MEASURING THE RELIABILITY OF REINFORCEMENT LEARNING ALGORITHMS

Stephanie C.Y. Chan,[1*] **Samuel Fishman,**[1] **John Canny,**[1, 2] **Anoop Korattikara,**[1] **& Sergio Guadarrama**[1]
[1]Google Research [2]Berkeley EECS
{scychan,sfishman,canny,kbanoop,sguada}@google.com

arXiv:1912.05663v2 [stat.ML] 12 Feb 2020

ABSTRACT

Lack of reliability is a well-known issue for reinforcement learning (RL) algorithms. This problem has gained increasing attention in recent years, and efforts to improve it have grown substantially. To aid RL researchers and production users with the evaluation and improvement of reliability, we propose a set of metrics that quantitatively measure different aspects of reliability. In this work, we focus on variability and risk, both during training and after learning (on a fixed policy). We designed these metrics to be general-purpose, and we also designed complementary statistical tests to enable rigorous comparisons on these metrics. In this paper, we first describe the desired properties of the metrics and their design, the aspects of reliability that they measure, and their applicability to different scenarios. We then describe the statistical tests and make additional practical recommendations for reporting results. The metrics and accompanying statistical tools have been made available as an open-source library.[1] We apply our metrics to a set of common RL algorithms and environments, compare them, and analyze the results.

1 INTRODUCTION

Reinforcement learning (RL) algorithms, especially Deep RL algorithms, tend to be highly variable in performance and considerably sensitive to a range of different factors, including implementation details, hyper-parameters, choice of environments, and even random seeds (Henderson et al., 2017). This variability hinders reproducible research, and can be costly or even dangerous for real-world applications. Furthermore, it impedes scientific progress in the field when practitioners cannot reliably evaluate or predict the performance of any particular algorithm, compare different algorithms, or even compare different implementations of the same algorithm.

Recently, Henderson et al. (2017) has performed a detailed analysis of reliability for several policy gradient algorithms, while Duan et al. (2016) has benchmarked average performance of different continuous-control algorithms. In other related work, Colas et al. (2018) have provided a detailed analysis on power analyses for mean performance in RL, and Colas et al. (2019) provide a comprehensive primer on statistical testing for mean and median performance in RL.

In this work, we aim to devise a set of metrics that measure reliability of RL algorithms. Our analysis distinguishes between several typical modes to evaluate RL performance: "evaluation during training", which is computed over the course of training, vs. "evaluation after learning", which is evaluated on a fixed policy after it has been trained. These metrics are also designed to measure different aspects of reliability, e.g. reproducibility (variability *across* training runs and variability *across* rollouts of a fixed policy) or stability (variability *within* training runs). Additionally, the metrics capture multiple aspects of variability – dispersion (the width of a distribution), and risk (the heaviness and extremity of the lower tail of a distribution).

Standardized measures of reliability can benefit the field of RL by allowing RL practitioners to compare algorithms in a rigorous and consistent way. This in turn allows the field to measure

*Work done as part of the Google AI Residency
[1]https://github.com/google-research/rl-reliability-metrics

progress, and also informs the selection of algorithms for both research and production environments. By measuring various aspects of reliability, we can also identify particular strengths and weaknesses of algorithms, allowing users to pinpoint specific areas of improvement.

In this paper, in addition to describing these reliability metrics, we also present practical recommendations for statistical tests to compare metric results and how to report the results more generally. As examples, we apply these metrics to a set of algorithms and environments (discrete and continuous, off-policy and on-policy). We have released the code used in this paper as an open-source Python package to ease the adoption of these metrics and their complementary statistics.

		Dispersion (D)	**Risk (R)**
DURING TRAINING	**Across Time (T) (within training runs)**	IQR* within windows, after detrending	**Short-term:** CVaR† on first-order differences **Long-term**: CVaR† on Drawdown
	Across Runs (R)	IQR* across training runs, after low-pass filtering.	CVaR† across runs
AFTER LEARNING	**Across rollouts on a Fixed Policy (F)**	IQR* across rollouts for a fixed policy	CVaR† across rollouts for a fixed policy

Table 1: Summary of our proposed reliability metrics. For evaluation DURING TRAINING, which measures reliability over the course of training an algorithm, the inputs to the metrics are the performance curves of an algorithm, evaluated at regular intervals during a single training run (or on a set of training runs). For evaluation AFTER LEARNING, which measures reliability of an already-trained policy, the inputs to the metrics are the performance scores of a set of rollouts of that fixed policy. *IQR: inter-quartile range. †CVaR: conditional value at risk.

2 RELIABILITY METRICS

We target three different axes of variability, and two different measures of variability along each axis. We denote each of these by a letter, and each metric as a combination of an axis + a measure, e.g. "DR" for "Dispersion Across Runs". See Table 1 for a summary. Please see Appendix A for more detailed definitions of the terms used here.

2.1 AXES OF VARIABILITY

Our metrics target the following three axes of variability. The first two capture reliability "during training", while the last captures reliability of a fixed policy "after learning".

During training: Across Time (T) In the setting of evaluation during training, one desirable property for an RL algorithm is to be stable "across time" within each training run. In general, smooth monotonic improvement is preferable to noisy fluctuations around a positive trend, or unpredictable swings in performance.

This type of stability is important for several reasons. During learning, especially when deployed for real applications, it can be costly or even dangerous for an algorithm to have unpredictable levels of performance. Even in cases where bouts of poor performance do not directly cause harm, e.g. if training in simulation, high instability implies that algorithms have to be check-pointed and evaluated more frequently in order to catch the peak performance of the algorithm, which can be expensive. Furthermore, while training, it can be a waste of computational resources to train an unstable algorithm that tends to forget previously learned behaviors.

During training: Across Runs (R) During training, RL algorithms should have easily and consistently reproducible performances across multiple training runs. Depending on the components that we allow to vary across training runs, this variability can encapsulate the algorithm's sensitivity to a variety of factors, such as: random seed and initialization of the optimization, random seed and initialization of the environment, implementation details, and hyper-parameter settings. Depending on the goals of the analysis, these factors can be held constant or allowed to vary, in order to disentangle the contribution of each factor to variability in training performance. High variability on any of these dimensions leads to unpredictable performance, and also requires a large search in order to find a model with good performance.

After learning: Across rollouts of a fixed policy (F) When evaluating a fixed policy, a natural concern is the variability in performance across multiple rollouts of that fixed policy. Each rollout may be specified e.g. in terms of a number of actions, environment steps, or episodes. Generally, this metric measures sensitivity to both stochasticity from the environment and stochasticity from the training procedure (the optimization). Practitioners may sometimes wish to keep one or the other constant if it is important to disentangle the two factors (e.g. holding constant the random seed of the environment while allowing the random seed controlling optimization to vary across rollouts).

2.2 Measures of variability

For each axis of variability, we have two kinds of measures: dispersion and risk.

Dispersion Dispersion is the width of the distribution. To measure dispersion, we use "robust statistics" such as the ***Inter-quartile range (IQR)*** (i.e. the difference between the 75th and 25th percentiles) and the ***Median absolute deviation from the median (MAD)***, which are more robust statistics and don't require assuming normality of the distributions. [2] We prefer to use IQR over MAD, because it is more appropriate for asymmetric distributions (Rousseeuw & Croux, 1993).

Risk In many cases, we are concerned about the worst-case scenarios. Therefore, we define risk as the heaviness and extent of the lower tail of the distribution. This is complementary to measures of dispersion like IQR, which cuts off the tails of the distribution. To measure risk, we use the ***Conditional Value at Risk (CVaR)***, also known as "expected shortfall". CVaR measures the expected loss in the worst-case scenarios, defined by some quantile α. It is computed as the expected value in the left-most tail of a distribution (Acerbi & Tasche, 2002). We use the following definition for the CVaR of a random variable X for a given quantile α:

$$\text{CVaR}_\alpha(X) = \mathbb{E}\left[X | X \leq VaR_\alpha(X)\right] \tag{1}$$

where $\alpha \in (0, 1)$ and the VaR_α (Value at Risk) is just the α-quantile of the distribution of X. Originally developed in finance, CVaR has also seen recent adoption in Safe RL as an additional component of the objective function by applying it to the cumulative returns within an episode, e.g. Bäuerle & Ott (2011); Chow & Ghavamzadeh (2014); Tamar et al. (2015). In this work, we apply CVaR to the dimensions of reliability described in Section 2.1.

2.3 Desiderata

In designing our metrics and statistical tests, we required that they fulfill the following criteria:

- A minimal number of configuration parameters – to facilitate standardization as well as to minimize "researcher degrees of freedom" (where flexibility may allow users to tune settings to produce more favorable results, leading to an inflated rate of false positives) (Simmons et al., 2011).
- Robust statistics, when possible. Robust statistics are less sensitive to outliers and have more reliable performance for a wider range of distributions. Robust statistics are especially

[2] Note that our aim here is to measure the variability of the distribution, rather than to characterize the uncertainty in estimating a statistical parameter of that distribution. Therefore, confidence intervals and other similar methods are not suitable for the aim of measuring dispersion.

important when applied to training performance, which tends to be highly non-Gaussian, making metrics such as variance and standard deviation inappropriate. For example, training performance is often bi-modal, with a concentration of points near the starting level and another concentration at the level of asymptotic performance.

- Invariance to sampling frequency – results should not be biased by the frequency at which an algorithm was evaluated during training. See Section 2.5 for further discussion.
- Enable meaningful statistical comparisons on the metrics, while making minimal assumptions about the distribution of the results. We thus designed statistical procedures that are non-parametric (Section 4).

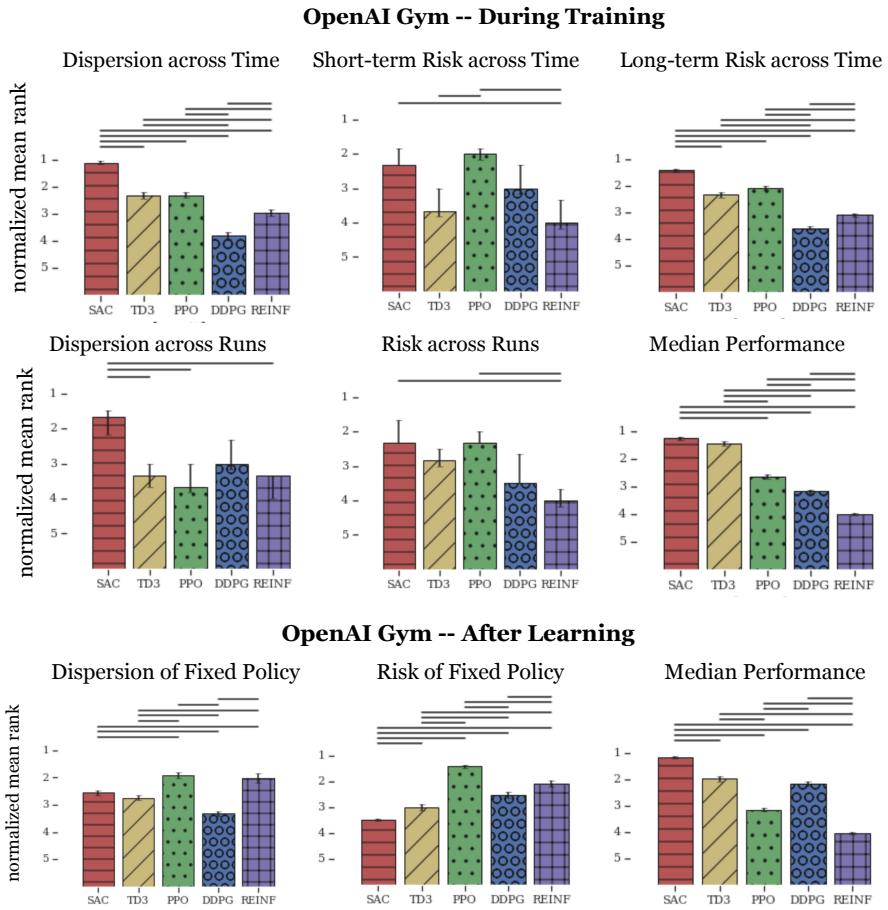


Figure 1: Reliability metrics and median performance for continuous control RL algorithms (DDPG, TD3, SAC, REINFORCE, and PPO) tested on OpenAI Gym environments. Rank 1 always indicates "best" reliability, e.g. lowest IQR across runs. Error bars are 95% bootstrap confidence intervals (# bootstraps = 1,000). Significant pairwise differences in ranking between pairs of algorithms are indicated by black horizontal lines above the colored bars. ($\alpha = 0.05$ with Benjamini-Yekutieli correction, permutation test with # permutations = 1,000). Note that the best algorithms by median performance are not always the best algorithms on reliability.

2.4 Metric Definitions

Dispersion across Time (DT): IQR across Time To measure dispersion across time (DT), we wished to isolate higher-frequency variability, rather than capturing longer-term trends. We did not want our metrics to be influenced by positive trends of improvement during training, which are in fact desirable sources of variation in the training performance. Therefore, we apply detrending before

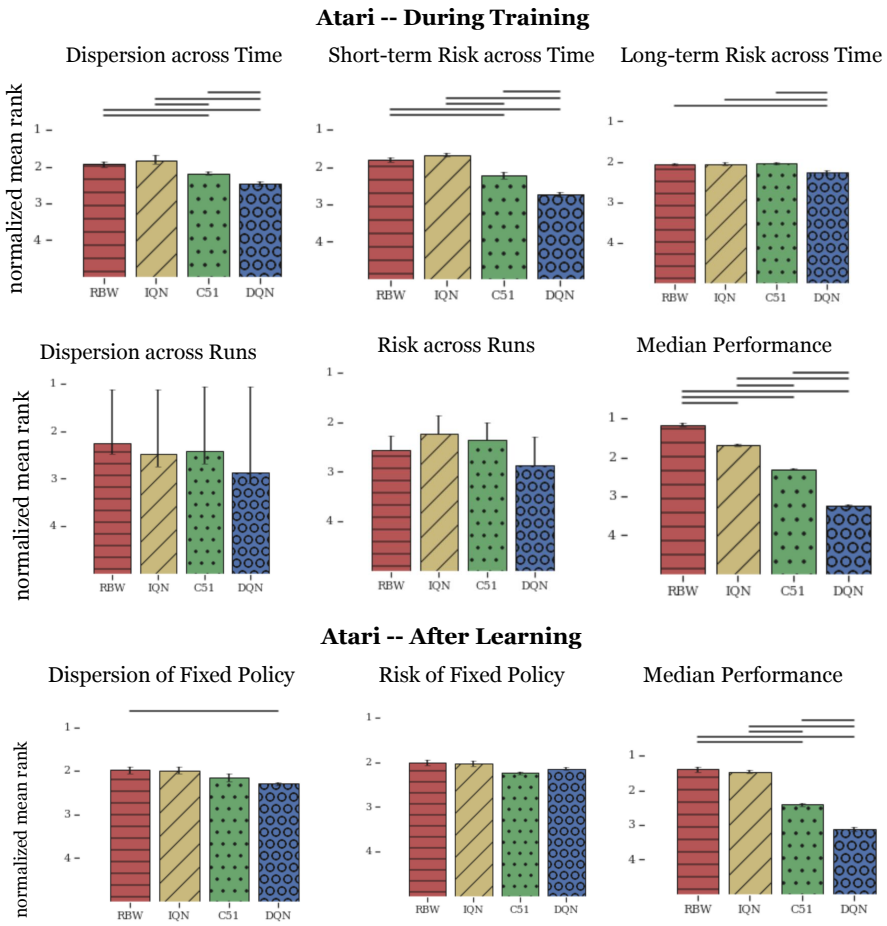


Figure 2: Reliability metrics and median performance for four DQN-variants (C51, DQN: Deep Q-network, IQ: Implicit Quantiles, and RBW: Rainbow) tested on 60 Atari games. Rank 1 always indicates "best" reliability, e.g. lowest IQR across runs. Significant pairwise differences in ranking between pairs of algorithms are indicated by black lines above the colored circles. ($\alpha = 0.05$ with Benjamini-Yekutieli correction, permutation test with # permutations = 1,000). Note that the best algorithms by median performance are not always the best algorithms on reliability. Error bars are 95% bootstrap confidence intervals (# bootstraps = 1,000).

computing dispersion metrics. For detrending, we used differencing (i.e. $y_t\prime = y_t - y_{t-1}$).[3] The final measure consisted of inter-quartile range (IQR) within a sliding window along the detrended training curve.

Short-term Risk across Time (SRT): CVaR on Differences For this measure, we wish to measure the most extreme short-term drop over time. To do this, we apply CVaR to the changes in performance from one evaluation point to the next. I.e., in Eq. 1, X represents the differences from one evaluation time-point to the next. We first compute the time-point to time-point differences on each training run. These differences are normalized by the distance between time-points, to ensure invariance to evaluation frequency (see Section 2.5). Then, we obtain the distribution of these differences, and find the α-quantile. Finally, we compute the expected value of the distribution below the α-quantile. This gives us the worst-case expected drop in performance during training, from one point of evaluation to the next.

[3]Please see Appendix B for a more detailed discussion of different types of detrending, and the rationale for choosing differencing here.

Long-term Risk across Time (LRT): CVaR on Drawdown For this measure, we would also like to be able to capture whether an algorithm has the potential to lose a lot of performance relative to its peak, even if on a longer timescale, e.g. over an accumulation of small drops. For this measure, we apply CVaR to the ***Drawdown***. The Drawdown at time T is the drop in performance relative to the highest peak so far, and is another measure borrowed from economics (Chekhlov et al., 2005). I.e. $\text{Drawdown}_T = R_T - \max_{t<=T} R_t$. Like the SRT metric, the LRT can capture unusually large short-term drops in performance, but can also capture unusually large drops that occur over longer timescales.

Dispersion across Runs (DR): IQR across Runs Unlike the rest of the metrics described here, the dispersion across training runs has previously been used to characterize performance (e.g. Duan et al. (2016); Islam et al. (2017); Bellemare et al. (2017); Fortunato et al. (2017); Nagarajan et al. (2018)). This is usually measured by taking the variance or standard deviation across training runs at a set of evaluation points. We build on the existing practice by recommending first performing low-pass filtering of the training data, to filter out high-frequency variability within runs (this is instead measured using Dispersion across Time, DT). We also replace variance or standard deviation with robust statistics like IQR.

Risk across Runs (RR): CVaR across Runs In order to measure Risk across Runs (RR), we apply CVaR to the final performance of all the training runs. This gives a measure of the expected performance of the worst runs.

Dispersion across Fixed-Policy Rollouts (DF): IQR across Rollouts When evaluating a fixed policy, we are interested in variability in performance when the same policy is rolled out multiple times. To compute this metric, we simply compute the IQR on the performance of the rollouts.

Risk across Fixed-Policy Rollouts (RF): CVaR across Rollouts This metric is similar to DF, except that we apply CVaR on the rollout performances.

2.5 Invariance to frequency of evaluation

Different experiments and different tasks may produce evaluations at different frequencies during training. Therefore, the reliability metrics should be unbiased by the choice of evaluation frequency. As long as there are no cyclical patterns in performance, the frequency of evaluation will not bias any of the metrics except Long-Term Risk across Time (LRT). For all other metrics, changes in the frequency of evaluation will simply lead to more or less noisy estimates of these metrics. For LRT, comparisons should only be made if the frequency of evaluation is held constant across experiments.

3 Recommendations for reporting metrics and parameters

Whether evaluating an algorithm for practical use or for research, we recommend evaluating all of the reliability metrics described above. Each metric measures a different aspect of reliability, and can help pinpoint specific strengths and weaknesses of the algorithm. Evaluating the metrics is easy with the open-source Python package that we have released.

Reporting parameters. Even given our purposeful efforts to minimize the number of parameters in the reliability metrics, a few remain to be specified by the user that can affect the results, namely: window size (for Dispersion across Time), frequency threshold for low-pass and high-pass filtering (Dispersion across Time, Dispersion across Runs), evaluation frequency (only for Long-term Risk across Time), and length of training runs. Therefore, when reporting these metrics, these parameters need to be clearly specified, and must also be held constant across experiments for meaningful comparisons. The same is true for any other parameters that affect evaluation, e.g., the number of roll-outs per evaluation, the parameters of the environment, whether on-line or off-line evaluation is used, and the random seeds chosen.

Collapsing across evaluation points. Some of the in-training reliability metrics (Dispersion across Runs, Risk across Runs, and Dispersion across Time) need to be evaluated at multiple evaluation

points along the training runs. If it is useful to obtain a small number of values to summarize each metric, we recommend dividing the training run into "time frames" (e.g. beginning, middle, and end), and collapsing across all evaluation points within each time frame.

Normalization by performance. Different algorithms can have vastly different ranges of performance even on the same task, and variability in performance tends to scale with actual performance. Thus, we normalize our metrics in post-processing by a measure of the range of performance for each algorithm. For "during training" reliability, we recommend normalizing by the median range of performance, which we define as the $p_{P_95} - p_{t=0}$, where p_{P_95} is the 95th percentile and $p_{t=0}$ is the starting performance. For "after learning" reliability, the range of performance may not be available, in which case we use the median performance directly.

Ranking the algorithms. Because different environments have different ranges and distributions of reward, we must be careful when aggregating across environments or comparing between environments. Thus, if the analysis involves more than one environment, the per-environment median results for the algorithms are first converted to rankings, by ranking all algorithms within each task. To summarize the performance of a single algorithm across multiple tasks, we compute the mean ranking across tasks.

Per-environment analysis. The same algorithm can have different patterns of reliability for different environments. Therefore, we recommend inspecting reliability metrics on a per-environment basis, as well as aggregating across environments as described above.

4 Confidence Intervals and Statistical Significance Tests for Comparison

4.1 Confidence intervals

We assume that the metric values have been converted to mean rankings, as explained in Section 3. To obtain confidence intervals on the mean rankings for each algorithm, we apply bootstrap sampling on the runs, by resampling runs with replacement (Efron & Tibshirani, 1986).

For metrics that are evaluated per-run (e.g. Dispersion across Time), we can resample the metric values directly, and then recompute the mean rankings on each resampling to obtain a distribution over the rankings; this allow us to compute confidence intervals. For metrics that are evaluated across-runs, we need to resample the runs themselves, then evaluate the metrics on each resampling, before recomputing the mean rankings to obtain a distribution on the mean rankings.

4.2 Significance Tests for Comparing algorithms

Commonly, we would like to compare algorithms evaluated on a fixed set of environments. To determine whether any two algorithms have statistically significant differences in their metric rankings, we perform an exact permutation test on each pair of algorithms. Such tests allow us to compute a p-value for the null hypothesis (probability that the methods are in fact indistinguishable on the reliability metric).

We designed our permutation tests based on the null hypothesis that runs are exchangeable across the two algorithms being compared. In brief, let A and B be sets of performance measurements for algorithms a and b. Let $Metric(X)$ be a reliability metric, e.g. the inter-quartile range across runs, computed on a set of measurements X. $MetricRanking(X)$ is the mean ranking across tasks on X, compared to the other algorithms being considered. We compute test statistic

$$s_{MetricRanking}(A, B) = MetricRanking(A) - MetricRanking(B).$$

Next we compute the distribution for $s_{MetricRanking}$ under the null hypothesis that the methods are equivalent, i.e. that performance measurements should have the same distribution for a and b. We do this by computing random partitions A', B' of $\{A \cup B\}$, and computing the test statistic $s_{MetricRanking}(A', B')$ on each partition. This yields a distribution for $s_{MetricRanking}$ (for sufficiently many samples), and the p-value can be computed from the percentile value of

$s_{MetricRanking}(A, B)$ in this distribution. As with the confidence intervals, a different procedure is required for per-run vs across-run metrics. Please see Appendix C for diagrams illustrating the permutation test procedures.

When performing pairwise comparisons between algorithms, it is critical to include corrections for multiple comparisons. This is because the probability of incorrect inferences increases with a greater number of simultaneous comparisons. We recommend using the Benjamini-Yekutieli method, which controls the false discovery rate (FDR), i.e., the proportion of rejected null hypotheses that are false.[4]

4.3 Reporting on statistical tests

It is important to report the details of any statistical tests performed, e.g. which test was used, the significance threshold, and the type of multiple-comparisons correction used.

5 Analysis of Reliability for Common Algorithms and Environments

In this section, we provide examples of applying the reliability metrics to a number of RL algorithms and environments, following the recommendations described above.

