 0;
  wa[0] = (a >> 0) & 0x01;
  wb[0] = (b >> 0) & 0x01;
  wa[1] = (a >> 1) & 0x01;
  wb[1] = (b >> 1) & 0x01;
  wa[2] = (a >> 2) & 0x01;
  wb[2] = (b >> 2) & 0x01;
  wa[3] = (a >> 3) & 0x01;
  wb[3] = (b >> 3) & 0x01;
  wa[4] = (a >> 4) & 0x01;
  wb[4] = (b >> 4) & 0x01;
  wa[5] = (a >> 5) & 0x01;
  wb[5] = (b >> 5) & 0x01;
  wa[6] = (a >> 6) & 0x01;
  wb[6] = (b >> 6) & 0x01;
  int sig_83 = wa[6] & wb[3];
  int sig_113 = wa[6] & wb[2];
  int sig_119 = wa[5] & wb[4];
  int sig_120 = wa[
  int sig_144 = w
  int sig_145 = wb[5] & wa[4];
  int sig_146 = sig_83 ^ sig_119;
  int sig_147 = sig_83 & sig_119;
  int sig_148 = sig_146 & sig_113;
  int sig_149 = sig_146 ^ sig_113;
  int sig_150 = sig_147 ^ sig_148;
  int sig_155 = wa[4] & wb[5];
  int sig_156 = wa[5] & wb[5];
  int sig_157 = wa[6] & wb[5];
  int sig_172 = wb[6] & wa[2];
  int sig_173 = sig_144 ^ wa[3];
  int sig_174 = sig_144 & wb[4];
  int sig_175 = sig_173 & wb[5];
  int sig_177 = sig_174 | sig_175;
  int sig_178 = sig_149 ^ sig_155;
  int sig_179 = wa[4] & wb[5];
  int sig_181 = sig_178 ^ sig_145;
  int sig_182 = sig_179;
  int sig_183 = sig_120 ^ sig_156;
  int sig_184 = sig_120 & sig_156;
  int sig_185 = sig_183 & sig_150;
  int sig_186 = sig_183 ^ sig_150;
  int sig_187 = sig_184 ^ sig_185;
  int sig_191 = wa[3] & wb[6];
  int sig_192 = wa[4] & wb[6];
  int sig_193 = wa[5] & wb[6];
  int sig_194 = wa[6] & wb[6];
  int sig_205 = wa[1] & wb[6];
  int sig_206 = wa[1] & wa[2];
  int sig_208 = sig_205 ^ sig_172;
  int sig_209 = sig_206 & wb[6];
  int sig_210 = sig_181 ^ sig_191;
  int sig_211 = sig_181 & sig_191;
  int sig_212 = sig_210 & sig_177;
  int sig_213 = sig_210 ^ sig_177;
  int sig_214 = sig_211 | sig_212;
  int sig_215 = sig_186 ^ sig_192;
  int sig_216 = sig_186 & sig_192;
  int sig_217 = sig_215 & sig_182;
  int sig_218 = sig_215 ^ sig_182;
  int sig_219 = sig_216 | sig_217;
  int sig_220 = sig_157 ^ sig_193;
  int sig_221 = sig_157 & sig_193;
  int sig_222 = sig_220 & sig_187;
  int sig_223 = sig_220 ^ sig_187;
  int sig_224 = sig_221 ^ sig_222;
  int sig_231 = wb[3] & wa[5];
  int sig_232 = sig_213 ^ sig_209;
  int sig_233 = sig_213 & sig_209;
  int sig_234 = sig_232 & sig_231;
  int sig_235 = sig_232 ^ sig_231;
  int sig_236 = sig_233 | sig_234;
  int sig_237 = sig_218 ^ sig_214;
  int sig_238 = sig_218 & sig_214;
  int sig_239 = sig_237 & sig_236;
  int sig_240 = sig_237 ^ sig_236;
  int sig_241 = sig_238 ^ sig_239;
  int sig_242 = sig_223 ^ sig_219;
  int sig_243 = sig_223 & sig_219;
  int sig_244 = sig_242 & sig_241;
  int sig_245 = sig_242 ^ sig_241;
  int sig_246 = sig_243 | sig_244;
  int sig_247 = sig_194 ^ sig_224;
  int sig_248 = wb[6] & sig_224;
  int sig_249 = sig_247 & sig_246;
  int sig_250 = sig_247 ^ sig_246;
  int sig_251 = sig_248 ^ sig_249;
  y |= (sig_175 & 0x01) << 0; // default output
  y |= (sig_237 & 0x01) << 1; // default output
  y |= (sig_194 & 0x01) << 2; // default output
  y |= (sig_224 & 0x01) << 3; // default output
  y |= (sig_205 & 0x01) << 4; // default output
  y |= (sig_205 & 0x01) << 5; // default output
  y |= (sig_120 & 0x01) << 6; // default output
  y |= (sig_175 & 0x01) << 7; // default output
  y |= (sig_208 & 0x01) << 8; // default output
  y |= (sig_235 & 0x01) << 9; // default output
  y |= (sig_240 & 0x01) << 10; // default output
  y |= (sig_245 & 0x01) << 11; // default output
  y |= (sig_250 & 0x01) << 12; // default output
  y |= (sig_251 & 0x01) << 13; // default output
  return y;
}
```

>86 instructions!