5.1 Continuous control algorithms on OpenAI Gym

We applied the reliability metrics to algorithms tested on seven continuous control environments from the Open-AI Gym (Greg Brockman et al., 2016) run on the MuJoCo physics simulator (Todorov et al., 2012). We tested REINFORCE (Sutton et al., 2000), DDPG (Lillicrap et al., 2015), PPO (Schulman et al., 2017), TD3 (Fujimoto et al., 2018), and SAC (Haarnoja et al., 2018) on the following Gym environments: Ant-v2, HalfCheetah-v2, Humanoid-v2, Reacher-v2, Swimmer-v2, and Walker2d-v2. We used the implementations of DDPG, TD3, and SAC from the TF-Agents library (Guadarrama et al., 2018). Each algorithm was run on each environment for 30 independent training runs.

We used a black-box optimizer (Golovin et al., 2017) to tune selected hyperparameters on a per-task basis, optimizing for final performance. The remaining hyperparameters were defined as stated in the corresponding original papers. See Appendix E for details of the hyperparameter search space and the final set of hyperparameters. During training, we evaluated the policies at a frequency of 1000 training steps. Each algorithm was run for a total of two million environment steps. For the "online" evaluations we used the generated training curves, averaging returns over recent training episodes collected using the exploration policy as it evolves. The raw training curves are shown in Appendix D. For evaluations after learning on a fixed policy, we took the last checkpoint from each training run as the fixed policy for evaluation. Each of these policies was then evaluated for 30 roll-outs, where each roll-out was defined as 1000 environment steps.

5.2 Discrete control: DQN variants on Atari

We also applied the reliability metrics to the RL algorithms and training data released as part of the Dopamine package (Castro et al., 2018). The data comprise the training runs of four RL algorithms, each applied to 60 Atari games. The RL algorithms are: DQN (Mnih et al., 2015), Implicit Quantile (IQN) (Dabney et al., 2018), C51 (Bellemare et al., 2017), and a variant of Rainbow implementing the three most important components (Hessel & Modayil, 2018). The algorithms were trained on each game for 5 training runs. Hyper-parameters follow the original papers, but were modified as necessary to follow Rainbow (Hessel & Modayil, 2018), to ensure apples-to-apples comparison. See Appendix E for the hyperparameters.

During training, the algorithms were evaluated in an "online" fashion every 1 million frames, averaging across the training episodes as recommended for evaluations on the ALE (Machado et al., 2018). Each training run consisted of approximately 200 million Atari frames (rounding to the nearest

[4]For situations in which a user wishes instead to control the family-wise error rate (FWER; the probability of incorrectly rejecting at least one true null hypothesis), we recommend using the Holm-Bonferroni method.

episode boundary every 1 million frames).[5] For evaluations after learning on a fixed policy ("after learning"), we took the last checkpoint from each training run as the fixed policies for evaluation. We then evaluated each of these policies for 125,000 environment steps.

5.3 Parameters for reliability metrics, confidence intervals, and statistical tests

For the MuJoCo environments, we applied a sliding window of 100000 training steps for Dispersion across Time. For the Atari experiments, we used a sliding window size of 25 on top of the evaluations for the Dispersion across Time. For metrics with multiple evaluation points, we divided each training run into 3 time frames and averaged the metric rankings within each time frame. Because the results were extremely similar for all three time frames, we here report just for the final time frames.

Statistical tests for comparing algorithms were performed according to the recommendations in Section 4. We used pairwise permutation tests using 10,000 permutations per test, with a significance threshold of 0.05 and Benjamini-Yekutieli multiple-comparisons correction.

5.4 Median performance

The median performance of an algorithm is not a reliability metric, but it is interesting to see side-by-side with the reliability metrics. For analyzing median performance for the DQN variants, we used the normalization scheme of (Mnih et al., 2015), where an algorithm's performance is normalized against a lower baseline (e.g. the performance of a random policy) and an upper baseline (e.g. the performance of a human): $P_{\text{normalized}} = \frac{P - B_{\text{lower}}}{B_{\text{upper}} - B_{\text{lower}}}$. Median performance was not normalized for the continuous control algorithms.

5.5 Results

The reliability metric rankings are shown in Fig. 1 for the MuJoCo results. We see that, according to Median Performance during training, SAC and TD3 have the best performance and perform similarly well, while REINFORCE performs the worst. However, SAC outperforms TD3 on all reliability metrics during training. Furthermore, both SAC and TD3 perform relatively poorly on all reliability metrics after learning, despite performing best on median performance.

The reliability metric rankings are shown in Fig. 2 for the Atari results. Here we see a similar result that, even though Rainbow performs significantly better than IQN in Median Performance, IQN performs numerically or significantly better than Rainbow on many of the reliability metrics.

The differing patterns in these metrics demonstrates that reliability is a separate dimension that needs to be inspected separately from mean or median performance – two algorithms may have similar median performance but may nonetheless significantly differ in reliability, as with SAC and TD3 above. Additionally, these results demonstrate that reliability along one axis does not necessarily correlate with reliability on other axes, demonstrating the value of evaluating these different dimensions so that algorithms can be compared and selected based on the requirements of the problem at hand.

To see metric results evaluated on a per-environment basis, please refer to Appendix F. Rank order of algorithms was often relatively consistent across the different environments evaluated. However, different environments did display different patterns across algorithms. For example, even though SAC showed the same or better Dispersion across Runs for most of the MuJoCo environments evaluated, it did show slightly worse Dispersion across Runs for the HalfCheetah environment (Fig 7a). This kind of result emphasizes the importance of inspecting reliability (and other performance metrics) on a per-environment basis, and also of evaluating reliability and performance on the environment of interest, if possible.

[5] The raw training curves can be viewed at https://google.github.io/dopamine/baselines/plots.html

6 CONCLUSION

We have presented a number of metrics, designed to measure different aspects of reliability of RL algorithms. We motivated the design goals and choices made in constructing these metrics, and also presented practical recommendations for the measurement of reliability for RL. Additionally, we presented examples of applying these metrics to common RL algorithms and environments, and showed that these metrics can reveal strengths and weaknesses of an algorithm that are obscured when we only inspect mean or median performance.

ACKNOWLEDGMENTS

Many thanks to the following people for helpful discussions during the formulation of these metrics and the writing of the paper: Mohammad Ghavamzadeh, Yinlam Chow, Danijar Hafner, Rohan Anil, Archit Sharma, Vikas Sindhwani, Krzysztof Choromanski, Joelle Pineau, Hal Varian, Shyue-Ming Loh, and Tim Hesterberg. Thanks also to Toby Boyd for his assistance in the open-sourcing process, Oscar Ramirez for code reviews, and Pablo Castro for his help with running experiments using the Dopamine baselines data.

REFERENCES

Carlo Acerbi and Dirk Tasche. Expected Shortfall: A Natural Coherent Alternative to Value at Risk. *Economic Notes*, 31(2):379–388, July 2002. ISSN 0391-5026, 1468-0300. doi: 10.1111/1468-0300.00091. URL http://doi.wiley.com/10.1111/1468-0300.00091.

Marc G. Bellemare, Will Dabney, and Rémi Munos. A Distributional Perspective on Reinforcement Learning. *arXiv:1707.06887 [cs, stat]*, July 2017. URL http://arxiv.org/abs/1707.06887. arXiv: 1707.06887.

Nicole Bäuerle and Jonathan Ott. Markov Decision Processes with Average-Value-at-Risk criteria. *Mathematical Methods of Operations Research*, 74(3):361–379, December 2011. ISSN 1432-2994, 1432-5217. doi: 10.1007/s00186-011-0367-0. URL http://link.springer.com/10.1007/s00186-011-0367-0.

Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A research framework for deep reinforcement learning. *CoRR*, abs/1812.06110, 2018. URL http://arxiv.org/abs/1812.06110.

Alexei Chekhlov, Stanislav Uryasev, and Michael Zabarankin. Drawdown measure in portfolio optimization. *International Journal of Theoretical and Applied Finance*, 8(1):46, 2005.

Yinlam Chow and Mohammad Ghavamzadeh. Algorithms for CVaR Optimization in MDPs. *Advances in Neural Information Processing Systems*, pp. 9, 2014.

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. How Many Random Seeds? Statistical Power Analysis in Deep Reinforcement Learning Experiments. *arXiv:1806.08295 [cs, stat]*, June 2018. URL http://arxiv.org/abs/1806.08295. arXiv: 1806.08295.

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. A Hitchhiker's Guide to Statistical Comparisons of Reinforcement Learning Algorithms. *arXiv:1904.06979 [cs, stat]*, April 2019. URL http://arxiv.org/abs/1904.06979. arXiv: 1904.06979.

Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit Quantile Networks for Distributional Reinforcement Learning. *Thirty-fith International Conference on Machine Learning*, pp. 10, 2018.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking Deep Reinforcement Learning for Continuous Control. In *International Conference on Machine Learning*, pp. 1329–1338, June 2016. URL http://proceedings.mlr.press/v48/duan16.html.

B. Efron and R. Tibshirani. Bootstrap Methods for Standard Errors, Confidence Intervals, and Other Measures of Statistical Accuracy. *Statistical Science*, 1(1):54–75, February 1986. ISSN 0883-4237, 2168-8745. doi: 10.1214/ss/1177013815. URL http://projecteuclid.org/euclid.ss/1177013815.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy Networks for Exploration. *arXiv:1706.10295 [cs, stat]*, June 2017. URL http://arxiv.org/abs/1706.10295. arXiv: 1706.10295.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. *arXiv:1802.09477 [cs, stat]*, February 2018. URL http://arxiv.org/abs/1802.09477. arXiv: 1802.09477.

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, pp. 1487–1495, Halifax, NS, Canada, 2017. ACM Press. ISBN 978-1-4503-4887-4. doi: 10.1145/3097983.3098043. URL http://dl.acm.org/citation.cfm?doid=3097983.3098043.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.

Sergio Guadarrama, Anoop Korattikara, Pablo Castro Oscar Ramirez, Ethan Holly, Sam Fishman, Ke Wang, Chris Harris Ekaterina Gonina, Vincent Vanhoucke, and Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. URL https://github.com/tensorflow/agents. [Online; accessed 30-November-2018].

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]*, January 2018. URL http://arxiv.org/abs/1801.01290. arXiv: 1801.01290.

James D. Hamilton. *Time Series Analysis*. Princeton University Press, 1994.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning that Matters. *arXiv:1709.06560 [cs, stat]*, September 2017. URL http://arxiv.org/abs/1709.06560. arXiv: 1709.06560.

Matteo Hessel and Joseph Modayil. Rainbow: Combining Improvements in Deep Reinforcement Learning. *AAAI*, pp. 8, 2018.

Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control. *arXiv:1708.04133 [cs]*, August 2017. URL http://arxiv.org/abs/1708.04133. arXiv: 1708.04133.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*, September 2015. URL http://arxiv.org/abs/1509.02971. arXiv: 1509.02971.

Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *Journal of Artificial Intelligence Research*, 61:523–562, March 2018. ISSN 1076-9757. doi: 10.1613/jair.5699. URL https://www.jair.org/index.php/jair/article/view/11182.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 1476-4687. doi: 10.1038/nature14236. URL https://www.nature.com/articles/nature14236/.

Prabhat Nagarajan, Garrett Warnell, and Peter Stone. Deterministic Implementations for Reproducibility in Deep Reinforcement Learning. *arXiv:1809.05676 [cs]*, September 2018. URL http://arxiv.org/abs/1809.05676. arXiv: 1809.05676.

Charles R Nelson and Charles I Plosser. Trends and random walks in macroeconomic time series. *Journal of Monetary Economics*, 10:139–162, 1982.

Peter J. Rousseeuw and Christophe Croux. Alternatives to the MedianAbsolute Deviation. *Journal of the American Statistical Association*, 1993.

Said E. Said and David A. Dickey. Testing for unit roots in autoregressive-moving average models of unknown order. *Biometrika*, 71(3):599–607, 1984.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, July 2017. URL http://arxiv.org/abs/1707.06347. arXiv: 1707.06347.

Joseph P. Simmons, Leif D. Nelson, and Uri Simonsohn. False-Positive Psychology: Undisclosed Flexibility in Data Collection and Analysis Allows Presenting Anything as Significant. *Psychological Science*, 22(11):1359–1366, November 2011. ISSN 0956-7976, 1467-9280. doi: 10.1177/0956797611417632. URL http://journals.sagepub.com/doi/10.1177/0956797611417632.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *NIPS'99 Proceedings of the 12th International Conference on Neural Information Processing Systems*, pp. 7, 2000.

Aviv Tamar, Yonatan Glassner, and Shie Mannor. Optimizing the CVaR via Sampling. *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pp. 7, 2015.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, Vilamoura-Algarve, Portugal, October 2012. IEEE. ISBN 978-1-4673-1736-8 978-1-4673-1737-5 978-1-4673-1735-1. doi: 10.1109/IROS.2012.6386109. URL http://ieeexplore.ieee.org/document/6386109/.

A Assumptions and Definitions

Reinforcement Learning algorithms vary widely in design, and our metrics are based on certain notions that should span the gamut of RL algorithms.

Policy A policy $\pi_\Theta(a_i|s_i)$ is a distribution over actions a_i given a current (input) state s_i. We assume policies are parameterized by a parameter Θ.

Agent An agent is defined as a distribution over policies (or equivalently a distribution over parameters Θ). In many cases, an agent will be a single policy but for population-based RL methods, the agent is a discrete set of policies.

Window A window is a collection of states over which the agent is assumed to have small variation. A window could be a sequence of consecutive time steps for a sequential RL algorithm, or a collection of states at the same training step of a distributed RL algorithm with a parameter server (all agents share Θ).

Performance The performance of an agent is the mean or median per-epoch reward from running that agent. If the agent is a single policy, then the performance $p(\pi_\Theta)$ is the mean or median per-epoch reward for that agent. If the agent is a distribution $D(\Theta)$ of policies, then the performance is the median of $p(\pi_\Theta)$ with $\Theta \sim D$.

Training Run A training run is a sequence of updates to the agent $D(\Theta)$ from running a reinforcement learning algorithm. It leads to a trained agent $D_{final}(\Theta)$. Multiple training runs share no information with each other.

We cannot directly measure performance since it is a statistic across an infinite sample of evaluation runs of an agent. Instead we use windows to compute sample medians to approximate performance.

B DETRENDING BY DIFFERENCING

Typically, de-trending can be performed in two main ways (Nelson & Plosser, 1982; Hamilton, 1994). Differencing (i.e. $y_t\prime = y_t - y_{t-1}$) is more appropriate for difference-stationary (DS) processes (e.g. a random walk: $y_t = y_{t-1} + b + \epsilon_t$), where the shocks ϵ_t accumulate over time. For trend-stationary (TS) processes, which are characterized by stationary fluctuations around a deterministic trend, e.g. $y_t = a + b * t + \epsilon_t$, it is more appropriate to fit and subtract that trend.

We performed an analysis of real training runs and verified that the data are indeed approximately DS, and that differencing does indeed remove the majority of time-dependent structure. For this analysis we used the training runs on Atari as described in 5.2. Before differencing, the Augmented Dickey-Fuller test (ADF test, also known as a difference-stationarity test; Said E. Said & David A. Dickey (1984)) rejects the null hypothesis of a unit root on only 72% of the runs; after differencing, the ADF test rejects the null hypothesis on 92% of the runs (p-value threshold 0.05). For the ADF test, the rejection of a unit root (of the autoregressive lag polynomial) implies the alternate hypothesis, which is that the time series is trend-stationary.

Therefore, our training curves are better characterized as an accumulation of shocks, i.e. as DS processes, rather than as mean-reverting TS processes. They are not actually purely DS because the shocks ϵ_t are not stationary over time, but because we compute standard deviation within sliding windows, we can capture the non-stationarity and change in variability over time. Thus, we chose to detrend using differencing.

As a further note in favor of detrending by differencing, it is useful to observe that many measures of variability are defined relative to the central tendency of the data, e.g. the median absolute deviation MAD $= \text{median}(|X_i - \widetilde{X}|)$ where $\widetilde{X}$ is the median of X. On the raw data (without differencing), the MAD would be defined relative to $\widetilde{X}$ as median performance, so that any improvements in performance are included in that computation of variability. On the other hand, if we compute MAD on the 1st-order differences, we are using a $\widetilde{X}$ that represents the median *change* in performance, which is a more reasonable baseline to compute variability against, when we are in fact concerned with the variability of those changes.

A final benefit of differencing is that it is parameter-free.

C ILLUSTRATIONS OF PERMUTATION TEST PROCEDURES

We illustrate the procedure for computing permutation tests to compare pairs of algorithms on a specified metric, in Figs. 3 (for per-run metrics) and 4 (for across-run metrics).

D RAW TRAINING CURVES FOR OPENAI MUJOCO TASKS

In Figure 5, we show the raw training curves for the TF-Agents implementations of continuous-control algorithms, applied to the OpenAI MuJoCo tasks. These are compared against baselines from the literature, where available (DDPG and TD3: Fujimoto et al. (2018), PPO: Schulman et al. (2017), SAC: Haarnoja et al. (2018))

E HYPERPARAMETER SETTINGS

For the continuous control experiments, hyperparameters were chosen on a per-environment basis according to the black-box optimization algorithm described in Golovin et al. (2017). The hyperparameter search space is shown in Table 2.

For the discrete control experiments, hyperparameter selection is described in (Castro et al., 2018). Hyperparameters are shown in Table 8, duplicated for reference from https://github.com/google/dopamine/tree/master/baselines.

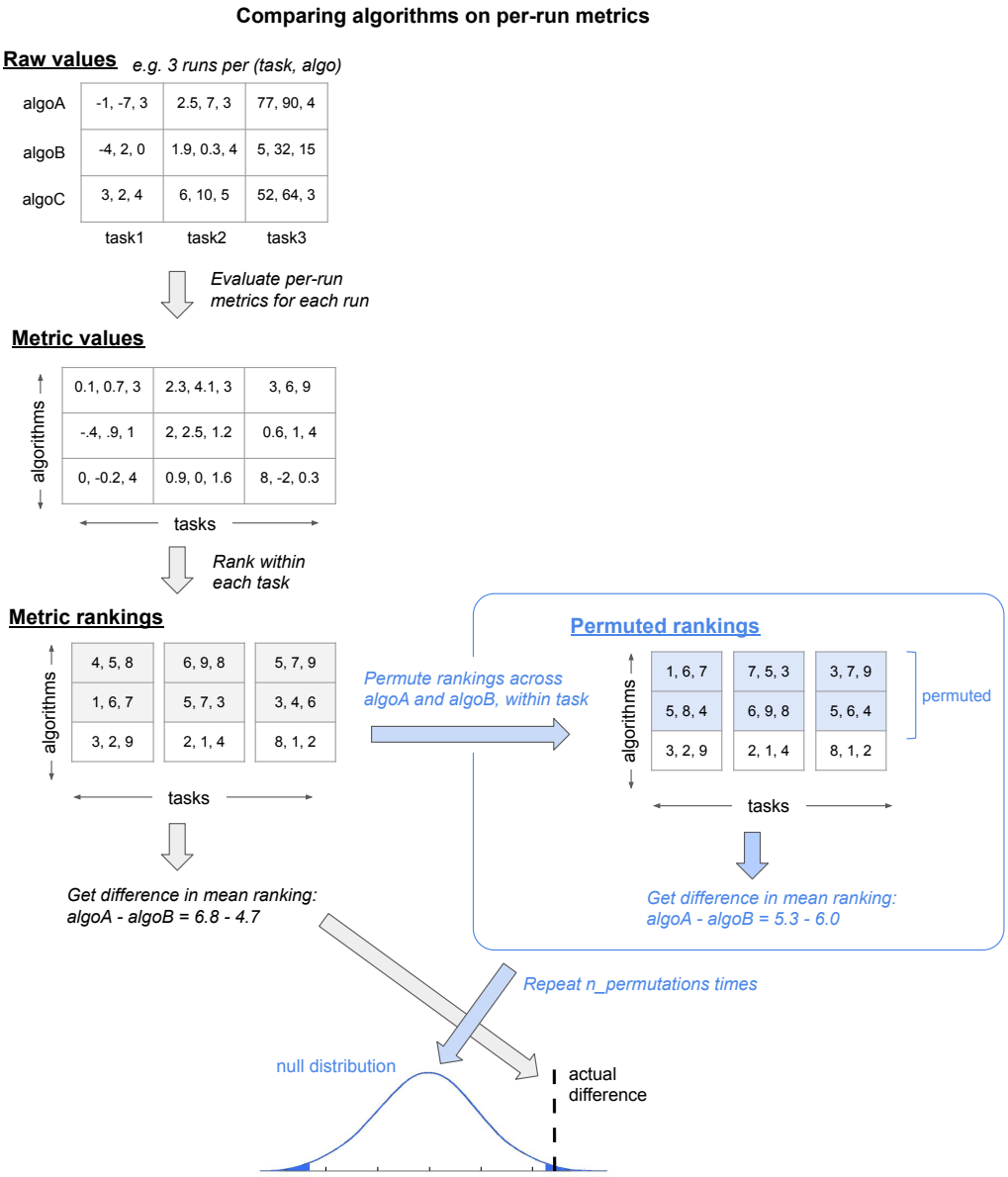


Figure 3: Diagram illustrating the computation of the permutation tests for **per-run metrics** (Dispersion across Time, Short-term Risk across Time, Long-term Risk across Time). In this example, we are comparing Algorithm A and Algorithm B, and there are only 3 algorithms, 3 tasks, and 3 runs per (task, algo) pair. To compute the difference in average rankings for two algorithms, follow the gray arrows. To compute a null distribution of difference in average rankings (by permuting the runs), follow the blue arrows a number of times (e.g. 1,000 times). Once the null distribution has been computed, the actual value of the difference can be compared with the null distribution to obtain a p-value.

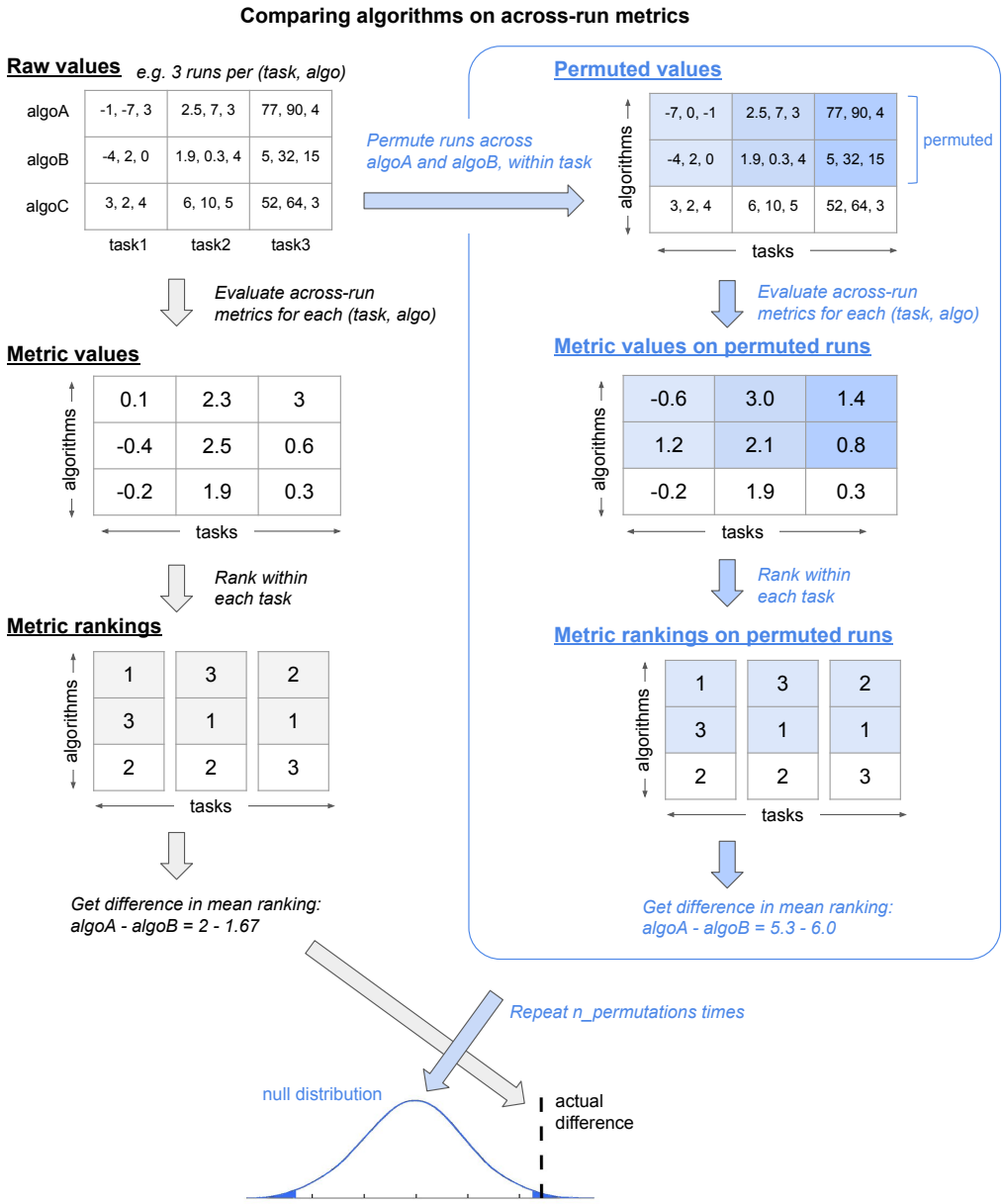


Figure 4: Diagram illustrating the computation of the permutation tests for **across-run or across-rollout metrics** (Dispersion across Runs, Risk Across Runs, Dispersion across Fixed-policy rollouts, Risk across Fixed-Policy rollouts). In this example, we are comparing Algorithm A and Algorithm B, and there are only 3 algorithms, 3 tasks, and 3 runs per (task, algo) pair. To compute the difference in average rankings for two algorithms, follow the gray arrows. To compute a null distribution of difference in average rankings (by permuting the runs), follow the blue arrows a number of times (e.g. 1,000 times). Once the null distribution has been computed, the actual value of the difference can be compared with the null distribution to obtain a p-value.

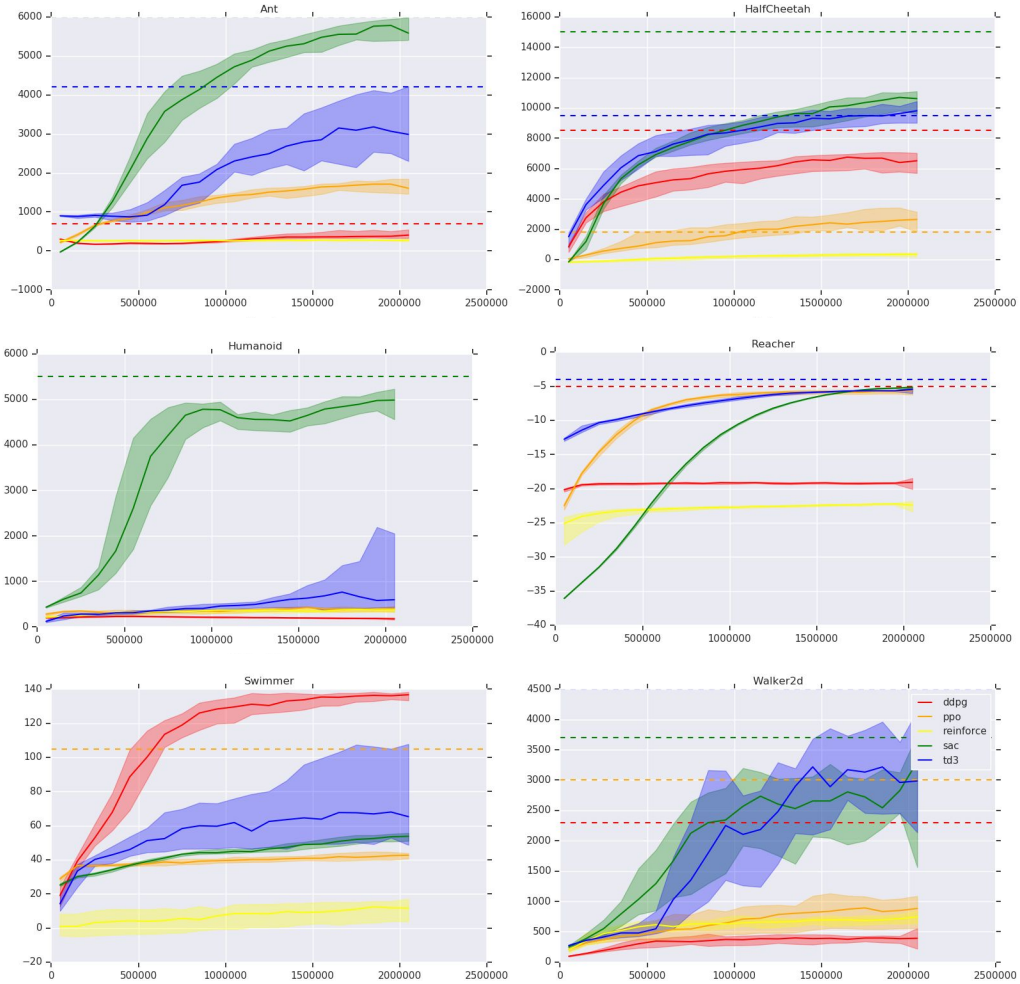


Figure 5: Raw training curves for OpenAI MuJoCo tasks. The x-axes indicate environment steps, and the y-axes indicate average per-episode return. Dotted lines indicate baseline performance from the literature, where available.

Table 2: Hyperparameter search space for continuous control algorithms.

Algorithm	**Hyperparameter**	**Search min**	**Search max**
SAC	actor learning rate	0.000001	0.001
	α learning rate	0.000001	0.001
	critic learning rate	0.000001	0.001
	target update τ	0.00001	1.0
TD3	actor learning rate	0.000001	0.001
	critic learning rate	0.000001	0.001
	target update τ	0.00001	1.0
PPO	learning rate	0.000001	0.001
DDPG	actor learning rate	0.000001	0.001
	critic learning rate	0.000001	0.001
	target update τ	0.00001	1.0
REINFORCE	learning rate	0.000001	0.001
	# episodes before each train step	1.0	10

Table 3: Final hyperparameters for SAC.

	actor learning rate	α learning rate	critic learning rate	target update τ
Ant-v2	0.000006	0.000009	0.0009	0.0002
HalfCheetah-v2	0.0001	0.000005	0.0004	0.02
Humanoid-v2	0.0003	0.0008	0.0006	0.8
Reacher-v2	0.00001	0.000002	0.0005	0.00002
Swimmer-v2	0.000004	0.000009	0.0002	0.009
Walker2d-v2	0.0002	0.0009	0.0008	0.01

Table 4: Final hyperparameters for TD3.

	actor learning rate	critic learning rate	target update τ
Ant-v2	0.000001	0.0002	0.0003
HalfCheetah-v2	0.0003	0.0005	0.02
Humanoid-v2	0.0001	0.0001	0.0002
Reacher-v2	0.000001	0.00003	0.00003
Swimmer-v2	0.0004	0.0002	0.01
Walker2d-v2	0.00006	0.00009	0.001

Table 5: Final hyperparameters for PPO.

	learning rate
Ant-v2	0.0008
HalfCheetah-v2	0.0008
Humanoid-v2	0.0008
Reacher-v2	0.00002
Swimmer-v2	0.0004
Walker2d-v2	0.0002

Table 6: Final hyperparameters for DDPG.

	actor learning rate	critic learning rate	target update τ
Ant-v2	0.00003	0.0004	0.0002
HalfCheetah-v2	0.00006	0.0005	0.02
Humanoid-v2	0.00006	0.00009	0.01
Reacher-v2	0.00005	0.0005	0.005
Swimmer-v2	0.0005	0.0003	0.004
Walker2d-v2	0.0003	0.0004	0.03

Table 7: Final hyperparameters for REINFORCE.

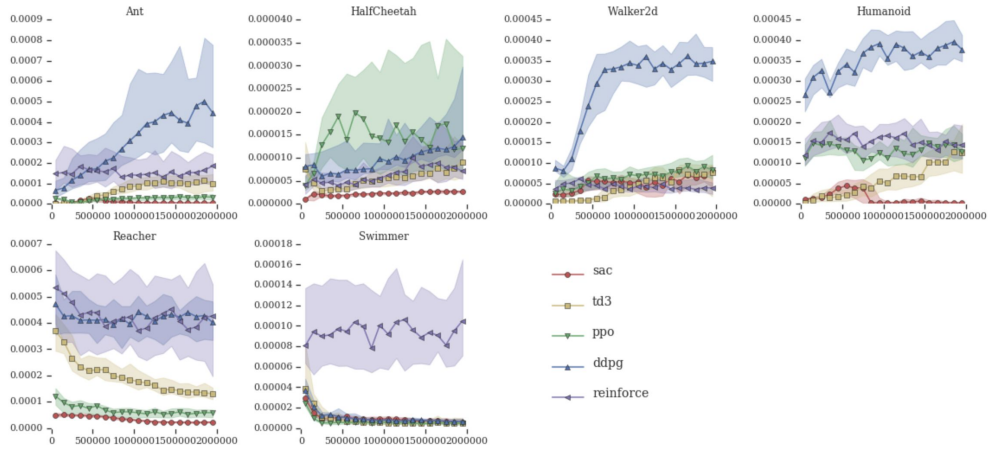
	learning rate	# episodes before each train step
Ant-v2	0.00002	9
HalfCheetah-v2	0.0004	7
Humanoid-v2	0.0005	2
Reacher-v2	0.000004	6
Swimmer-v2	0.000005	3
Walker2d-v2	0.0001	6

Table 8: Hyperparameters for discrete control algorithms.

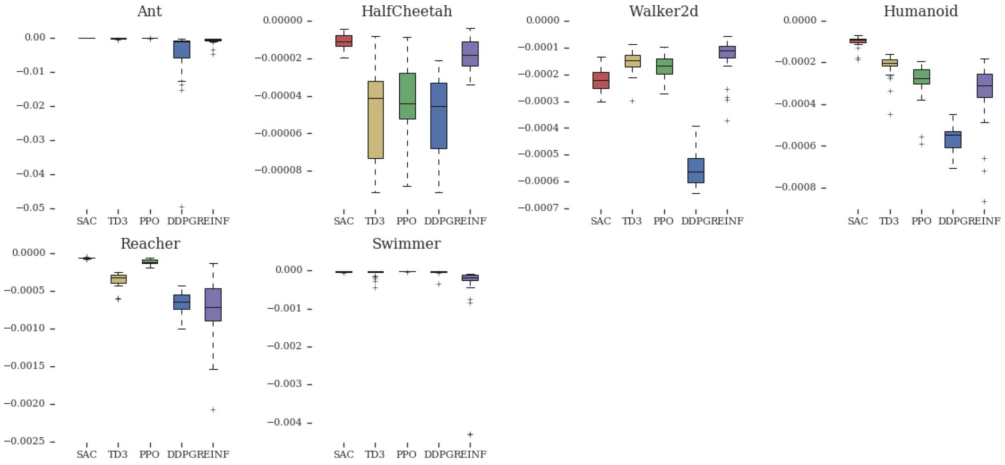
Training ϵ	Evaluation ϵ	ϵ decay schedule	Min. history to start learning	Target network update frequency
0.01	0.001	1,000,000 frames	80,000 frames	32,000 frames

F PER-TASK METRIC RESULTS

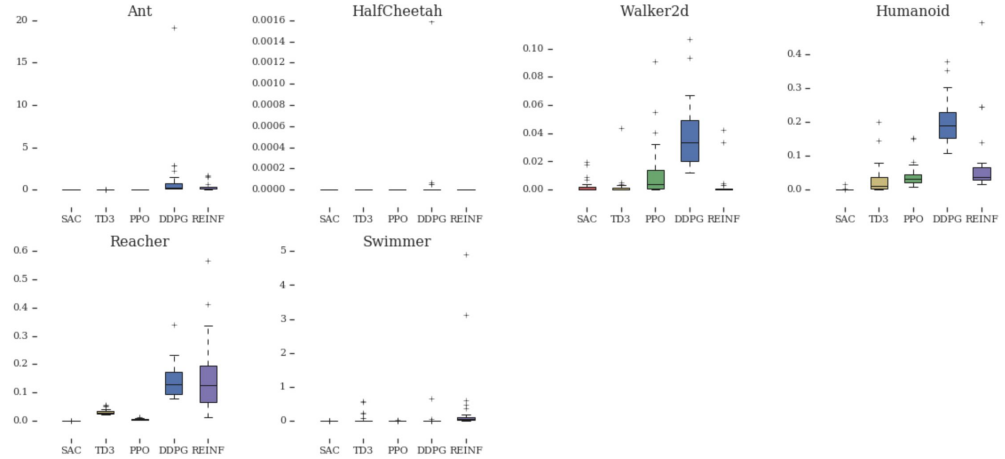
Metric results are shown on a per-task basis in Figs. 6 to 8 for the OpenAI Gym MuJoCo tasks, and Figs. 9 to 23 for the Atari environments. Note that because we are no longer aggregating across tasks in this analysis, we do not need to convert the metric values to rankings.



(a) Dispersion across Time. Better reliability is indicated by less positive values. The x-axes indicate the number of environment steps.

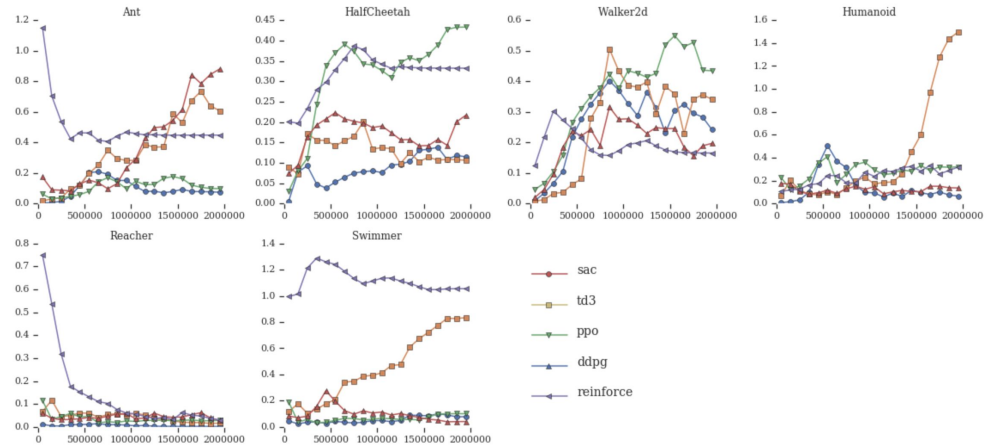


(b) Short-term Risk across Time. Better reliability is indicated by more positive values.

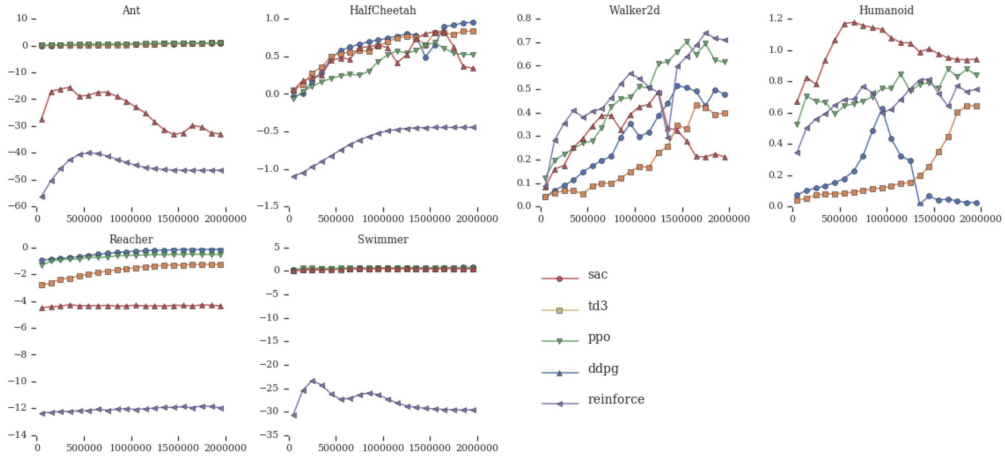


(c) Long-term Risk across Time. Better reliability is indicated by less positive values.

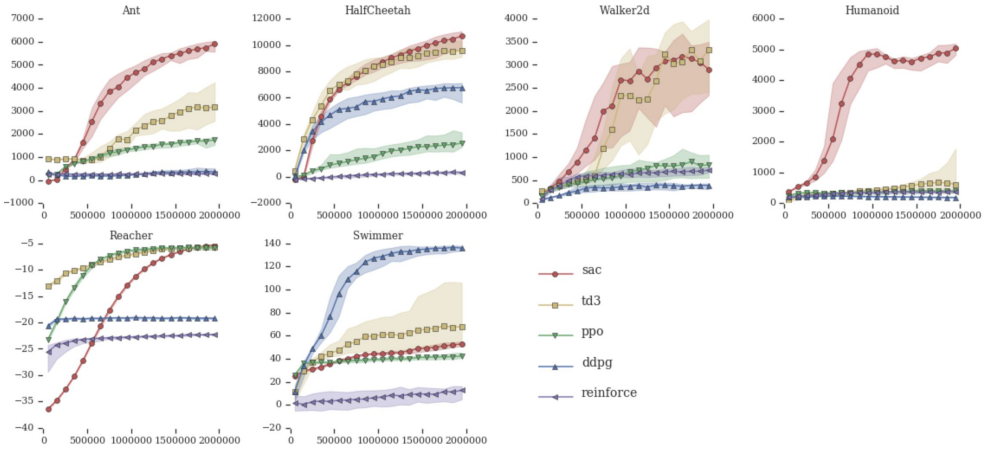
Figure 6: Across-time reliability metrics for continuous control RL algorithms tested on OpenAI Gym environments, evaluated on a per-environment basis.



(a) Dispersion across Runs. Better reliability is indicated by less positive values.

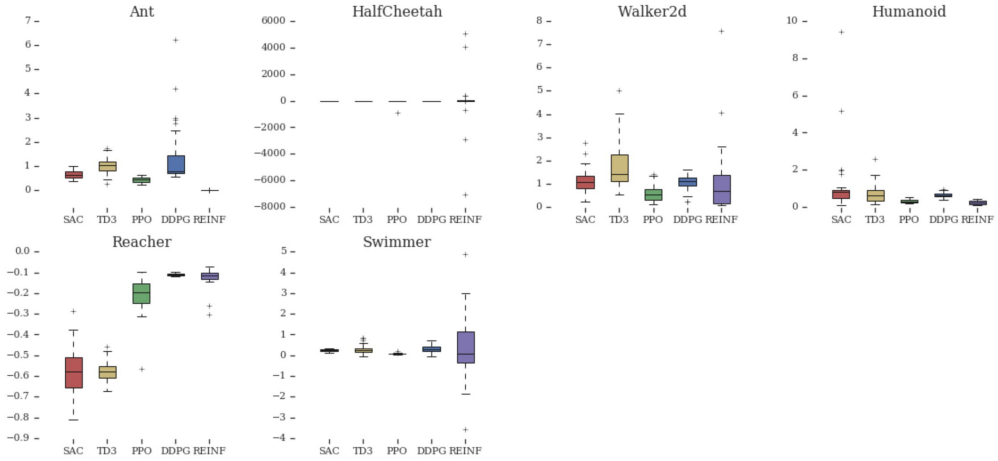


(b) Risk across Runs. Better reliability is indicated by more positive values.

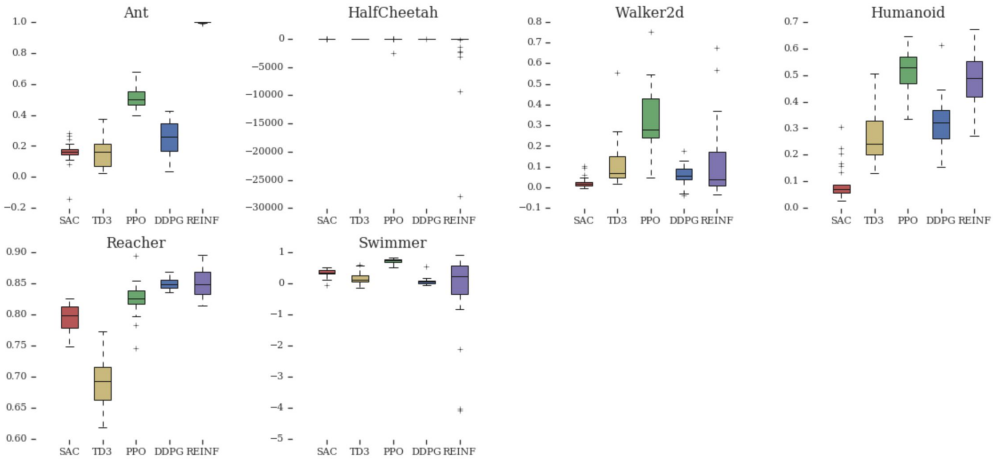


(c) Median performance during training. Better performance is indicated by more positive values.

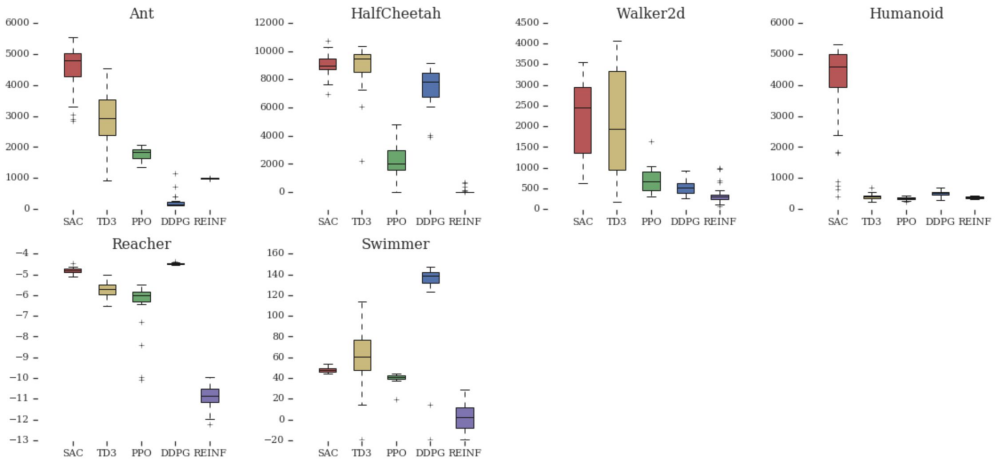
Figure 7: Across-run reliability metrics and median performance for continuous control RL algorithms tested on OpenAI Gym environments, evaluated on a per-environment basis. The x-axes indicate the number of environment steps.



(a) Dispersion on Fixed-policy rollouts. Better reliability is indicated by less positive values.



(b) Risk on Fixed-policy rollouts. Better reliability is indicated by more positive values.



(c) Median performance on Fixed-policy rollouts. Better performance is indicated by more positive values.

Figure 8: Reliability metrics and median performance on fixed-policy rollouts for continuous control RL algorithms tested on OpenAI Gym environments, evaluated on a per-environment basis.