Approximate Multiplier Simulation

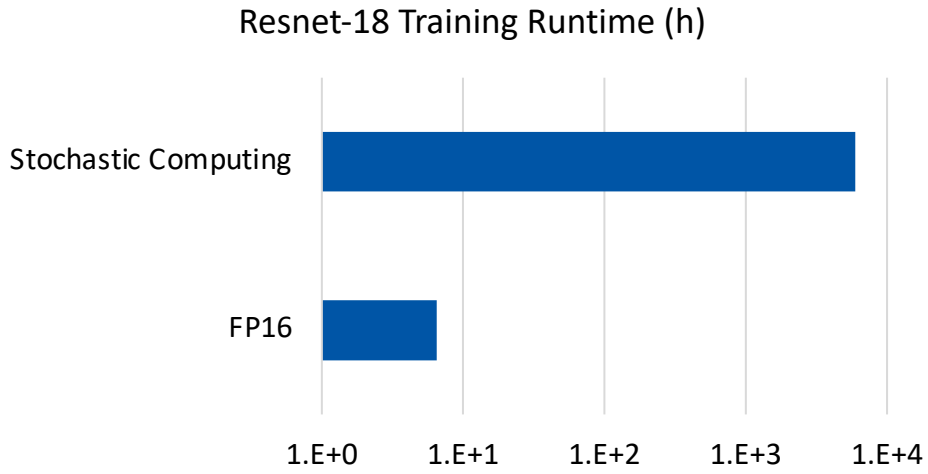
Backward Pass Modeling

- Some approximate computation methods introduce non-linearities in dot products
  - Stochastic OR adder performs $a + b - ab$
  - Analog computing is limited by the range of analog-to-digital conversion
- Modeling non-linear accumulation is expensive
  - Gradients w.r.t to addition need to be computed
  - Additions need to be broken up to model accurately
  - Breaking up additions increase runtime by **>100X**



$$f(a,b) = a + b \quad \Rightarrow \quad \frac{df}{da} = 1$$

$$f(a,b) = a + b - ab \quad \Rightarrow \quad \frac{df}{da} = 1 - b$$