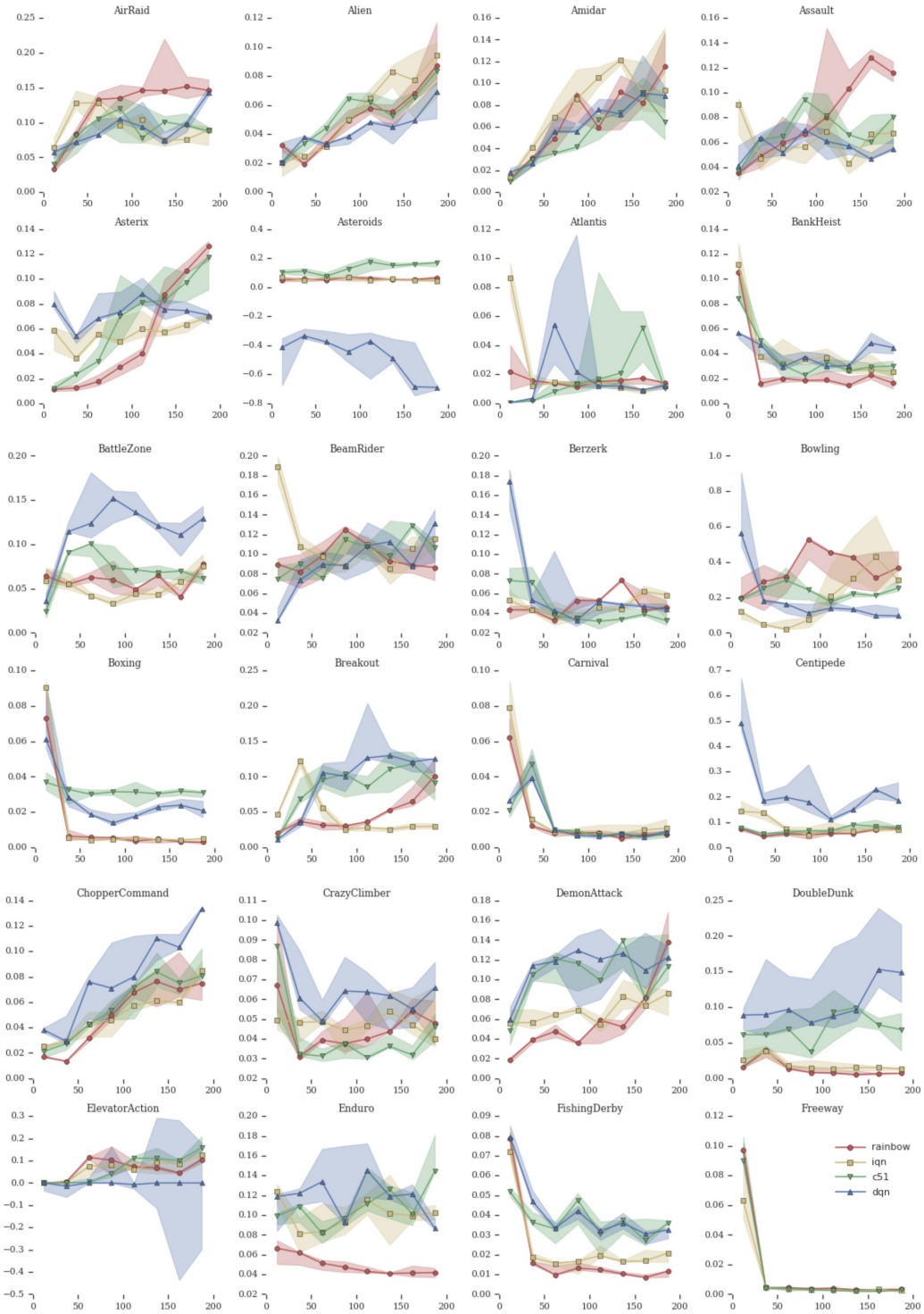


Figure 9: Dispersion across Time for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 1). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

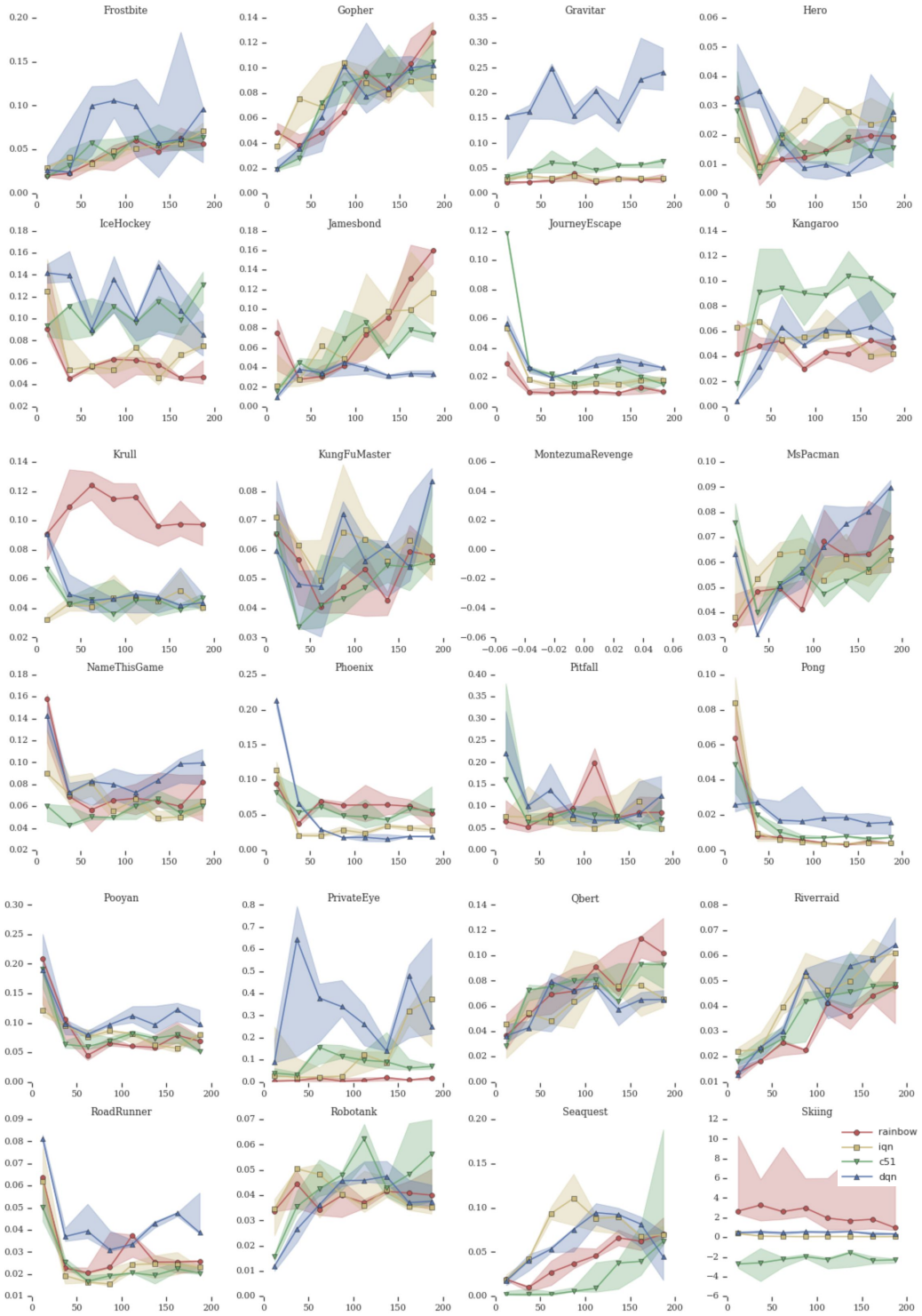


Figure 10: Dispersion across Time for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 2). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

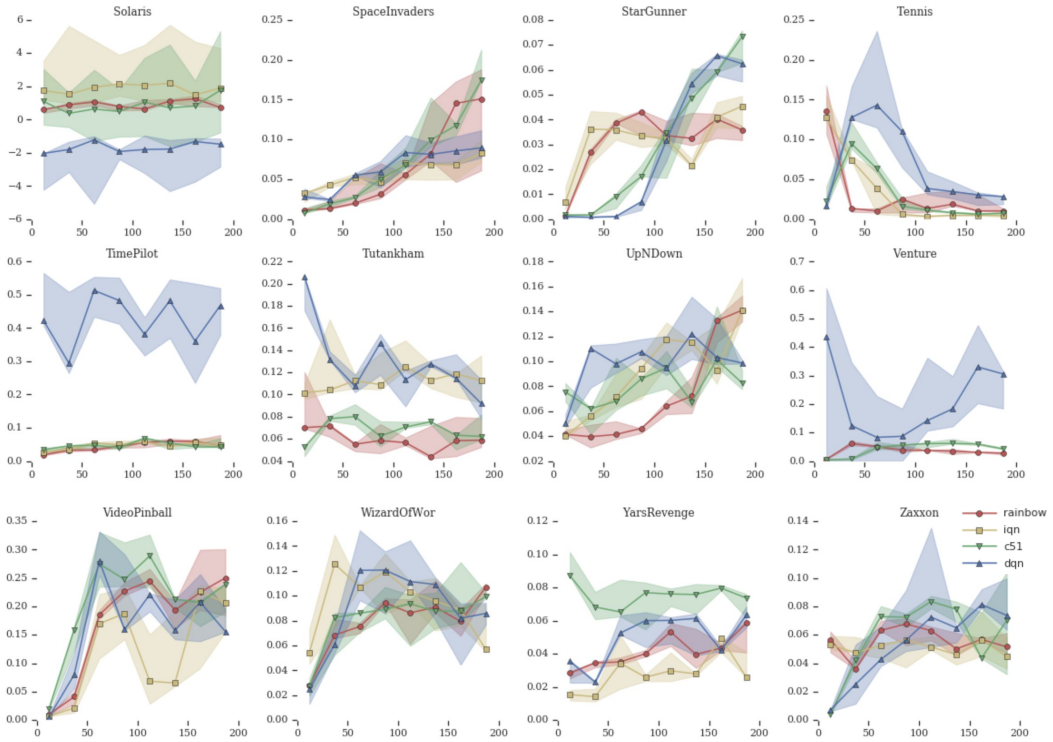


Figure 11: Dispersion across Time for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 3). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

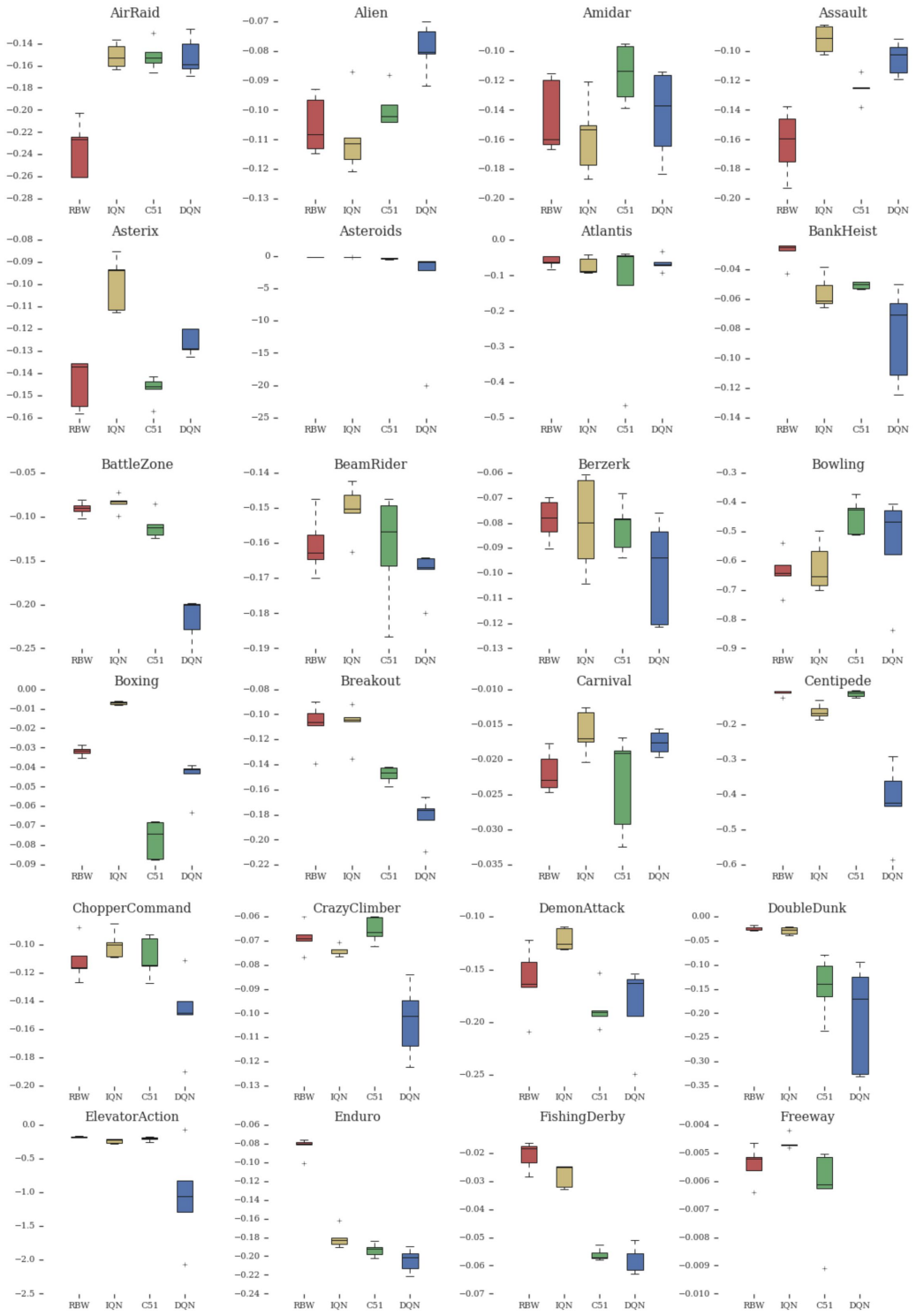


Figure 12: Short-term Risk across Time for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 1). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

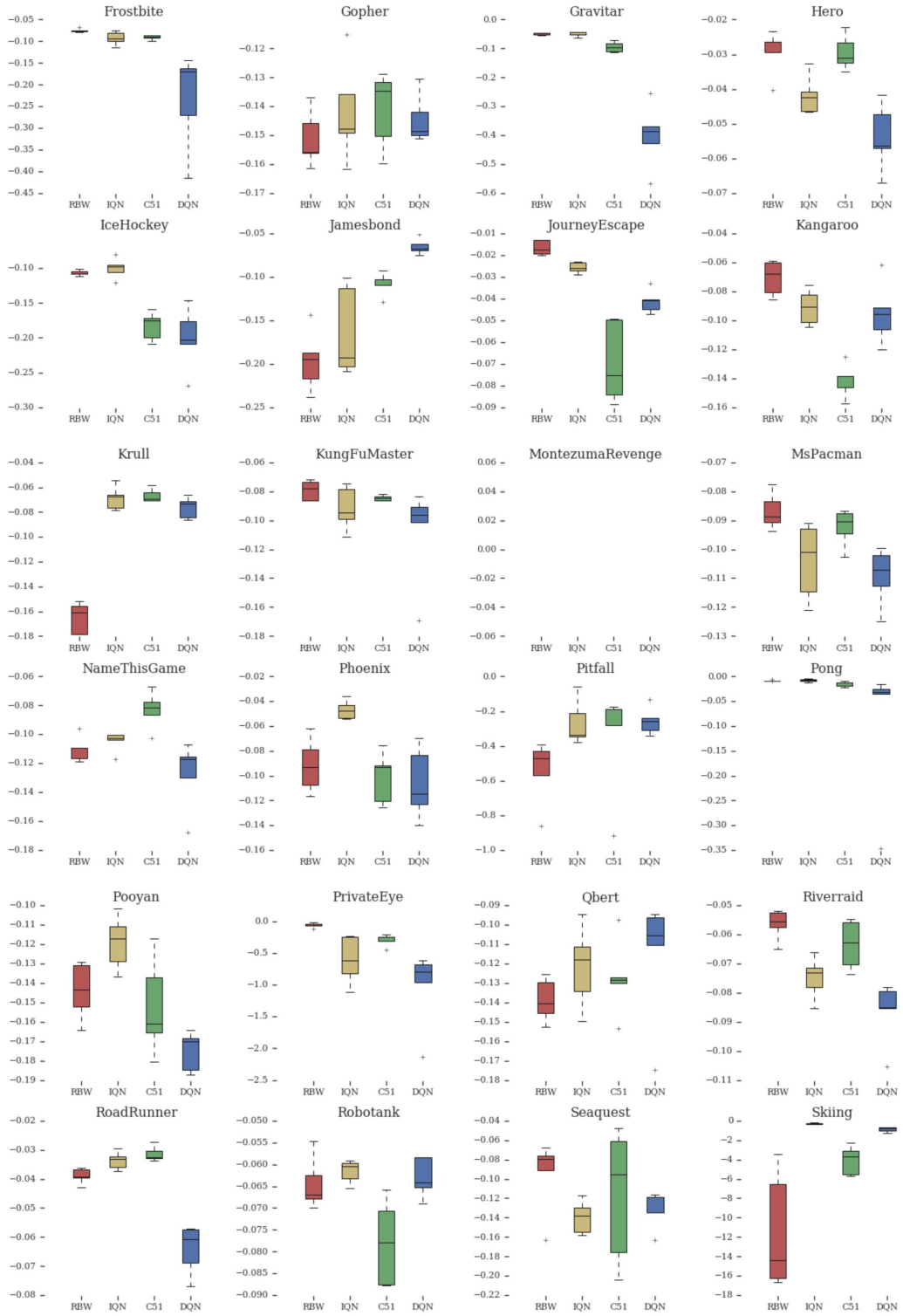


Figure 13: Short-term Risk across Time for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 2). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

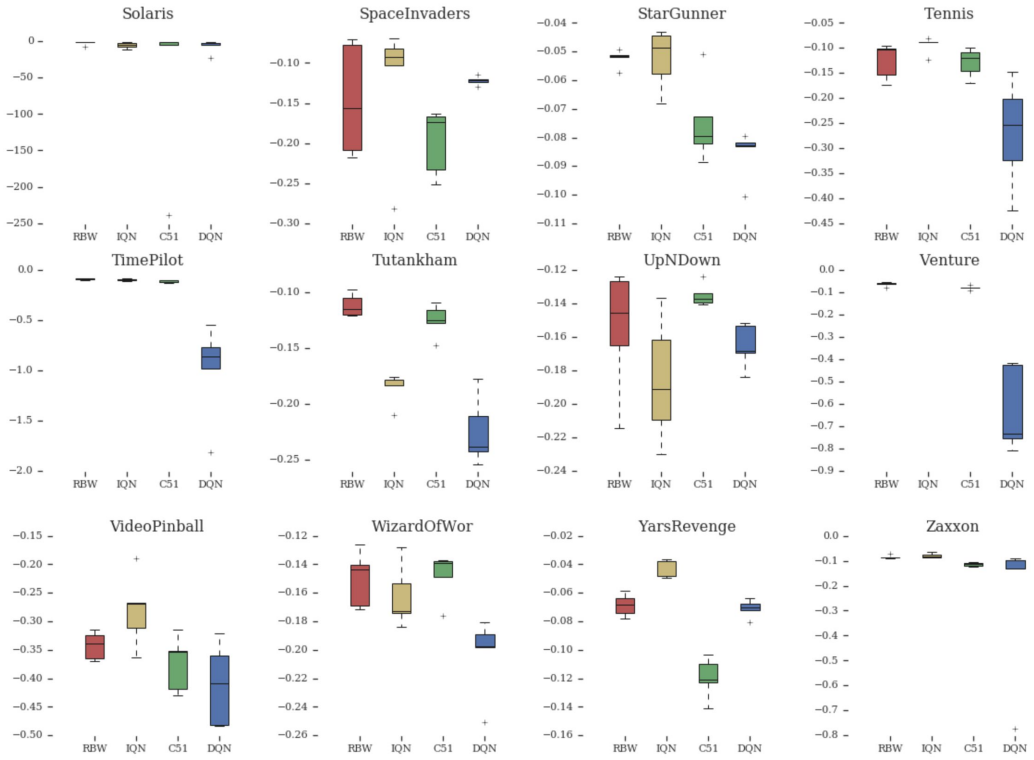


Figure 14: Short-term Risk across Time for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 3). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

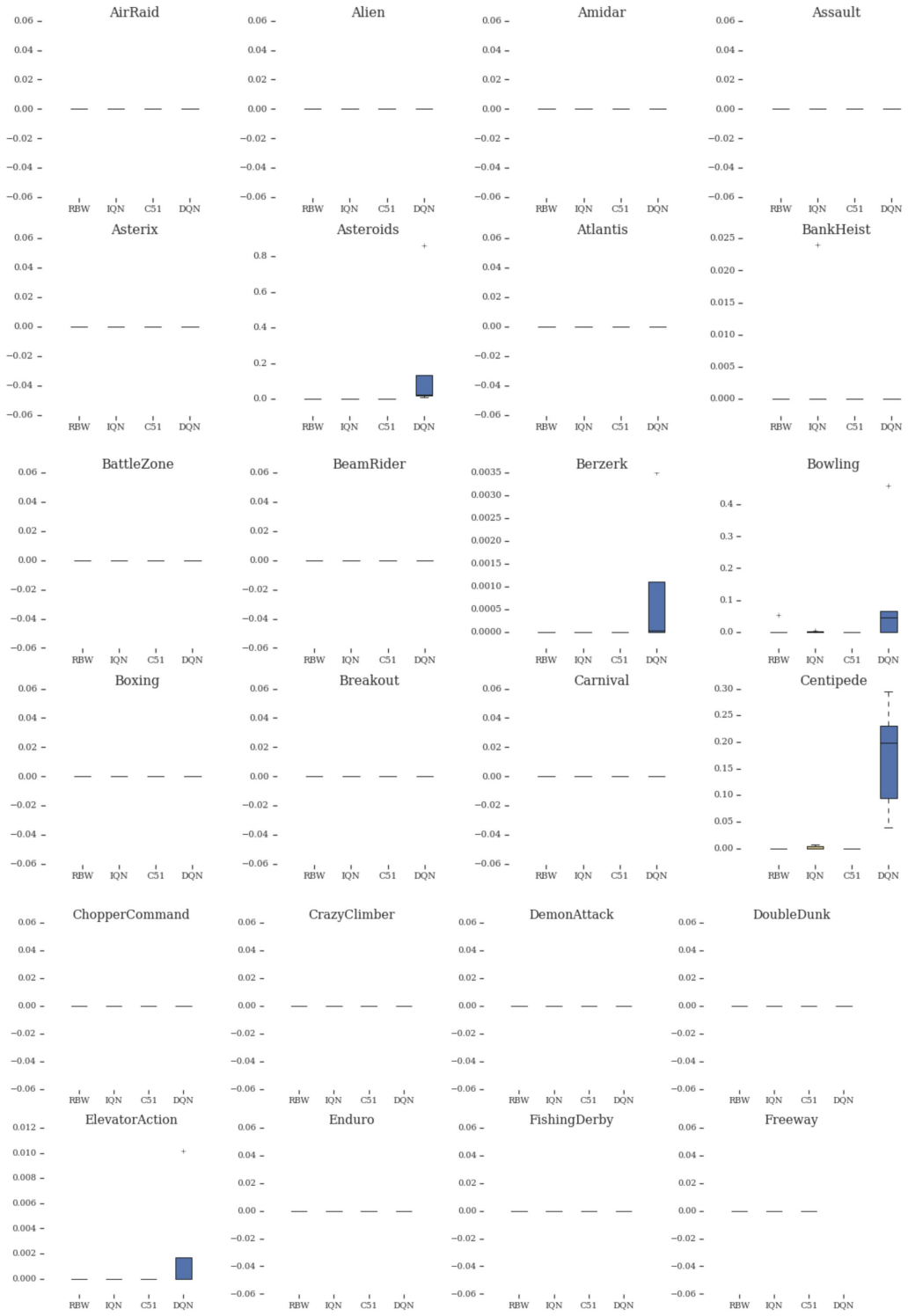


Figure 15: Long-term Risk across Time for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 1). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

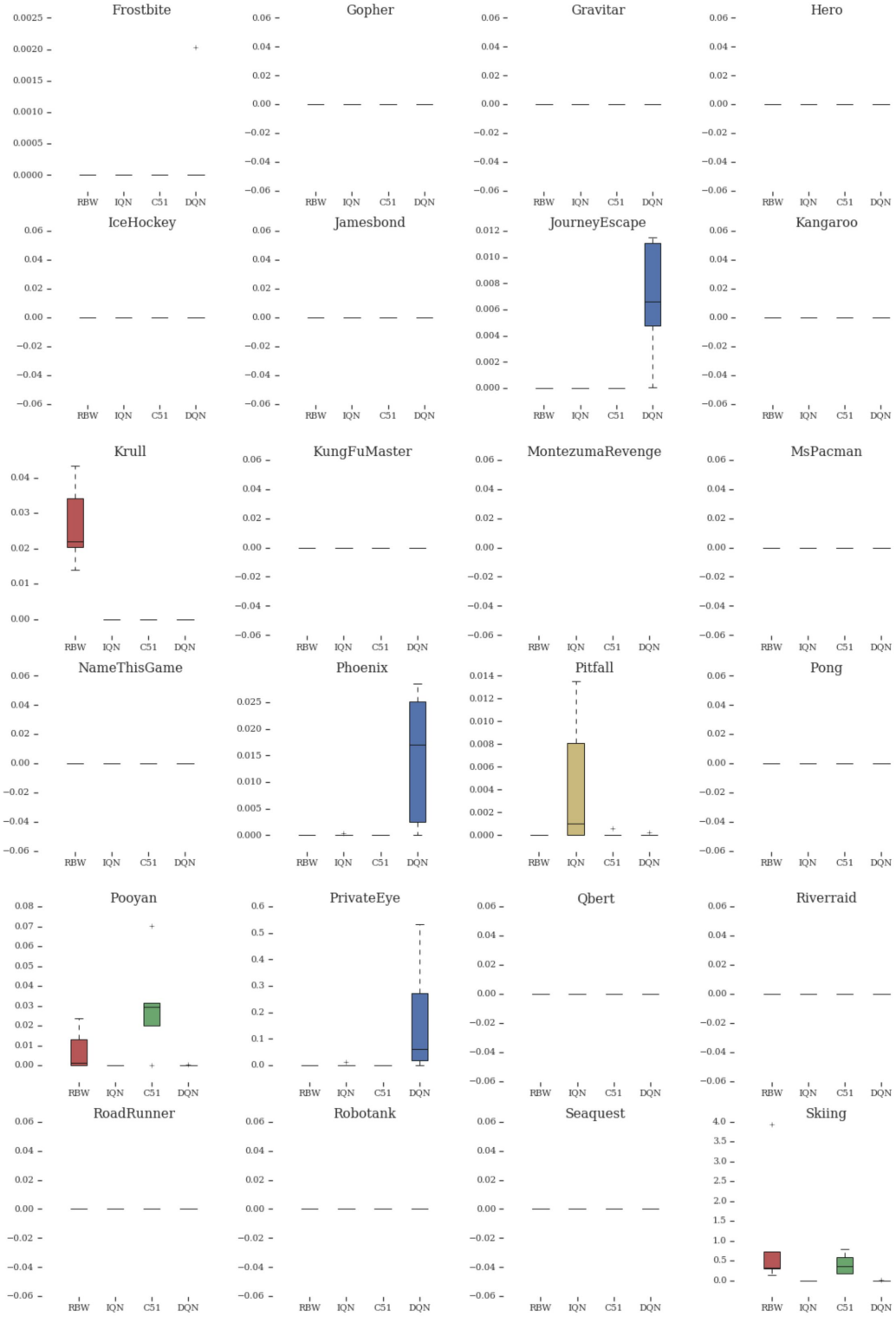


Figure 16: Long-term Risk across Time for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 2). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

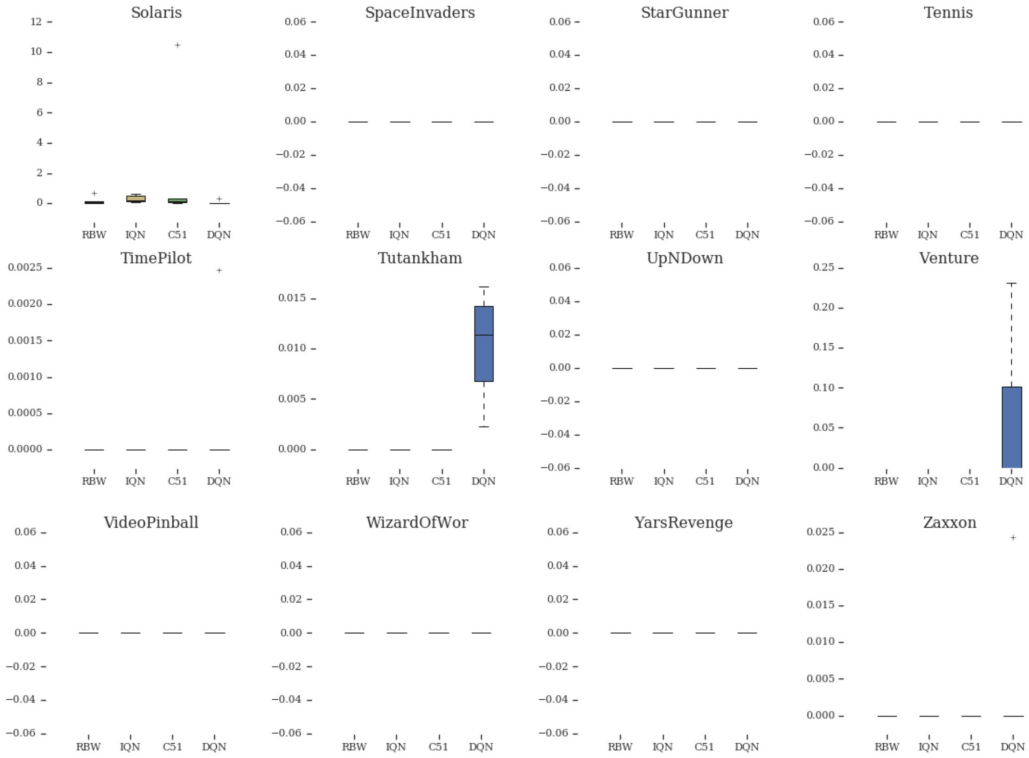


Figure 17: Long-term Risk across Time for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 3). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

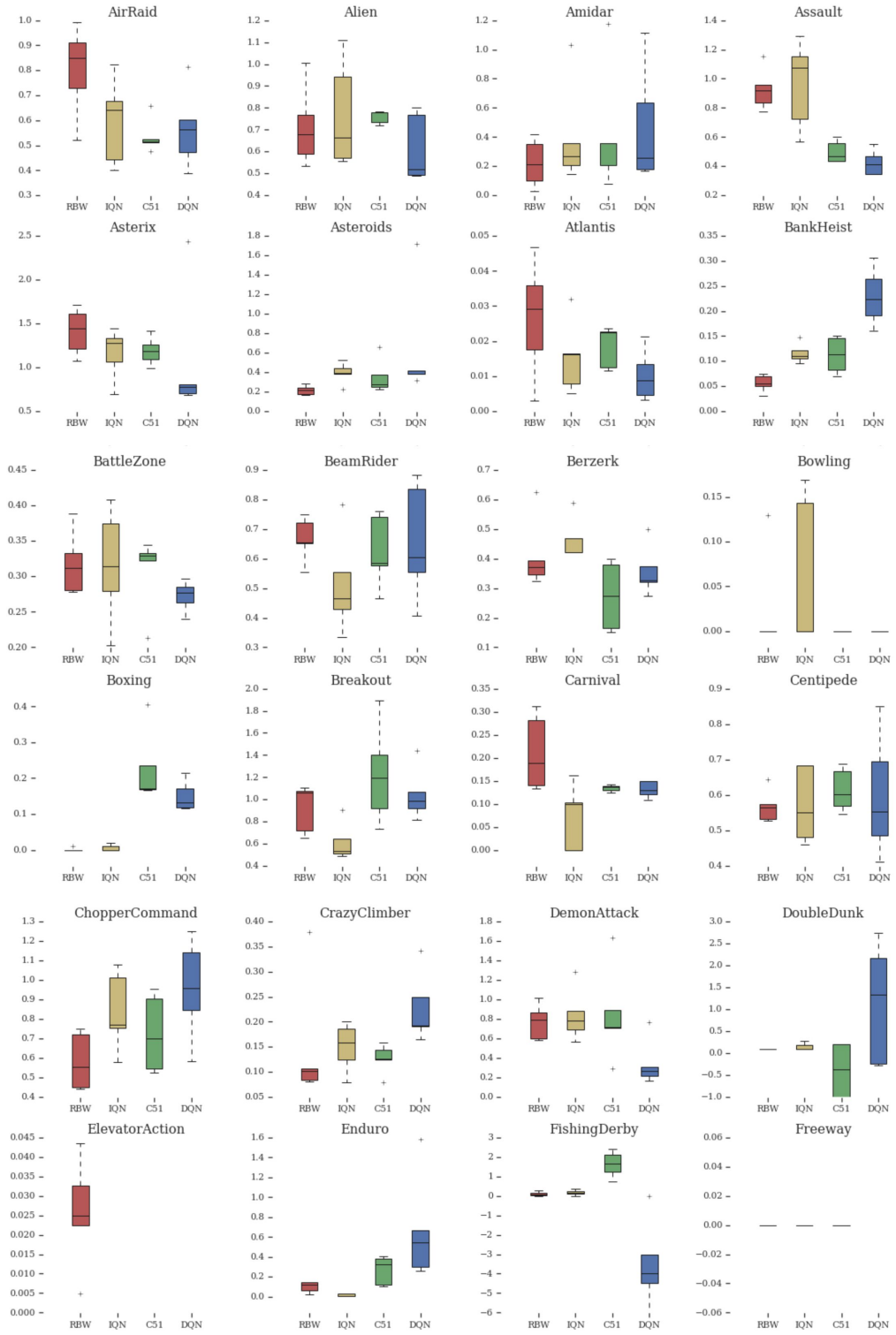


Figure 18: Dispersion across Fixed-policy Rollouts for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 1). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

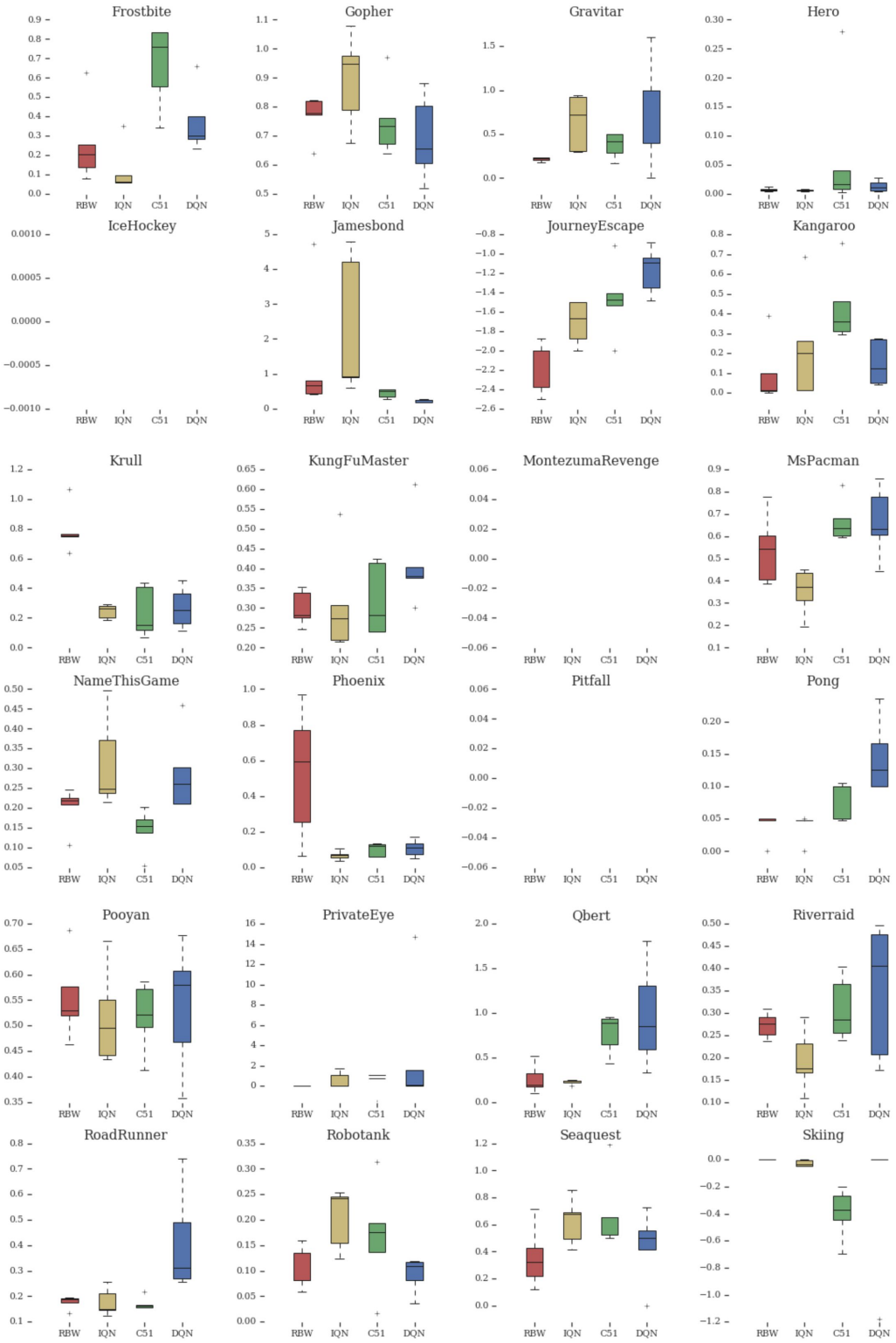


Figure 19: Dispersion across Fixed-policy Rollouts for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 2). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

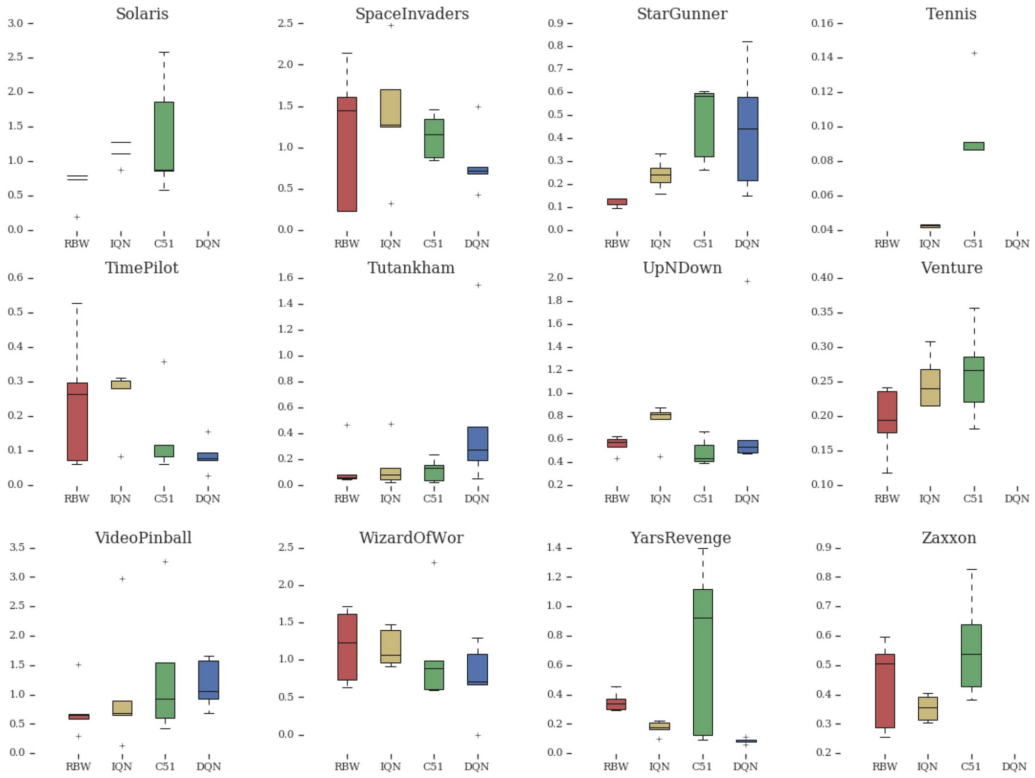


Figure 20: Dispersion across Fixed-policy Rollouts for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 3). Better reliability is indicated by less positive values. The x-axes indicate millions of Atari frames.

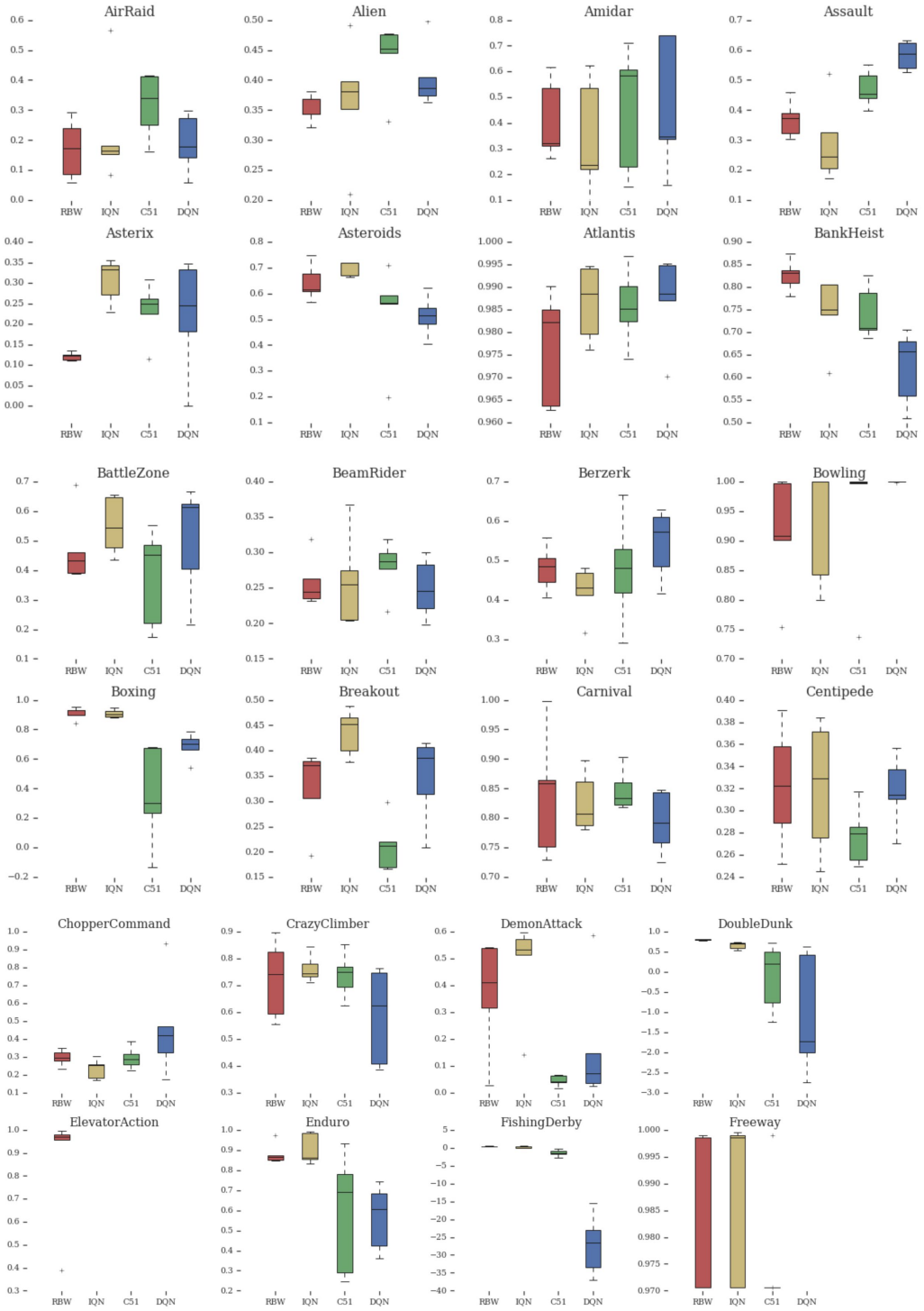


Figure 21: Risk across Fixed-policy Rollouts for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 1). Better reliability is indicated by more positive values. The x-axes indicate millions of Atari frames.

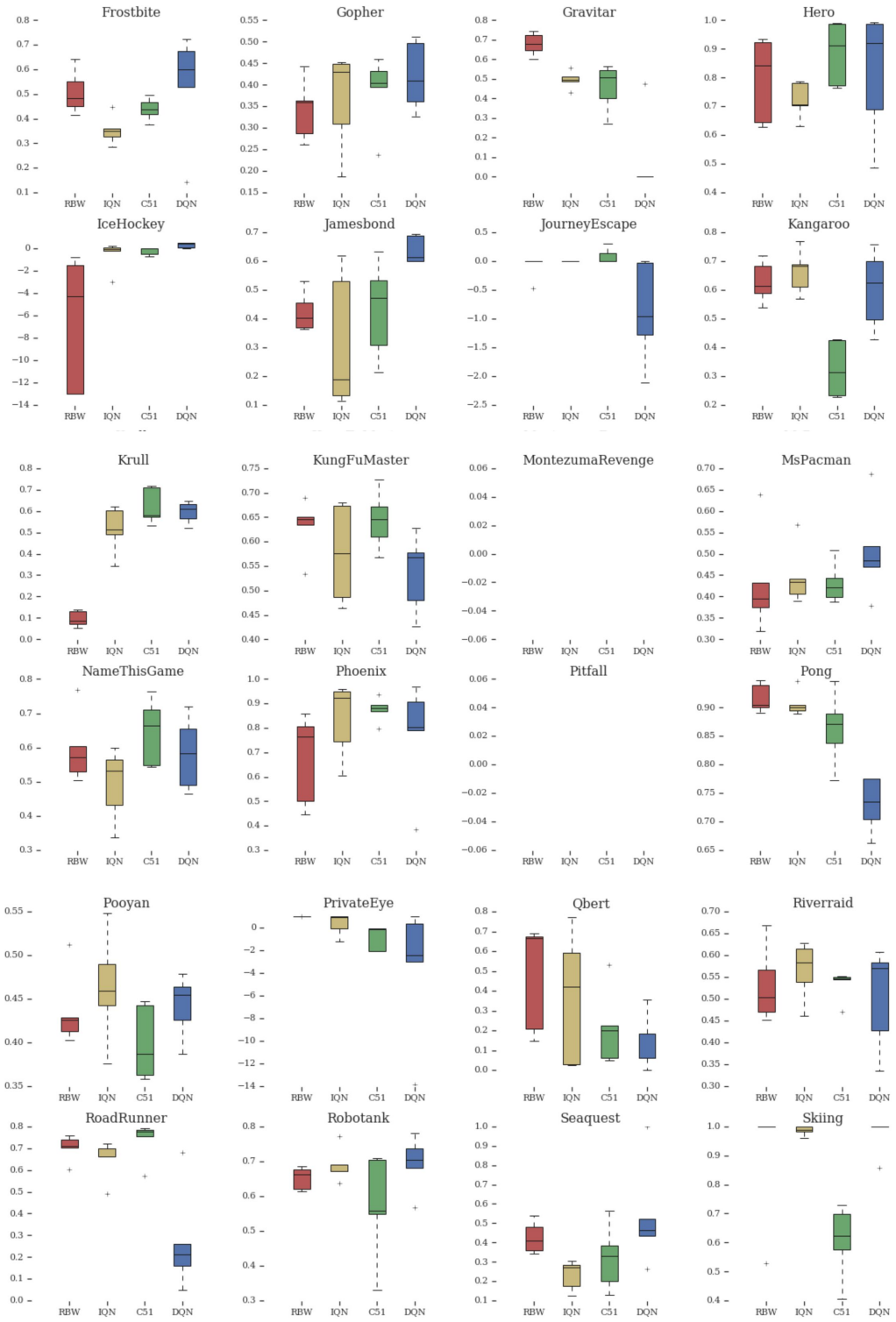


Figure 22: Risk across Fixed-policy Rollouts for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 2). Better reliability is indicated by more positive values. The x-axes indicate millions of Atari frames.

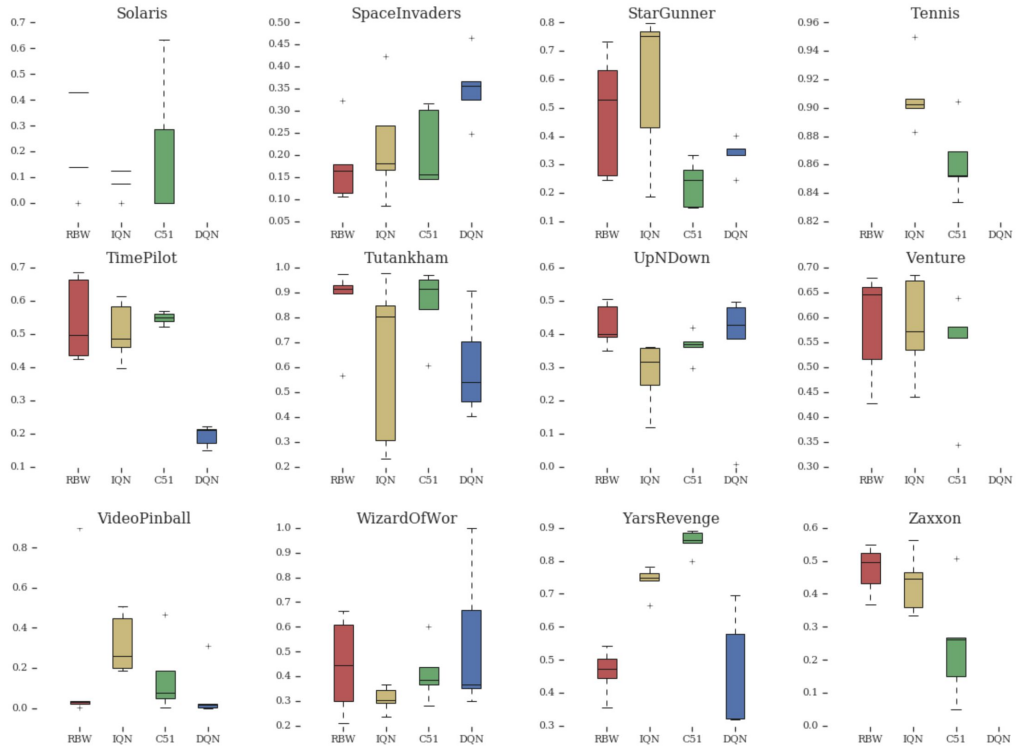


Figure 23: Risk across Fixed-policy Rollouts for DQN-variants tested on 60 Atari games, evaluated on a per-environment basis (page 3). Better reliability is indicated by more positive values. The x-axes indicate millions of Atari frames.