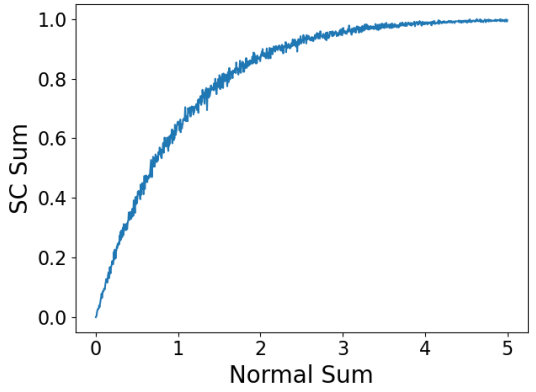


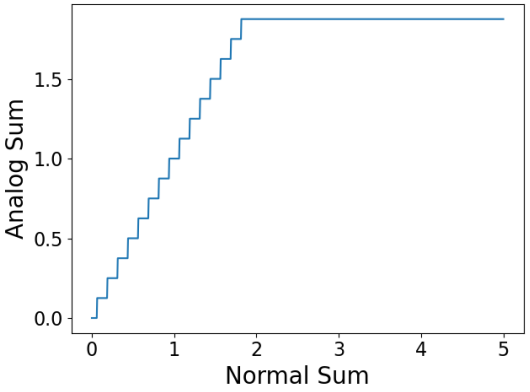
Overview

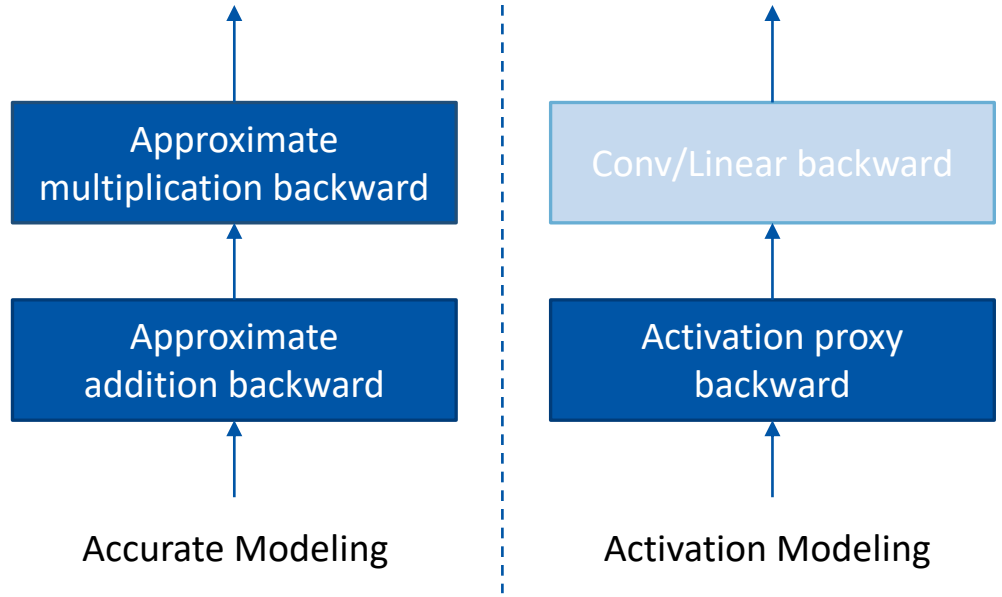
- Methodology
  - Backward pass - approximate proxy activation
  - Forward pass - error injection + fine tuning
  - Memory management - gradient checkpointing
- Results
  - Accuracy impact
  - Runtime impact

Backward: Activation Proxy Modeling

- Modeling approximate computing in the backward pass is expensive
  - Error profile is not a smooth function
  - Modeling non-linearity requires breaking up additions
- Small gradient error does not impact convergence
- Approximate non-linearity using an activation function
  - Cheap to implement (point-wise function)