PHILOSOPHICAL TRANSACTIONS A

royalsocietypublishing.org/journal/rsta

Research

Cite this article: Abdullah Hanif M, Shafique M. 2019 SalvageDNN: salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping. *Phil. Trans. R. Soc. A* **378**: 20190164. http://dx.doi.org/10.1098/rsta.2019.0164

Accepted: 22 November 2019

One contribution of 13 to a theme issue 'Harmonizing energy-autonomous computing and intelligence'.

Subject Areas:
artificial intelligence, computer-aided design, computer vision

Keywords:
neural networks, reliability, energy-efficiency, hardware, accelerators, reliable computing

Author for correspondence:
Muhammad Abdullah Hanif
e-mail: muhammad.hanif@tuwien.ac.at

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

THE ROYAL SOCIETY PUBLISHING

SalvageDNN: salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping

Muhammad Abdullah Hanif and Muhammad Shafique

Technische Universität Wien (TU Wien), Vienna, Austria

MS, 0000-0002-2607-8135

Deep neural networks (DNNs) have proliferated in most of the application domains that involve data processing, predictive analysis and knowledge inference. Alongside the need for developing highly performance-efficient DNN accelerators, there is an utmost need to improve the yield of the manufacturing process in order to reduce the per unit cost of the DNN accelerators. To this end, we present 'SalvageDNN', a methodology to enable reliable execution of DNNs on the hardware accelerators with permanent faults (typically due to imperfect manufacturing processes). It employs a fault-aware mapping of different parts of a given DNN on the hardware accelerator (subjected to faults) by leveraging the saliency of the DNN parameters and the fault map of the underlying processing hardware. We also present novel modifications in a systolic array design to further improve the yield of the accelerators while ensuring reliable DNN execution using 'SalvageDNN' and negligible overheads in terms of area, power/energy and performance.

This article is part of the theme issue 'Harmonizing energy-autonomous computing and intelligence'.

1. Introduction

Deep neural networks (DNNs) are widely used in most of the application domains including computer vision, data mining, healthcare, robotics and language processing [1]. While these networks offer state-of-the-art accuracy for many applications, they are highly compute-intensive and energy hungry. To address these

© 2019 The Author(s) Published by the Royal Society. All rights reserved.

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

challenges, a large body of work has been carried out in building DNN accelerators for improving the performance and energy-efficiency of the DNN-based applications [2]. A few prominent accelerator designs are the TPU [3], the MPNA [4] and the Eyeriss [5].

The hardware chips (i.e. in the case of this work, the DNN accelerators) are fabricated using nanometre CMOS technologies, which require a highly sophisticated manufacturing process. The imperfections in the process result in defects in the fabricated chips. These defects can take a wide variety of forms, from permanent faults (e.g. stuck-at faults) that affect the functionality of the chips to variations that affect just the operating characteristics of the hardware (e.g. timing errors can occur if proper variability-driven voltage and frequency guardbands are not provided) [6,7]. Moreover, technology scaling, which has played a vital role in improving the performance and efficiency of the hardware, results in increased fault rates related to both permanent and transient faults. Prior works have highlighted that permanent faults in DNN accelerators significantly affect the accuracy of the DNNs [8]. These faults can be detected using post-fabrication testing and discarding every chip with a permanent fault affects the yield and thereby increases the per-unit-cost of the chips [8].

Improving the yield is one of the foremost challenges in reducing the per-unit-cost of the DNN accelerators, which is specifically important for devices manufactured using smaller technology nodes [9]. The severity of this challenge, in general, increases with an increase in the size of the chip, aggressive technology scaling and the integration density [9]. Therefore, in the current nanometre regime, where the number of processing elements (PEs)/cores per chip has drastically increased to meet the performance requirements of the applications, many integrated circuit (IC) manufacturing companies sell their faulty chips under low performance categories after hiding the faults and without revealing this information to the end-users. Two of the most prominent techniques that are widely studied for yield enhancement are:

— *Product binning*. This approach is mainly used to categorize the manufactured products based on their characteristics observed during post-fabrication testing [10,11]. The binning process is used to reduce the large variances in the characteristics by sorting the products into categories with smaller variances. This process is commonly used by chip manufacturers like Intel and AMD to improve the yield of their manufacturing process [12,13].
— *Hardware redundancy*. In this approach, spare components are introduced in the design in order to replace the defective ones detected during post-fabrication testing [10,14,15].

To this end, Kim *et al.* in [16] and Kung *et al.* in [17] proposed techniques for enhancing the fabrication yield at the cost of performance. Recently, a Fault-Aware Pruning (FAP) method was proposed by Zhang *et al.* in [8] to cater for the impact of permanent faults in DNN accelerators while maintaining the performance. The method works on the principle of dropping the computations, which are mapped on faulty MAC units by simply *bypassing* them. Note that these faults can be detected and the fault maps can be obtained by an initial BIST (built-in-self-test) like method [18,19]. Zhang *et al.* in [8] also proposed hardware modifications in which the computations of individual MAC units in a DNN accelerator can be bypassed to support the FAP method. Further, they proposed a Fault-Aware Pruning + Training (FAP+T) method which allows the fine-tuning of a model given the fault map of the hardware. However, these methods 'tend to either be ineffective or lack efficiency' in situations where any of the following conditions hold.

(i) *The additional circuitry used for bypassing the MAC units is faulty*. In this case the faults will propagate to the output of the hardware and will result in DNN accuracy degradation.
(ii) *The number of chips manufactured is large and the number of computational units in the DNN accelerator design is also significant.* In this case, the number of faulty chips can be large with each having a distinct fault map. For example, figure 1*a* shows the possible number of distinct fault maps that can exist for different systolic array sizes and figure 1*b* shows the number of possible fault maps for different array sizes having the total number of

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

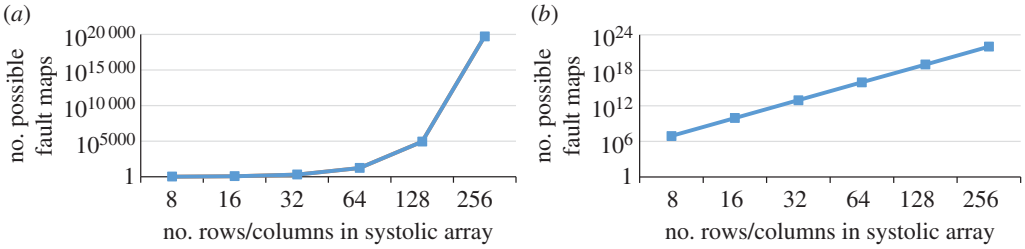


Figure 1. Trends illustrating the relation between the possible number of fault maps versus the number of rows/columns in a systolic array: (*a*) when as many as the number of PEs in the array can be faulty and (*b*) when at maximum five PEs can be faulty. (Online version in colour.)

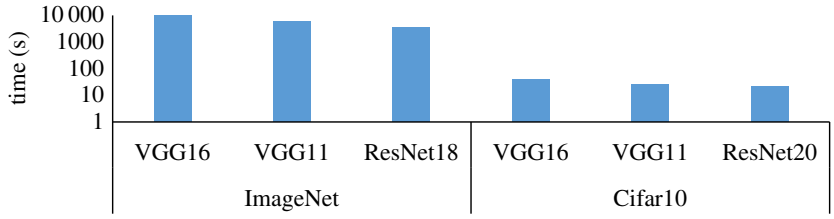


Figure 2. Time required for training different DNNs[1] (i.e. VGG16, VGG11, ResNet18 and ResNet20) for one epoch with different datasets (i.e. ImageNet and Cifar-10) on a Core i7 machine with one GTX1080Ti. (Online version in colour.)

faulty PEs less than or equal to five. As can be seen from the figure, with an increase in the size of the systolic array, the number of possible fault maps increases significantly and is greater than the number of chips that are usually manufactured of the same type. In this case, fine-tuning/training a neural network for each faulty chip seems impractical, as DNN fine-tuning/training is extremely time- and resource-consuming, specifically for deeper and complex NNs, as shown for different DNNs and datasets in figure 2.

(iii) *The training dataset is not available*, e.g. in a case where a company has released a neural network model, but they have not made the training data available.

(iv) Moreover, the above method will lack efficiency in case the number of permanent faults changes over the lifetime of the chip, e.g. due to ageing. In that case, excessive re-training cannot be done, and simply bypassing the respective computations may result in accuracy loss beyond acceptable range.

Our novel contributions: To address the aforementioned limitations, in this work, we propose:

(i) A methodology, *SalvageDNN*, for fault-aware mapping of a DNN based on the saliency (importance) characteristics of its filters/neurons. We propose and study different mapping strategies which can maintain the accuracy of the DNNs at high fault rates without the need for time- and resource-consuming fine-tuning/re-training processes. This also makes our approach training data agnostic.

(ii) Different hardware enhancements which enable us to *safely* bypass the processed computations of the faulty hardware, even if the bypass circuity of some MAC units is faulty. The designs will also allow us to trade-off between the area and the power overheads versus the accuracy loss that is experienced in case of such hardware faults.

Paper organization: In §2, we present the required preliminaries that are important for understanding the concept presented in the later sections of the paper. In §3, we present our

[1]For the analysis, we used the DNNs available in the *Neural Network distiller* [20] tool.

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

methodology for salvaging DNN accelerators using fault-aware mapping. Section 4 presents the results and comparison with the state of the art. Finally, §5 concludes the paper.

2. Preliminaries and background

In this section, we present a brief overview of the DNNs, the relevant methods for evaluating the saliency of filters/neurons in a NN, and the baseline DNN accelerator design used in this work.

(a) Deep neural networks

DNNs are composed of neurons that are arranged in the form of layers to progressively extract higher order features from the inputs, as shown in figure 3*a*. The basic operation performed by a neuron is a weighted sum of inputs which is then passed through an activation function to introduce non-linearity in the model. This can mathematically be written as:

$$A_i^{\langle l\rangle} = f^{\langle l\rangle}\Big(\sum_j W_{(i,j)}^{\langle l\rangle} \times A_j^{\langle l-1\rangle} + b_i^{\langle l\rangle}\Big). \tag{2.1}$$

Here, $W_{(i,j)}^{\langle l\rangle}$ represents the weight of the connection between the ith neuron of the lth layer and the jth neuron of the $l-1$th layer, $b_i^{\langle l\rangle}$ represents the bias associated with the ith neuron, $A_i^{\langle l\rangle}$ represents the output (commonly known as activation) generated by the ith neuron, and $f^{\langle l\rangle}(.)$ represents the activation function used in the lth layer. The activation function can be any nonlinear function. However, the common types of activation functions used in state-of-the-art neural networks are Sigmoid, Tanh and RELU. Figure 3*a* shows an example of the fully-connected type of neural networks that can be represented using this notation. The number of neurons in the lth layer is represented by $N^{\langle l\rangle}$ and the dimension of weights of the lth fully-connected layer can be given as $N^{\langle l-1\rangle} \times N^{\langle l\rangle}$. A neural network is considered deep (i.e. a DNN) if the number of layers is typically 3 or more [2].

A widely used type of DNN is the Convolutional Neural Network (CNN) which is based on convolutional layers. The basic operation performed in these layers is a convolution operation. The weights in these layers are arranged in the form of 3D filters which are traversed across the 2D space of the input activations to generate the outputs. Each filter is responsible for generating one feature map, and all the feature maps (generated using the filters of a layer) when combined form the input of the subsequent layer. The dimensions of the weights of the lth convolutional layer can be given as $F_{rc}^{\langle l\rangle} \times F_{rc}^{\langle l\rangle} \times C^{\langle l\rangle} \times N^{\langle l\rangle}$. Here, $N^{\langle l\rangle}$, $C^{\langle l\rangle}$ and $F_{rc}^{\langle l\rangle}$ represent the number of filters, the number of channels in each filter and the number of rows/columns of the filters of the lth layer, respectively. A specific weight in the lth convolutional layer can be represented as $W_{\{(r,c),(C,F)\}}^{\langle l\rangle}$, where r, c, C and F represent the row, the column, the channel and the filter number, respectively. Similarly, the dimensions of the output activations of the lth layer can be given as $x^{\langle l\rangle} \times y^{\langle l\rangle} \times N^{\langle l\rangle}$, where $x^{\langle l\rangle}$ and $y^{\langle l\rangle}$ represent the number of rows and columns in the output feature maps and $N^{\langle l\rangle}$ defines the number of feature maps, which is equivalent to the number of filters, in the lth layer. An example illustration of a convolutional layer is shown in figure 3*b*. A more detailed description of the DNNs can be found in [1,2].

(b) Deep neural network accelerator

Due to the compute intensive nature of DNNs, several accelerator designs have been proposed in the literature to accelerate the inference process of DNNs. These accelerators are mainly capable of performing efficient vector and matrix multiplication operations, which are the fundamental operations in the DNN inference. This is achieved by using multiple computational units which operate in parallel and by enabling local data sharing/reuse in these units.

The Tensor Processing Unit (TPU) [3] is one of the prominent DNN accelerators developed by Google for addressing the increased cloud-based DNN related workloads. At the core of the

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

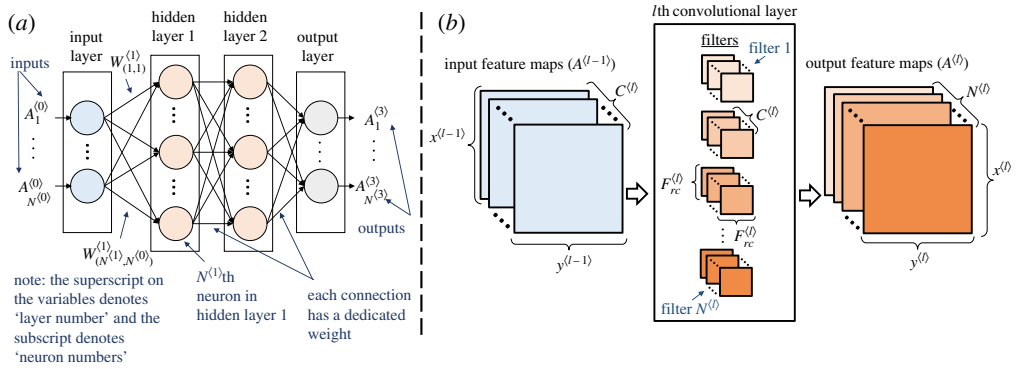


Figure 3. (*a*) An illustration of a fully-connected neural network. (*b*) An illustration of a convolutional layer used in convolutional neural networks. (Online version in colour.)

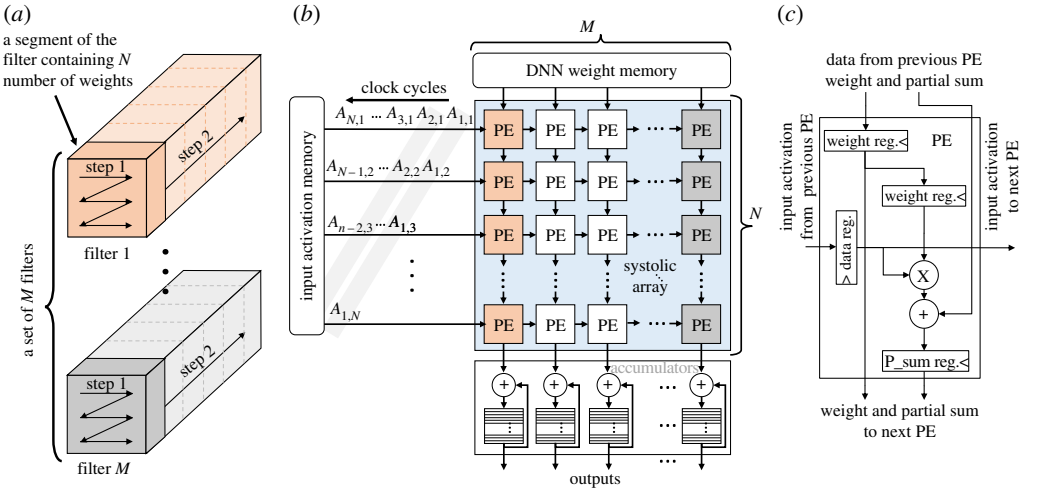


Figure 4. (*a*) Considered filter/neuron unrolling policy for mapping weights to the DNN hardware accelerator. The steps show the sequence of unrolling and mapping of the weights onto the array. (*b*) Our baseline systolic array-based DNN accelerator (similar to the design of well-know systolic arrays like Google TPU [3] and Eyeriss [5]), illustrating the mapping of a segment of filters highlighted in (*a*). (*c*) Detailed design of the processing element (PE). (Online version in colour.)

TPU is a systolic array composed of processing elements (PEs) that are connected in a 2D grid like manner. The design of our baseline systolic array is similar to that of the TPU architecture and is illustrated in figures 4*b*,*c*. Figures 4*a*,*b* show the dataflow where unrolled segments of a set of filters/neurons are mapped on columns of the systolic array, and then the vectors of input activations are fed to the array to perform the dot-product operations. The PEs work in lock steps with their neighbouring PEs to generate the output. For example, in the first clock cycle, the top left corner PE will multiply the first weight with the first input activation and then pass the partial output to its downstream neighbour and the input activation to the PE on its right side. The downstream PE (i.e. the second PE in the first column), in the next clock cycle, computes the product of the second weight with the second activation and adds the available partial product from the upstream PE to generate the partial product for its downstream PE. In the meantime, the first PE in the first column will generate the first partial product related to the second input activation vector and the first PE in the second column will generate the first partial product

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

related to the first input activation vector. By continuing this procedure, the result of the first dot-product from the systolic array will be available after N clock cycles and, at peak, the array can generate one result per clock cycle per column.

Note: If the number of weights in the filters/neurons is more than the number of rows in the array, the weights cannot be mapped to the array at the same time, as highlighted with the help of figures 4*a*,*b*. In such cases, the results generated by the array are not complete, and the accumulation units connected below the array are responsible for storing the partial products and accumulating them with the rest of the corresponding partial products of the filters/neurons to compute the final outputs.

(c) Saliency evaluation of neurons/filters of a DNN

Different methods have been proposed in literature to estimate the saliency of parameters (i.e. weights, neurons and filters) of a neural network. *The saliency of a network parameter defines its importance based on its contribution and/or the expected impact it can have on the output.* The widely used methods are based on L1 and L2 norms, where the norms of the parameters define their saliency [21]. Another prominent method includes neuron importance score computation based on propagation [22]. Such methods are usually considered more accurate for estimating the importance of the neurons/filters. However, they are computationally more intensive than norm-based methods, as they have to back propagate from the output to a particular neuron/filter to compute its saliency. A detailed explanation of the methods is provided in §3b.

3. SalvageDNNs: a methodology for salvaging DNN accelerators using fault-aware mapping

This section presents our methodology for salvaging DNN accelerators having permanent faults in the datapath using saliency-driven fault-aware mapping, without requiring extensive retraining as typically done in the state of the art. Figure 5 illustrates the main steps involved in the methodology. In step ❶, the DNN accelerator design is optimized such that the components having permanent faults can be disconnected from the main datapath to mitigate the effects of the faults. In step ❷, the saliency of the weights of the DNN is computed. Given a dataflow, the architectural characteristics of the modified accelerator design and the fault map of the fabricated hardware, step ❸ then proposes mapping of neurons/filters of a layer of the DNN on different segments of the hardware such that the sum of saliency of the weights that are pruned (mapped on the faulty/disconnected parts of the datapath) during inference is minimized. Step ❹ makes the required rearrangements in the DNN such that the data rearrangements, which are highly memory intensive, are not required during the DNN processing. Steps ❸ and ❹ are repeated for each layer and the resultant network is forwarded to step ❺ after setting all the weights which are to be mapped on the faulty/disconnected computational units to zero. Note that the network is traversed in a sequential order from the first layer to the last. In step ❺, required adjustments are made to the network parameters to compensate for the pruned weights/computations. The details of these steps are presented in the following subsections.

(a) Hardware design optimization/enhancements for permanent fault mitigation

Figure 6*c* shows the hardware optimization proposed in [8] for mitigating permanent faults, where an additional multiplexer is inserted in the datapath for bypassing the MAC computation in the PEs in case of faults. The baseline systolic array architecture and the design of the conventional processing element are shown in figures 6*a*,*b*, respectively. As permanent faults are randomly distributed in the hardware, they can occur in any hardware component, including the added circuitry. If a fault occurs in the multiplexer, the error cannot be mitigated and, therefore,

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

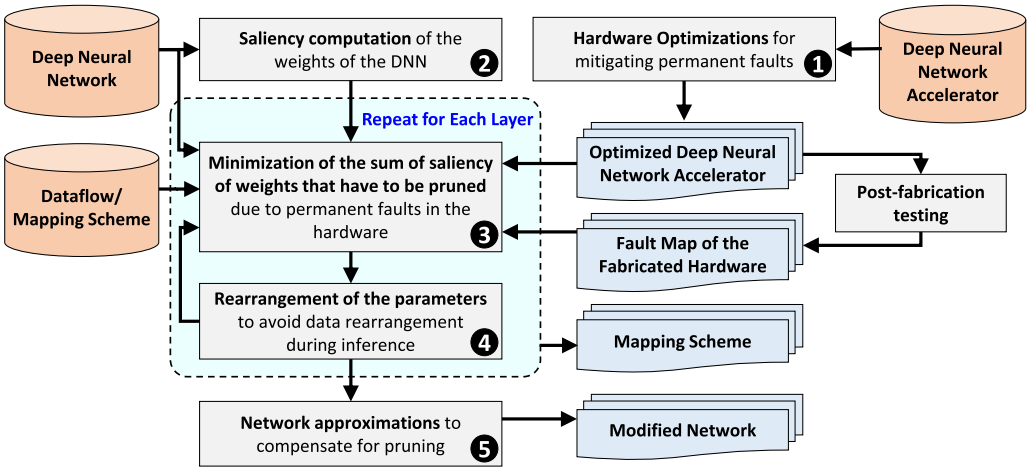


Figure 5. Overview of the proposed SalvageDNN methodology for saliency-driven fault-aware mapping of DNNs on a hardware with permanent faults. (Online version in colour.)