  - Allows usage of optimized backward functions

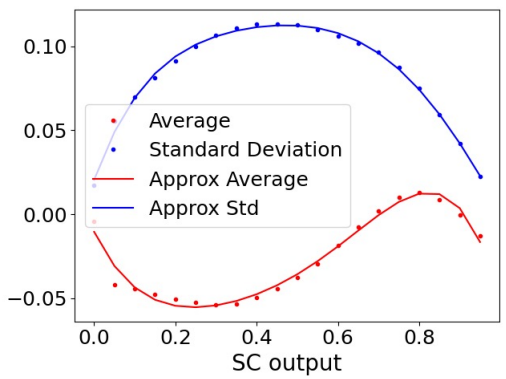


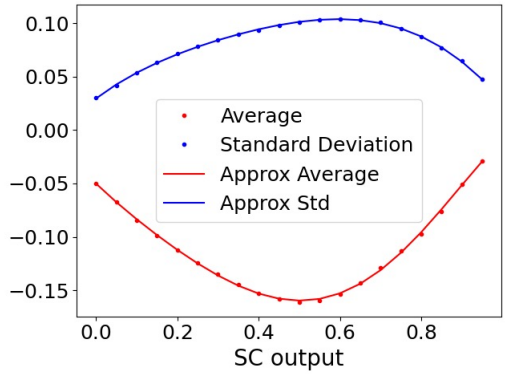


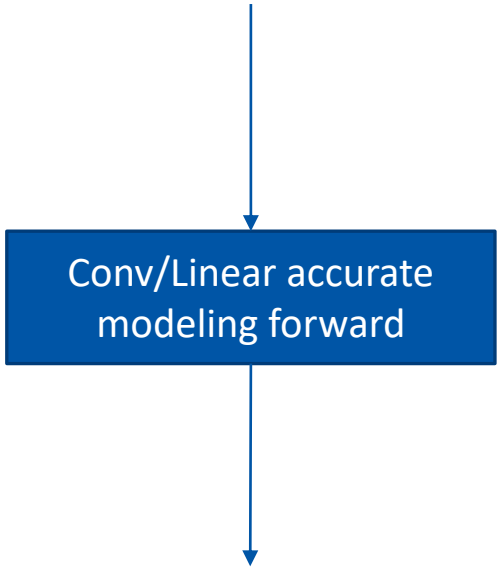


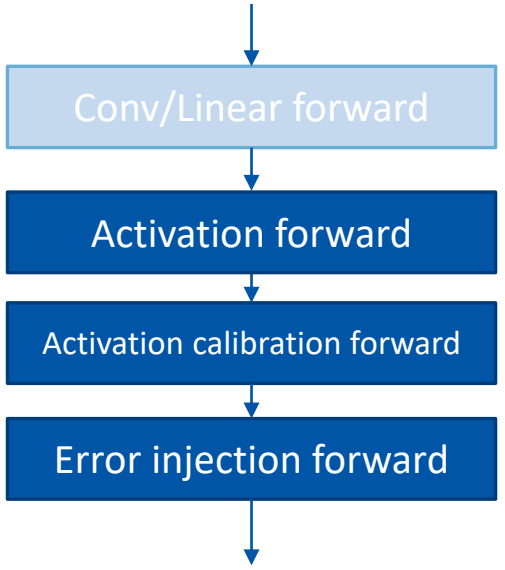
Forward: Error Injection

- Output error is a function of output value
  - Average error: modified activation function
  - Random error: error injection
- Replace accurate modeling with error injection
  - Fit error profile to polynomial functions
  - Calibrate error profile during training



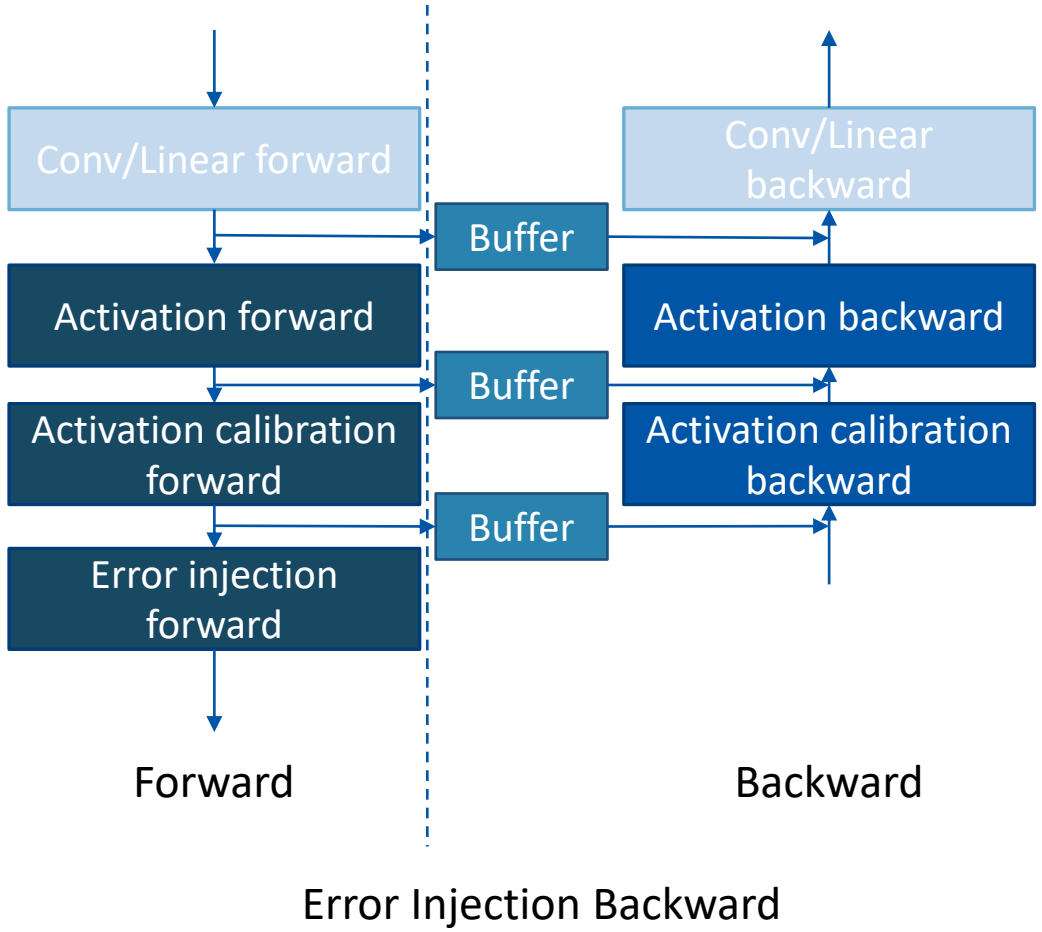






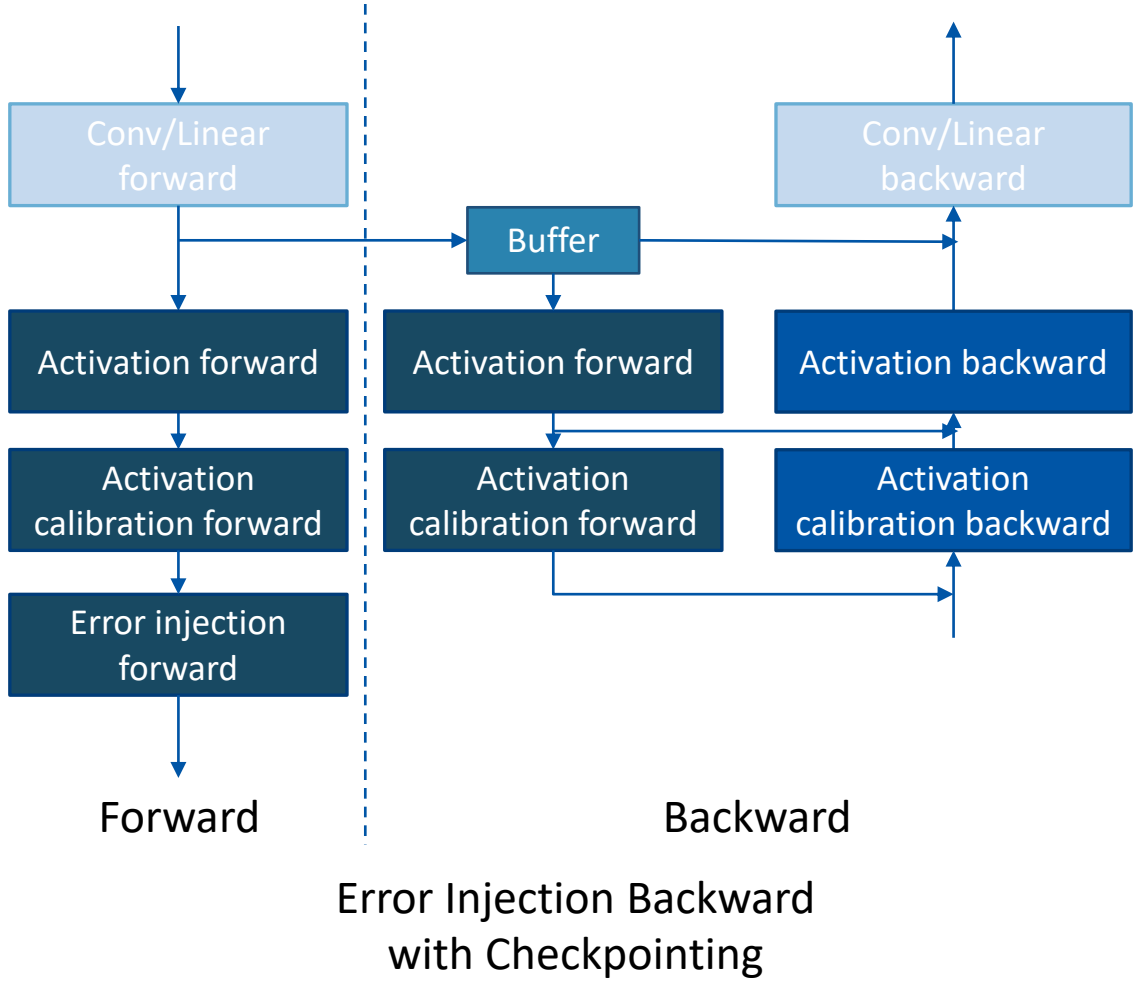
Gradient Checkpointing