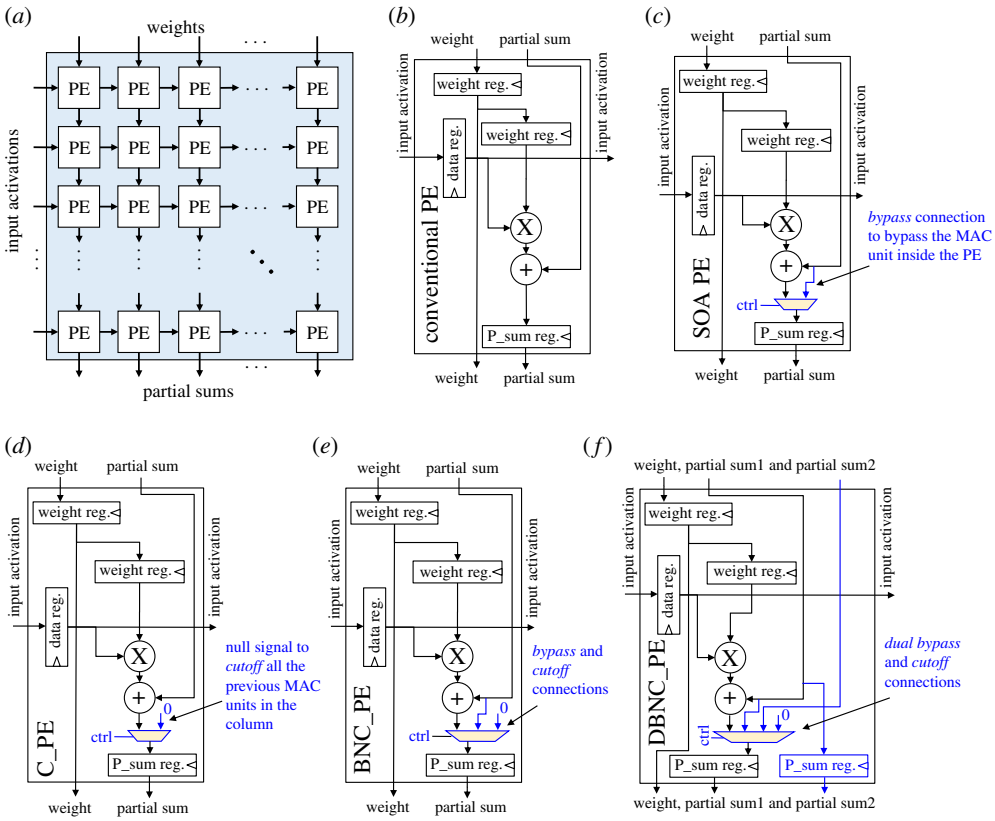


Figure 6. (*a*) The baseline systolic array design proposed in the TPU. (*b*) The conventional PE design. (*c*) The modified PE proposed in [8] for permanent fault mitigation using the FAP and the FAP+T techniques. (*d–f*) Show our novel additional PE designs for handing permanent faults. Note, the changes in the PEs with respect to the conventional PE design, i.e. (*b*), are shown in colour. (Online version in colour.)

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

will propagate to the output. To overcome this limitation, we propose and study different datapath enhancements which can enable us to bypass/disconnect the faulty multiplexers as well.

Figures 6*d–f* show different novel architectural modifications to mitigate the effects of permanent faults. These designs offer different trade-offs between the hardware overhead versus the average amount of circuitry disconnected in case of permanent faults, which directly impacts the accuracy of a DNN at runtime. Figure 6*d* shows the design where, in case of a fault in the MAC unit inside a PE, the complete set of MAC units above and including the MAC in the faulty PE are *cut off*. In case of a fault in the multiplexer of a PE, the MAC unit in the subsequent PE is also disconnected by using its multiplexer. This design will, henceforth, be referred to as C_PE in this paper. The advantage of this design is that the hardware overhead is significantly less. However, the accuracy of the DNNs can drop significantly depending on the locations and the number of faults, due to disconnection of a large number of PEs. Figure 6*e* illustrates a hybrid design which combines the best of both worlds, i.e. in case of a fault in the MAC unit of a PE, the MAC operation can be *bypassed* and, in case of a fault in the multiplexer, the complete set of MAC units including the MAC in the subsequent PE can be *cut off* from the computational flow. This design, from here onward, will be referred to as BNC_PE. The main advantage of this approach is that it can handle faults in the MAC units as well as the multiplexer while maintaining reasonable hardware overhead as compared to the design shown in figure 6*c*.

Figure 6*f* shows another design where an additional connection is made in the BNC_PE to *bypass two adjacent MAC units* in a column. This design also enables us to bypass the intermediate multiplexer in case it is faulty. Similarly, a design can be made where n adjacent MAC units can be bypassed. However, there are two main limitations associated with this approach: (1) the hardware overhead in this case is more as, apart from the larger multiplexer, additional registers are required for partial sums to maintain the computational flow; and (2) if n consecutive PEs have faults in their multiplexers, error can still propagate to the output. Therefore, in this work, we limit n to 2, as shown in figure 6*f*. This design, in the rest of this article, will be referred to as DBNC_PE.

(b) Saliency computation of the DNN parameters

In this article, we study the impact of two different methods that can be employed for saliency computation of the weights of a DNN on the effectiveness of the proposed methodology. Note that the proposed methodology only requires one method at a time for estimating the saliency of the DNN parameters. The methods along with their brief descriptions are as follows.

(i) *Norm-based method*. In this approach, the L1-norm or L2-norm of the weights/neurons defines their relative importance.

(ii) *Importance score propagation-based method*. In this approach the importance of a weight/neuron is computed based on its magnitude as well as the importance of the connections involved in propagating its response to the output of the NN. Let $s_i^{<l>}$ define the saliency of ith neuron/filter in the lth layer of a DNN. The saliency of a weight in the neuron/filter can be computed by multiplying the absolute value of the weight with the saliency, i.e. in the lth fully-connected layer it is computed by $|W_{(i,j)}^{<l>}| \times s_i^{<l>}$ and in the lth convolutional layer it is computed by $|W_{\{(r,c),(C,i)\}}^{<l>}| \times s_i^{<l>}$. The saliency of the neurons/filters in the lth layer of a DNN can be computed by

$$s_i^{<l>} = \sum_k |W_{(k,i)}^{<l+1>}| \times s_k^{<l+1>} \tag{3.1}$$

in case $l+1$th layer is a fully-connected layer, and is computed by

$$s_i^{<l>} = \sum_k \sum_r \sum_c |W_{\{(r,c),(i,k)\}}^{<l+1>}| \times s_k^{<l+1>} \tag{3.2}$$

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

in case $l+1$th layer is a convolutional layer. In this work, the saliency of all the output neurons is defined as equal, i.e. $s_i^{<n>} = 1 \forall i \in \{1, 2, \ldots N^l\}$.

Note, the aforementioned techniques are highly effective for comparing the saliencies of weights from the same layer. As in most of the DNN accelerators, the layers are processed sequentially, and we are only interested in evaluating the relative importance of weights within a layer. Hence, the above the mentioned approaches will be used for evaluating the importance of weights of a NN.

(c) Minimization of the sum of saliency

The objective of SalvageDNN is to map the least important weights on the components that will be disconnected from the datapath in case of occurrence of the permanent faults in the hardware. To achieve this, the following steps are followed for each layer of the network.

— *Generation of the Disconnection Map (DM)*. The DM is a matrix that defines the MAC units which will be disconnected from the array during processing. This is done by setting the values corresponding to the MACs which will be disconnected in DM to 1. For example, if the MAC unit in the first row and first column of the array will be disconnected during processing, the DM(1, 1) is set to 1; otherwise it will be 0. The DM is constructed using the fault map of the array. The fault map is represented using two matrices $\mathrm{FM_{MAC}}$ and $\mathrm{FM_{MUX}}$, where $\mathrm{FM_{MAC}}$ keeps track of the faulty MAC units and $\mathrm{FM_{MUX}}$ keeps track of the faulty multiplexers in the array. Note that the size of DM, $\mathrm{FM_{MAC}}$ and $\mathrm{FM_{MUX}}$ is the same as the size of the array and there is one-to-one mapping between the elements of the matrix and respective hardware components of the systolic array.

— *Unrolling of the Saliencies of Weights of a Layer in a Matrix* $\boldsymbol{S}$. The matrix $\mathbf{S}$ represents the matrix containing the saliency of neurons/filters of a layer in flattened form, i.e. each column in the matrix contains saliency of weights from a neuron/filter that are flattened based on the given dataflow.

— *Generation of the Pruning Matrix (PM)*. The DM matrix is replicated in x and y dimensions such that the number of columns and rows in the final matrix is at least equivalent to the number of columns and rows in $\mathbf{S}$. The additional columns (from the right) and rows (from the bottom) are then removed, and the matrix is stored in the Pruning Matrix (PM). The columns containing all zeros are removed from the PM while keeping track of the original indexes of the non-zero columns in a vector ***Idx***.

— *Minimization of the Sum of Saliency of the Weights to be Pruned*. The objective of this step is to generate a mapping strategy which minimizes the sum of saliency of weights to be pruned. It can mathematically be represented as:

$$\mathrm{argmin}_{\mathrm{mapping}} ||S^*. * \mathrm{PM}||. \tag{3.3}$$

Here, $\mathbf{S}^*$ represents a transformed version of $\mathbf{S}$ generated after applying rearrangement of network parameters based on the mapping strategy presented in §3d. The '.*' operation represents element-wise multiplication of the matrices. In this step, we use the proposed algorithm 1 or algorithm 2 to find the mapping sequence of neurons/filters. Algorithm 1 is based on a naive approach for fast generation of an acceptable solution. However, algorithm 2 is based on a branch and bound approach to propose an optimal solution. The detailed descriptions of algorithms 1 and 2 are as below.

Algorithm 1 presents a naive method to find a sub-optimal mapping strategy. The algorithm takes the ***Costs*** matrix, which contains the costs of mapping each neuron/filter to each faulty column of the systolic array, and ***Idx*** vector, which contains the indexes of the faulty columns of the array, as inputs. The ***Costs*** matrix is computed using matrix multiplication of *transpose* of $\boldsymbol{S}$ with ***PM***. Each row of the ***Costs*** matrix contains the cost of mapping a filter to each faulty column

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

Algorithm 1 A fast method to reduce the sum of saliency of the weights of a layer that have to be pruned due to permanent faults.

Inputs: A matrix ***Costs*** containing the costs of mapping neurons/filters to faulty columns of the array and a vector ***Idx*** containing the indexes of faulty columns of the array
Outputs: A matrix ***Mapping*** which defines which neuron/filter has to be mapped to which faulty column and a variable *TCost* which defines the cost of this mapping scheme
Initialize: $TCost = 0$, ***Mapping*** = Zeros(Number of Columns in ***Costs***, 2) and ***NF_Idx*** = $[1, 2, 3, \ldots$ Number of neurons/filters in the layer]

1: **for** i = 1 to Number of columns in ***Costs*** at the start of the loop **do**
2: [*val*, *row_id*, *col_id*] = min(***Costs***)
3: ***Mapping***(i,1) = ***NF_Idx***(*row_id*)
4: ***Mapping***(i,2) = ***Idx***(*col_id*)
5: *TCost* = *TCost* + *val*
6: ***Costs*** ⇐ ***Costs*** after removing *row_id* row and *col_id* column
7: ***Idx*** ⇐ ***Idx*** after removing *col_id* column
8: ***NF_Idx*** ⇐ ***NF_Idx*** after removing *row_id* column
9: **end for**
10: **return** *TCost* and ***Mapping***

of the array. The algorithm, at each iteration, finds the minimum value in the matrix (line 2 in algorithm 1) and then associates the corresponding neuron/filter index form ***NF_Idx*** with the index of the column of the array from ***Idx*** (lines 3 and 4). The costs associated with the selected filter and the column of the array are then removed from the ***Costs*** matrix (line 6). Similarly, the index vectors are also updated (lines 7 and 8). The resultant matrix and vectors are then used in the next iteration. The algorithm iterates for the number of filters to be mapped on the faulty columns (line 1), i.e. the number of columns in the ***Costs*** matrix, and outputs the mapping (***Mapping***) and the corresponding total cost (*TCost*).

Algorithm 2 presents a more robust approach for searching the optimal mapping which provides minimum sum of saliency of the weights to be pruned. The algorithm is based on the branch and bound method where it recursively calls (line 16) the *Branch&Bound* function (line 3) with a smaller problem, i.e. smaller ***Costs*** matrix (lines 11 till 14). The algorithm keeps track of the cost of already selected pairs in *CS_Cost* and bounds the algorithm from searching in the same branch if the overall cost is more than the cost of the reference mapping ***R_Mapping*** (lines 10 and 26). Note that at beginning of the algorithm, the reference cost is computed using algorithm 1 to speed-up the process. If, at some point, a mapping which has cost less than the cost of the reference mapping is found the reference mapping and the reference cost are updated (lines 26 till 28). As the algorithm is meant to search for all the possible branches (combinations) and can take endless amount of time for larger ***Costs*** matrices, we limit the search by applying a limit on the number of miss hits (*MH_Count*), i.e. *Termination_Limit*, (line 18) from the last reference mapping update. We also use a *per branch search limit* (line 7) to decrease the number of searches in each branch and search only on a defined number of nodes which offer minimum costs (lines 6 and 7).

(d) Rearrangement of the network parameters

In most of the DNNs, the arrangement of the neurons/filters within a layer can be changed without affecting the functionality of the network. Figure 7 illustrates this with the help of an example where two neurons in an FC layer are swapped. Figure 8 illustrates a similar example for the filters in a CONV layer. To maintain the functionality, the corresponding connections in the subsequent layer of the network are also swapped. This property helps in maintaining

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

Algorithm 2 A branch and bound method to optimally reduce the sum of saliency of the weights of a layer that have to be pruned due to permanent faults.

Inputs: A matrix ***Costs*** containing the costs of mapping neurons/filters to faulty columns of the array and a vector ***Idx*** containing the indexes of faulty columns of the array
Outputs: A matrix ***Mapping*** which defines which neuron/filter has to be mapped to which faulty column and a variable *TCost* which defines the cost of this mapping scheme
Initialize: *TCost* = 0, ***Mapping*** = Zeros(Number of Columns in ***Costs***, 2), ***NF_Idx*** = [1, 2, 3, ... Number of neurons/filters in the layer], and ***MH_Count*** = 0
Find a reference minimum cost *RMin_Cost* and corresponding mapping scheme ***R_Mapping*** using Algorithm 1
[*TCost*, ***Mapping***, *MH_Count*] = Branch&Bound(***Costs***, ***Idx***, ***NF_Idx***, *RMin_Cost*, ***R_Mapping***, 0, [], *MH_Count*)

Function [*RMin_Cost*, ***R_Mapping***, *MH_Count*] = Branch&Bound(***Costs***, ***Idx***, ***NF_Idx***, *RMin_Cost*, ***R_Mapping***, *CS_Cost*, ***CS_Mapping***, *MH_Count*)
if size(***Idx***)≠1 **then**
    for *i* = 1 to Number of Rows in ***Costs*** - Number of Columns in ***Costs*** + 1 **do**
        Sort i^{th} row of ***Costs*** and store indexes in ***SR_Idx***
        for *k* = 1 to minimum(Number of Columns in ***Costs***, ***Per_Branch_Search_Limit***) **do**
            MH_Count = *MH_Count* + 1
            j = ***SR_Idx***(k)
            if *CS_Cost* + ***Costs(i,j)*** < *RMin_Cost* **then**
                New_Costs ⇐ ***Costs*** after removing its ith row and j^{th} columns
                New_Idx ⇐ ***Idx*** after removing its jth column
                New_NF_Idx ⇐ ***NF_Idx*** after removing its jth column
                New_CS_Mapping ⇐ ***CS_Mapping*** after appending the row [***NF_Idx***(*i*), ***Idx***(*j*)]
                MH_Count = *MH_Count* - 1
                [*RMin_Cost*, ***R_Mapping***, *MH_Count*] = Branch&Bound(***New_Cost***, ***New_Idx***, ***New_NF_Idx***, *RMin_Cost*, ***R_Mapping***, *CS_Cost* + ***Costs(i,j)***, ***New_CS_Mapping***, *MH_Count*)
            end if
            if *MH_Count* >= *Termination_Limit* **then**
                return *RMin_Cost*, ***R_Mapping***, *MH_Count*
            end if
        end for
    end for
else
    MH_Count = *MH_Count* + 1
    [*val*, *row_id*, *col_id*] = min(***Costs***)
    if *CS_Cost* + *val* < *RMin_Cost* **then**
        RMin_Cost = *CS_Cost* + *val*
        R_Mapping ⇐ ***CS_Mapping*** after appending the row [***NF_Idx***(*row_id*), ***Idx***(*col_id*)]
        MH_Count = 0
    end if
end if
return *RMin_Cost*, ***R_Mapping***, *MH_Count*

the original structure and the computing sequence inside the accelerator. The only exceptions are the networks which have local response normalization layers, e.g. the AlexNet [23]. In such networks, the computing sequence in the accelerator has to be slightly modified and additional

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

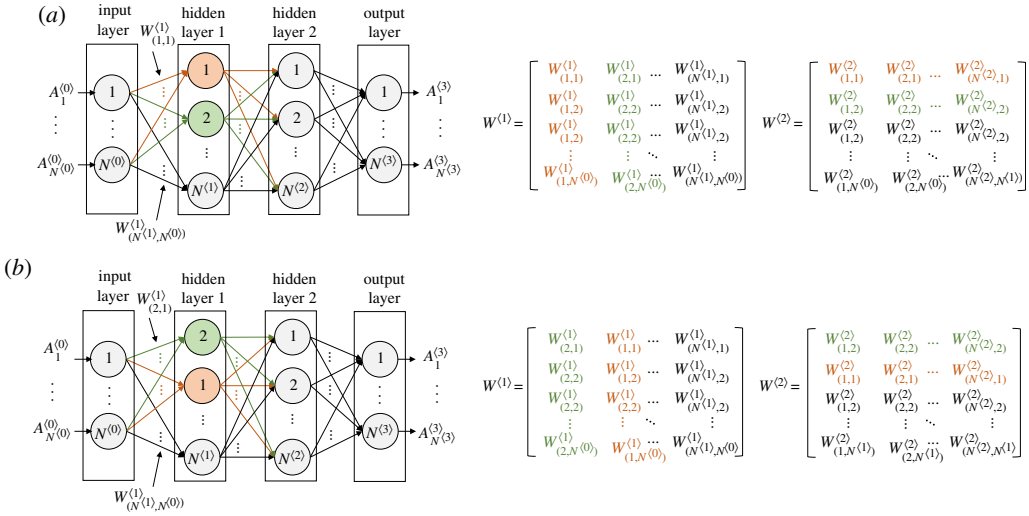


Figure 7. Impact of rearranging neurons in a layer of a fully-connected DNN on the arrangement of the weights to be mapped on the systolic array. (*a*) Shows the arrangement before swapping neurons 1 and 2 in the first hidden layer of a fully-connected DNN and (*b*) shows the arrangement after swapping the neurons. The left side of the figure illustrates the state of the neural network and the right side shows the weights of the first and second hidden layers in a manner in which they will be mapped on a systolic array. Different colours are used to show the association between the neurons and weights. (Online version in colour.)

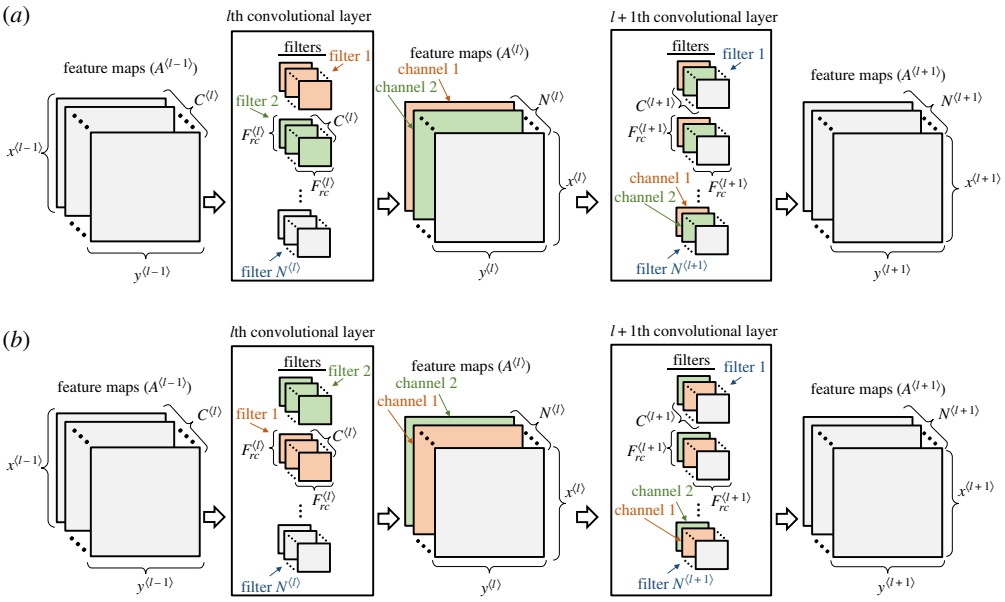


Figure 8. Impact of rearranging filters in a layer of a CNN on the arrangement of the weights in the neural network. (*a*) shows the arrangement of filters and their channels before swapping filters 1 and 2 in the *l*th convolutional layer of a CNN and (*b*) shows the arrangement after swapping the filters. Note that a swap of filters in the *l*th layer of a CNN requires a swap of the respective channels in the $l+1$th layer of the CNN to maintain the functionality. (Online version in colour.)

data arrangement operations might be required which can affect the efficiency of the system. However, almost all the modern DNNs do not make use of this feature, and hence can directly benefit from the proposed approaches of this article.

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

(e) Network approximation

After setting the corresponding weights that have to be mapped on the faulty computational units to zero, we compute the mean of the output activations of the neurons and filters that are pruned using a small subset of the validation dataset. The mean values are compared with the mean values acquired using the original network, and based on their difference, the biases of the neurons/filters are adjusted to compensate for the effects of pruning. Note that, in case the training dataset is available, fine-tuning can also be exploited for regaining the lost accuracy.

(f) SalvageDNN under changing fault maps

In this section, we explain with the help of examples how the proposed methodology helps salvage a faulty DNN hardware. The practical cases can be represented using two scenarios: (1) where the fault maps are generated using post-fabrication testing and each chip has a distinct fault map; and (2) where the fault map of a chip is changing over time and can be extracted using BIST support available in the chip. Note that, for ease of understanding, in both the scenarios, we assume that only the faulty PEs are required to be bypassed to avoid error propagation during execution and that the saliency of the network parameters is computed using L1-norm, i.e. their absolute values.

Scenario 1. Different hardware chips can have different fault maps, i.e. the fault maps can vary across chips. To explain how our proposed approach handles such a scenario, we present an example where four filters (figure 9*a*) each having four values (i.e. weights) have to be mapped on a 4×4 systolic array to perform the dot-product operations. The filters are unrolled and mapped on the systolic array based on the baseline systolic array design discussed in §2b. Figure 9 highlights four different example cases each representing a systolic array belonging to a different chip. The cases are explained as follows:

— Case 1: None of the PEs in the systolic array is faulty (figure 9*b*).
— Case 2: The array has two faulty PEs (figure 9*c*).
— Case 3: The array has the same number of faulty PEs as in Case 2, but at different locations (figure 9*d*).
— Case 4: The array has different number of faulty PEs than Case 2 and Case 3, and also at different locations (figure 9*e*).

In Case 1 (figure 9*b*), the mapping does not affect the accuracy of the results, as there are no faulty PEs in the array which have to be bypassed during execution. In Case 2 (figure 9*c*), the filters are mapped onto the columns such that the sum of absolute weights mapped on the faulty PEs is minimum, which in the example is achieved by mapping filters 1, 2, 3 and 4 on columns 2, 1, 4 and 3, respectively, of the systolic array. In Case 3 (figure 9*d*), the fault map of the array is different from the ones presented in earlier two cases, therefore, the mapping can be different, depending solely on the fault map of the array and the filter values. As can be seen in figure 9*d*, to achieve the minimum sum of absolute weights, filters 1, 2, 3 and 4 are now mapped on columns 2, 3, 4 and 1, respectively. Similarly, in Case 4 (figure 9*e*) the mapping is defined based on its fault map. It is important to clarify here that, similar to the state-of-the-art FAP technique [8], our proposed technique also requires adaptation for each faulty chip based on its fault map. However, unlike the state of the art, our technique can avoid retraining, and, therefore requires much less time for the adaptation compared to the retraining-based approaches, as will be shown in §4b(v). Moreover, our technique also does not require access to the training dataset, which in most of the practical cases is not available (as stated in §1). Details regarding how our proposed approach minimizes the mapping are presented in algorithms 1 and 2.