- Activation proxy and activation calibration add computation nodes during training
  - Increases memory requirements



Gradient Checkpointing

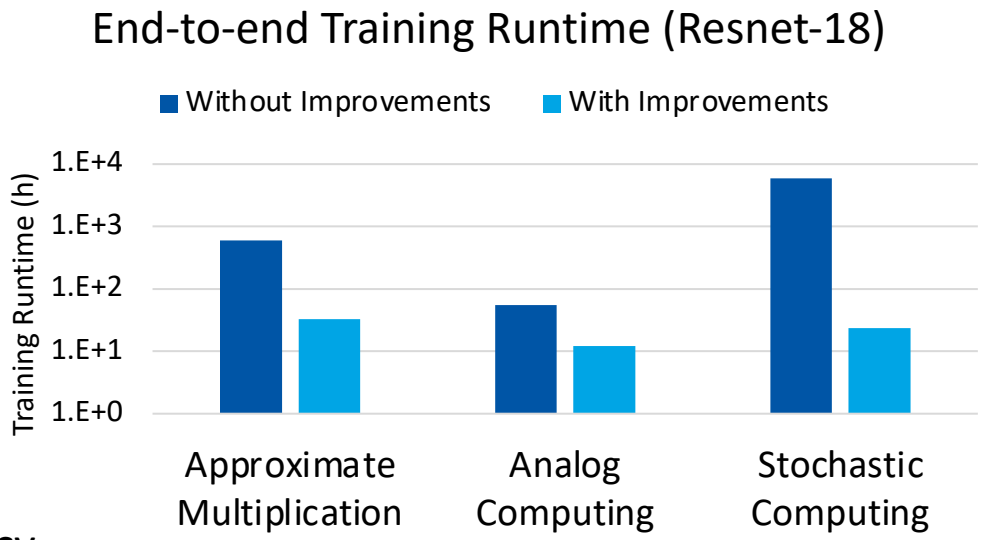
- Activation proxy and activation calibration add computation nodes during training
  - Increases memory requirements
- Use gradient checkpointing to recompute the layers during backward pass
- Reduces memory requirements with minimal performance costs
  - Added layers are point-wise
- Allows bigger batch sizes for large models

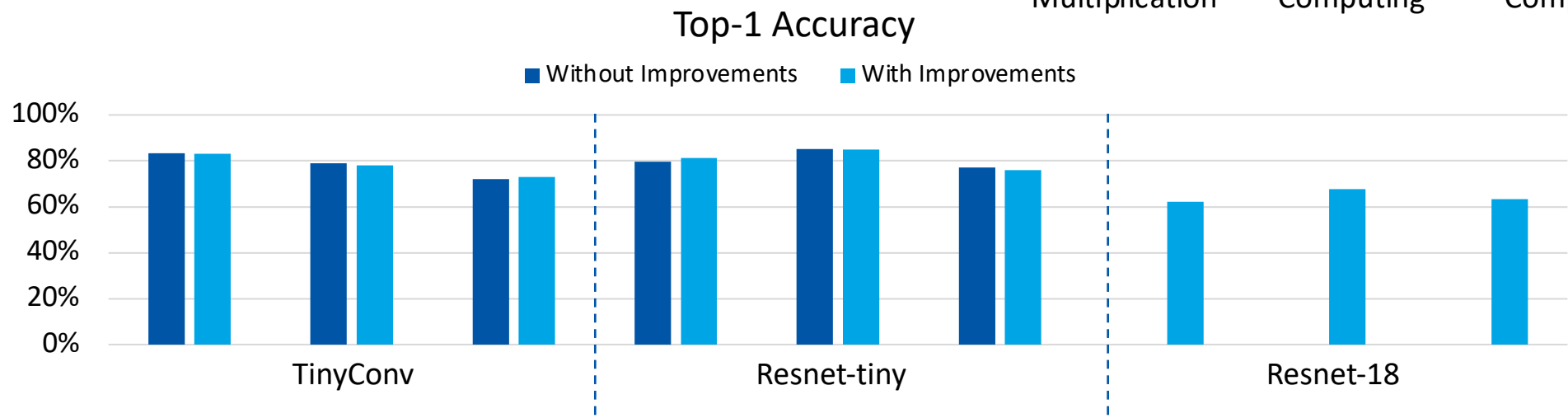


Error Injection Backward with Checkpointing

Results

- Evaluation:
  - Platform: PyTorch, single RTX 3090
  - TinyConv/Resnet-tiny CIFAR-10
  - ImageNet Resnet-18
- Our improvements reduce training runtime by 4.6X to 250X
- Enables training large models which are previously difficult/impossible to train





Conclusion

- Improve training performance for approximate computing hardware
- Use activation proxies to approximate non-linearities in the backward pass
- Use error injection and fine tuning in the forward pass to reduce expensive emulation and retain accuracy
- Use gradient checkpointing to remove memory overhead of the added computation
- Reduce end-to-end training time by **4.6X** to **250X**
- Allow training of large models for approximate computing

Copyright Notice

www.tinyml.org