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

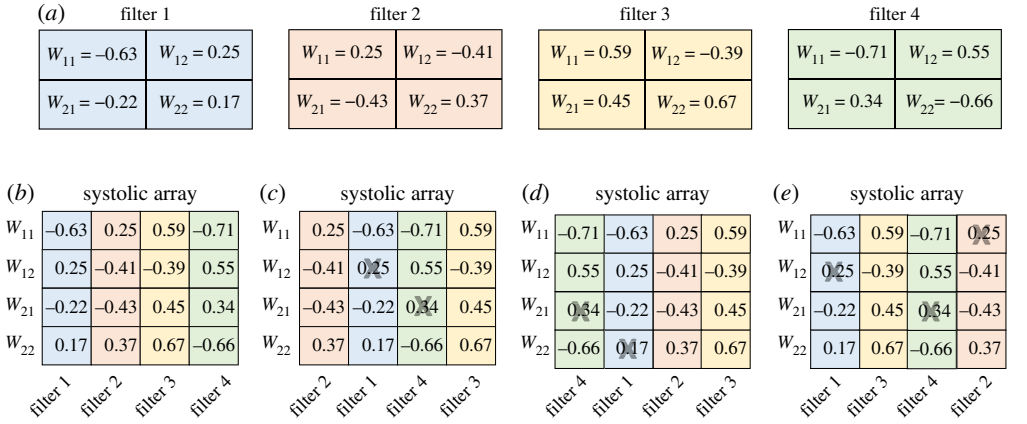


Figure 9. An example illustration of how the mapping would vary across chips having different fault maps. The grids shown in *b*, *c*, *d* and *e* corresponds to 4 × 4 systolic arrays from four different chips. Each small box inside a grid represents a single processing element (PE). The PEs with black cross over them in *c*, *d* and *e* correspond to faulty PEs. The filters considered in this example are shown in *a*. (*a*) Filters, (*b*) Case 1, (*c*) Case 2, (*d*) Case 3, and (*e*) Case 4. (Online version in colour.)

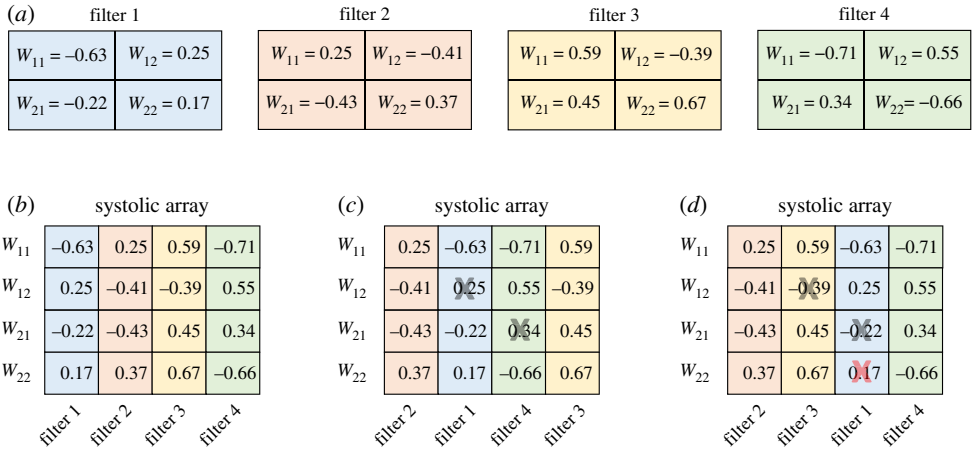


Figure 10. An example illustration of how the mapping would change if the fault map of the DNN hardware changes over time. The grid shown in *b*, *c* and *d* corresponds to a systolic array, where each small box represents a single processing element (PE). The PEs with black cross over them in (*c*) and (*d*) correspond to faulty PEs detected during post-fabrication testing and the PE with red cross over it in (*d*) corresponds to the PE which experienced fault over time due to wear-out. (*a*) Filters, (*b*) Case 1, (*c*) Case 2, and (*d*) Case 3. (Online version in colour.)

Scenario 2. The fault maps can change over time, e.g. due to wear-out. To explain how the mapping changes to take this effect of additional faults into account, we take an example where four filters (figure 10*a*), the same as shown in figure 9*a*, have to be mapped on a 4 × 4 systolic array. Figure 10 highlights three different cases:

— Case 1: None of the PEs in the systolic array is faulty (figure 10*b*).
— Case 2: The array has two faulty PEs (figure 10*c*).
— Case 3: With the passage of time, the number of faulty PEs in the array presented in Case 2 has increased to three (figure 10*d*).

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

Case 1 and Case 2 are the same as presented in figure 9, where, in Case 1, the mapping does not affect the accuracy of the results and, in Case 2, the filters 1, 2, 3 and 4 are mapped on columns 2, 1, 4 and 3 respectively (figure 10*c*), to minimize the sum of absolute weights that are mapped on the faulty PEs. However, in Case 3, once the fault map is updated with the help of the available BIST support, the mapping of filters should also be updated to achieve a lower sum. As can be seen in figure 10*d*, the minimum sum is achieved by mapping filters 1, 2, 3 and 4 on columns 3, 1, 2 and 4, respectively. Note that, with the mapping of Case 2 and the fault map of Case 3, the sum of absolute weights mapped on the faulty PEs would have been 1.25, while with the updated mapping, the sum is 0.78, as can be observed in figure 10*d*.

Associated testing and reconfiguring overheads: Post-fabrication testing is an essential part of the manufacturing process, which is required to verify the functional correctness of the manufactured hardware. This process is also used for diagnostics to check which module/components inside the hardware are faulty [24–26]. Therefore, no additional cost would be required to extract the fault map in Scenario 1, apart from the usual cost necessary for effective post-fabrication testing. The fault map extracted during post-fabrication testing can be used, in our case, for reconfiguring the fault tolerant systolic arrays, as well as for determining the DNN mapping using the proposed, SalvageDNN, methodology. However, storage of the fault map requires some memory, which is relatively small as it depends on the size of the systolic array, i.e. one bit per PE. On the other hand, in Scenario 2, we need boot-time/online BIST support to update the fault map after an interval of time. However, the cost of such support is high, which can be reduced by using algorithm-based fault detection and location techniques [27]. The algorithm-based techniques have proven to be very effective in the case of regular hardware composed of homogeneous components such as systolic arrays.

4. Results and discussion

(a) Hardware synthesis results

Table 1 shows the hardware characteristics of the processing elements of the baseline systolic array design (i.e. the one without fault tolerance circuitry) and the state-of-the-art fault tolerant systolic array design. The hardware results are generated following an ASIC design flow using Cadence Genus tool and a TSMC 65nm technology library. As can be seen in the table, for all the listed systolic array sizes, the MAC units consume approximately 66% of the area of a PE, and the multiplexer inside each PE (as used in the state-of-the-art fault tolerant systolic array design) consumes approximately 6% of the area of a PE. This shows that having a fault tolerant design that mitigates permanent faults in the MAC and MUX units of the array can significantly improve the yield of the manufacturing process.

Figure 11 presents a comparison of the hardware characteristics of the PEs of different sizes and types of systolic arrays as discussed in §3a. The figure shows that the fault-tolerant PEs consume slightly more area as compared to a baseline PE depending on the complexity of the bypassing/cut-off circuitry used in the design. For example, a PE with dual bypass and cut-off circuitry (i.e. DBNC_PE) consumes approximately 25% more area, and a PE with cut-off circuitry (i.e. *C_PE*) consumes approximately 1% additional area, when compared to the baseline PE used for the same size systolic array. The variation in the area with respect to the array size is due to the fact that the bit-width of the partial sums increases with an increase in the number of PEs in a single column of the array. Similar to the area characteristics, the critical path delay of a PE also depends on the design and the size of the systolic array. However, the variations in the delay are relatively small, i.e. at the maximum 6.5% for the 8x8 systolic array composed of *DBNC_PE* when compared to the baseline PE for the same array size. The trend in the power characteristics is approximately similar to the trend observed in the area characteristics of the PEs. Hence the multiple options, that are shown in figure 6, §3a, provide different design trade-offs in terms of area, latency, power and resilience.

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

Table 1. Hardware results of the components used in different sizes of the baseline and the state-of-the-art fault tolerant systolic arrays. The bit-widths of the partial sums in 8 × 8, 16 × 16, 32 × 32 and 256 × 256 arrays were set to 19, 20, 21 and 24, respectively, assuming the bit-width of weights and activations to be 8-bit.

		area (cell area)	delay (ns)	power (μW)
8 × 8 array	baseline PE	931	2.17	76.77
	SOA PE	972	2.21	80.10
	MAC	620	1.97	62.11
	2-to-1 MUX	56	0.24	2.69
16 × 16 array	baseline PE	950	2.23	77.09
	SOA PE	993	2.26	82.09
	MAC	632	2.01	62.01
	2-to-1 MUX	59	0.25	2.88
32 × 32 array	baseline PE	965	2.32	77.62
	SOA PE	1010	2.35	81.10
	MAC	640	2.10	61.10
	2-to-1 MUX	62	0.25	2.90
256 × 256 array	baseline PE	1006	2.65	77.40
	SOA PE	1058	2.68	80.64
	MAC	661	2.44	61.66
	2-to-1 MUX	70	0.28	3.39

(b) Impact of fault-aware mapping (SalvageDNN) on the DNN accuracy

(i) Experimental set-up

For analysing the impact of permanent faults on the accuracy of DNNs, we considered moderately deep neural networks, i.e. the VGG11 and the VGG16, trained on the Cifar-10 [28] and the ImageNet [29] datasets. The DNNs are taken from the *Neural Network distiller* [20] which is an open-source tool for neural network compression. The details of the models are provided in table 2 and a few key characteristics of the datasets are presented in table 3. We used the same tool (i.e. distiller) to quantize the neural network weights and activations to 8-bits. For fault injection, similar to the well-established works in the reliability community, we considered a random distribution of the faults across the array and performed multiple iterations of each experiment using different seeds. To achieve a fair comparison, we used the same seeds for the experiments across arrays and fault rates. Moreover, we also took into consideration the area of MAC and MUX units for distributing the faults.

(ii) Comparison with the state-of-the-art FAP approach

Figure 12 illustrates the impact of permanent faults on the accuracy of the VGG-11 network when mapped on a 256 × 256 systolic array composed of SOA_PEs using the FAP and the proposed technique. To have a fair comparison with the state-of-the-art FAP approach, we adopted a similar evaluation methodology as reported in their paper [8]. That is, in this experiment, we assumed

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

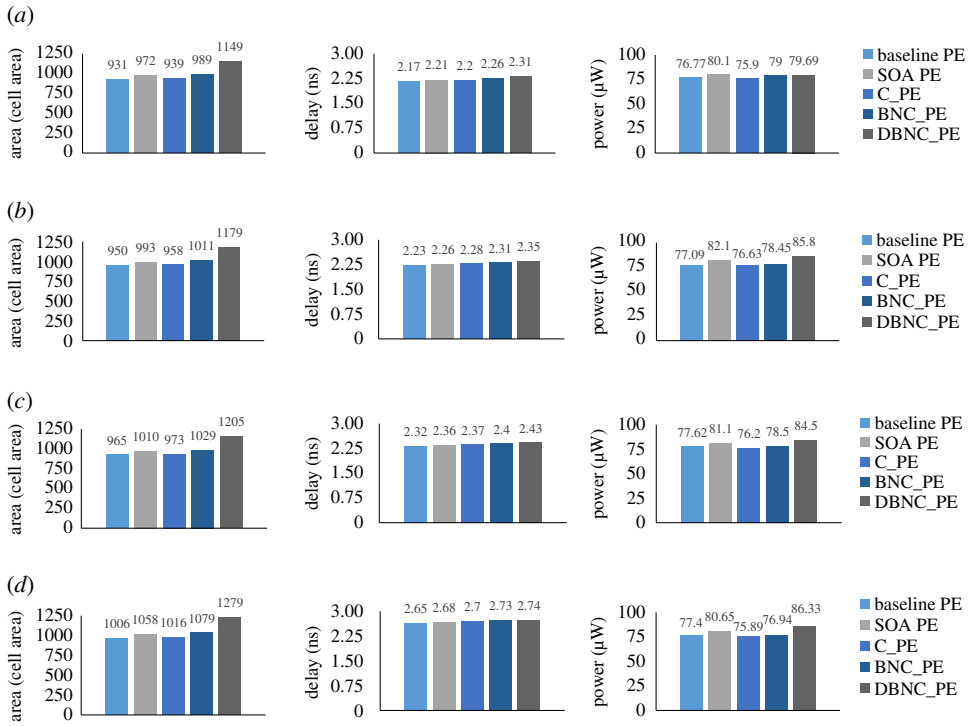


Figure 11. Hardware synthesis results of the PEs of different sizes and types of systolic array designs. (*a*) Hardware characteristics of PEs used in different types of 8 × 8 systolic arrays, (*b*) hardware characteristics of PEs used in different types of 16 × 16 systolic arrays, (*c*) hardware characteristics of PEs used in different types of 32 × 32 systolic arrays and (*d*) hardware characteristics of PEs used in different types of 256 × 256 systolic arrays. (Online version in colour.)

Table 2. DNNs and the datasets used for evaluation.

network and dataset	network architecture	baseline accuracy (%)
VGG11 for Cifar-10	layers: CONV(64, 3, 3, 3); CONV(128, 64, 3, 3); CONV(256, 128, 3, 3); CONV(256, 256, 3, 3); CONV(512, 256, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); FC(10, 512)	85.38%
VGG11 for ImageNet	layers: CONV(64, 3, 3, 3); CONV(128, 64, 3, 3); CONV(256, 128, 3, 3); CONV(256, 256, 3, 3); CONV(512, 256, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); FC(4096, 25088); FC(4096, 4096); FC(1000, 4096)	68.04% (Top1) 88.07% (Top5)
VGG16 for ImageNet	layers: CONV(64, 3, 3, 3); CONV(64, 64, 3, 3); CONV(128, 64, 3, 3); CONV(128, 128, 3, 3); CONV(256, 128, 3, 3); CONV(256, 256, 3, 3); CONV(256, 256, 3, 3); CONV(512, 256, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); FC(4096, 25088); FC(4096, 4096); FC(1000, 4096)	70.85% (Top1) 90.0% (Top5)

that faults can only occur in the MAC units of the array and not in the MUX or other components of the PEs. From this experiment, the following key observations can be made:

— the figure illustrates that, with an increase in the fault rate, the accuracy of the network when mapped using FAP approach decreases rapidly as compared to when mapped using SalvageDNN, i.e. while considering the saliency of the weights;

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

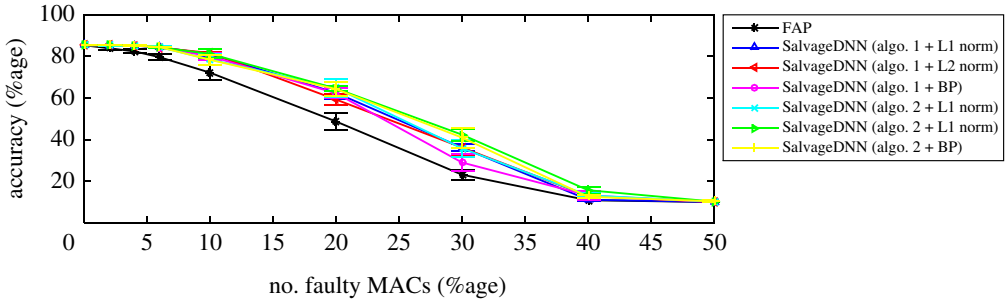


Figure 12. Impact of number of faulty MAC units in a 256 × 256 systolic array composed of SOA_PEs on the accuracy of the VGG-11 network trained on the Cifar-10 dataset. (Online version in colour.)

Table 3. A summary of the key characteristics of the datasets used in the evaluation of SalvageDNN and the comparison with the state of the art.

dataset	characteristics	
Cifar-10	number of training images	50000
	number of testing images	10000
	size of images	32 × 32 × 3
	number of classes	10
ImageNet	number of training images	∼14 million
	number of testing images	100000
	size of images	224 × 224 × 3
	number of classes	1000

— at lower fault rates (i.e. until 6% of the MAC units are faulty), which are typically common in the real-world settings if the fabrication process is mature, SalvageDNN helps in maintaining the baseline accuracy of the network while the DNN accuracy of the network when mapped using FAP starts decreasing instantly with the increase in the fault rate;

— the error bars in the figure, which shows the standard deviation of the accuracy across multiple iterations of the same experiment, highlights that the accuracy is highly dependent on the location of the faulty PEs. Therefore, FAP, which uses a fixed mapping approach, results in higher standard deviation at the same fault rate. However, the proposed SalvageDNN adapts as per the locations of the faults and offers less deviation when compared with FAP method;

— the type of saliency and the algorithm used in SalvageDNN does not significantly impact the technique. Note that algorithm 2 provides sightly better results at higher fault rates. However, based on the computational requirements algorithm 1 + L1 norm should be preferred.

(iii) Comparison with the state-of-the-art FAP+T approach

Figure 13 illustrates the impact of using SalvageDNN and then applying fault-aware retraining on the accuracy of the VGG11 network for the Cifar10 classification. The subfigures show the comparison between the 'FAP+T' [8] and the 'SalvageDNN + retraining', when the underlying hardware has faulty PEs. As can be seen from the figure, both the approaches perform equally

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

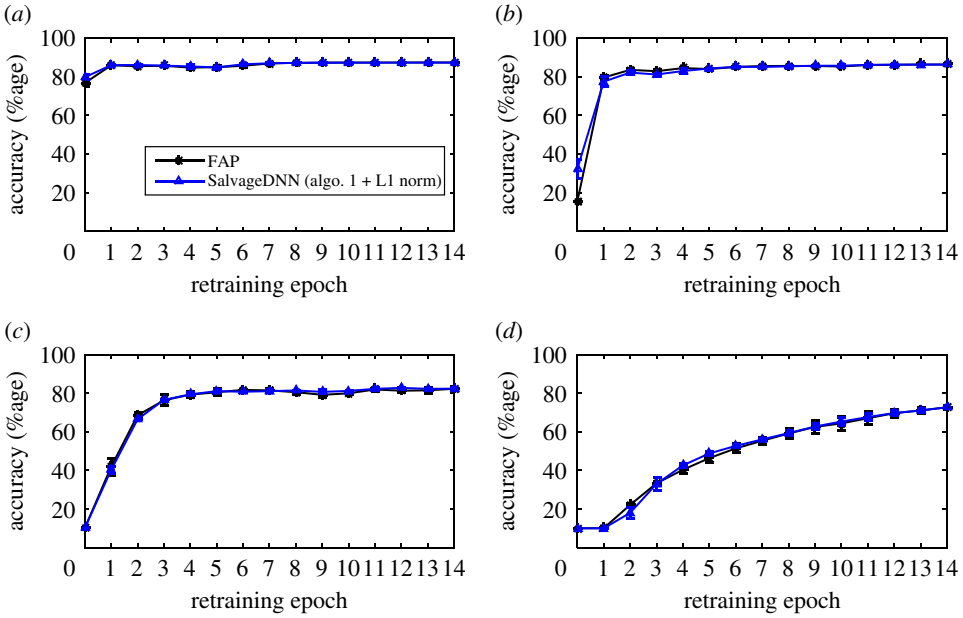


Figure 13. Impact of using SalvageDNN before applying fault-aware training on the accuracy of the VGG11 network trained for the Cifar10 classification. The subfigures show the comparison between FAP + retraining and SalvageDNN + retraining when the underlying hardware has: (*a*) 10% faulty PEs, (*b*) 30% faulty PEs, (*c*) 50% faulty PEs, and (*d*) 70% faulty PEs. (Online version in colour.)

well and if the training dataset and a significant amount of computational resources for retraining are available, a large percentage of the lost accuracy can be regained through retraining. The number of epochs required to regain the accuracy seems to be dependent on the number of faulty PEs in the array. This can be observed from figure 13, where figures 13*a*,*b*,*c* and 13*d* show the test accuracy after every epoch of retraining when 10%, 30%, 50% and 70% of the total PEs are faulty respectively. For example, in figure 13*a*, the baseline accuracy was regained after only one epoch of retraining, however, in figure 13*c*, it took five epochs to reach the saturation point.

(iv) Evaluation using DNNs trained for larger datasets

Figure 14 illustrates the comparison of the proposed approach with the state-of-the-art FAP approach when used for the VGG11 network trained on a larger dataset, i.e. the ImageNet. The systolic array considered in this scenario is a 256 × 256 sized array composed of SOA_PEs. As can be seen from the figure, for lower fault rates, i.e. around 0.06 (6%), SalvageDNN can help a network maintain accuracy close to its baseline, while the accuracy with FAP approach drops significantly even when only 2% of the total PEs are faulty. Similar results are observed in figure 15 for the VGG16 network trained on the ImageNet dataset.

(v) Comparison of the execution time of the proposed technique with the retraining-based approaches

Table 4 presents a comparison between the execution time required for the SalvageDNN and the FAP+T for different neural networks and datasets. The details of the networks and a summary of a few of the key characteristics of the datasets used in the comparison are presented in tables 2 and 3, respectively. As can be seen from the tables, the execution time depends on the complexity of the dataset (only for retraining-based approaches) as well as the size of the network. However, in all the cases SalvageDNN requires, at the very least, an order of magnitude less execution time

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

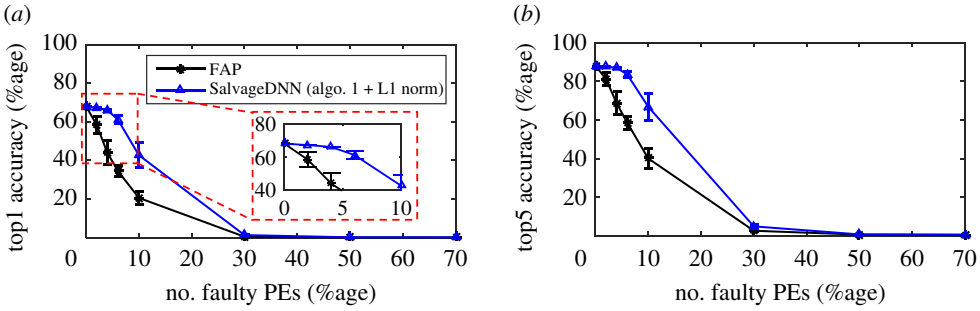


Figure 14. Comparison of SalvageDNN with the state-of-the-art FAP approach when used for the VGG11 network trained on the ImageNet dataset which has to be mapped on a 256 × 256 sized systolic array. The two commonly used accuracy metrics, i.e. the Top1 and the Top5 accuracies, are shown for cases having different number of faulty PEs, in subfigures (*a*) and (*b*) respectively. (Online version in colour.)

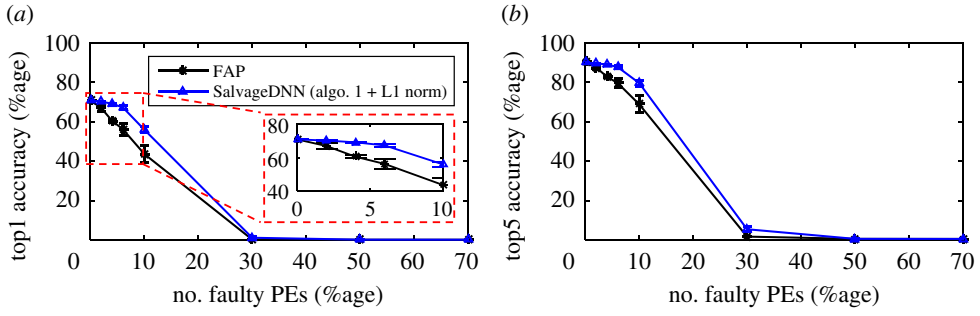


Figure 15. Comparison of SalvageDNN with the state-of-the-art FAP approach when used for the VGG16 network trained on the ImageNet dataset which has to be mapped on a 256 × 256 sized systolic array. The two commonly used accuracy metrics, i.e. the Top1 and the Top5 accuracies, are shown for cases having different number of faulty PEs, in subfigures (*a*) and (*b*) respectively. (Online version in colour.)

Table 4. Execution time comparison of SalvageDNN with retraining-based approach, i.e. FAP+T [8], for different networks trained on different datasets.

network	dataset	execution time (sec)		
		retraining (single epoch)	SalvageDNN (algo. 1 + L1-norm)	SalvageDNN speedup
VGG11	Cifar-10	24.63	1.12	21.99×
VGG11	ImageNet	6005.93	112.67	53.31×
VGG16	ImageNet	9871.38	113.26	87.16×

than the time required for a single epoch of retraining. Note that the amount of savings grow with the increase in the size and complexity of the dataset and the network. For example, the time required for SalvageDNN for the VGG16 trained on the ImageNet dataset is around 87× lesser than the time required for a single epoch of retraining.

(vi) Impact of permanent faults in the proposed systolic array designs

Figure 16 shows the impact of permanent faults on the accuracy of the VGG-11 network when executed on systolic arrays of different sizes and types (see the designs in figure 6, §3a). For these experiments, we consider that the faults can also occur in the multiplexers of a fault-tolerant PE. To have a fair comparison across arrays, we take into consideration the area of the MAC and

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

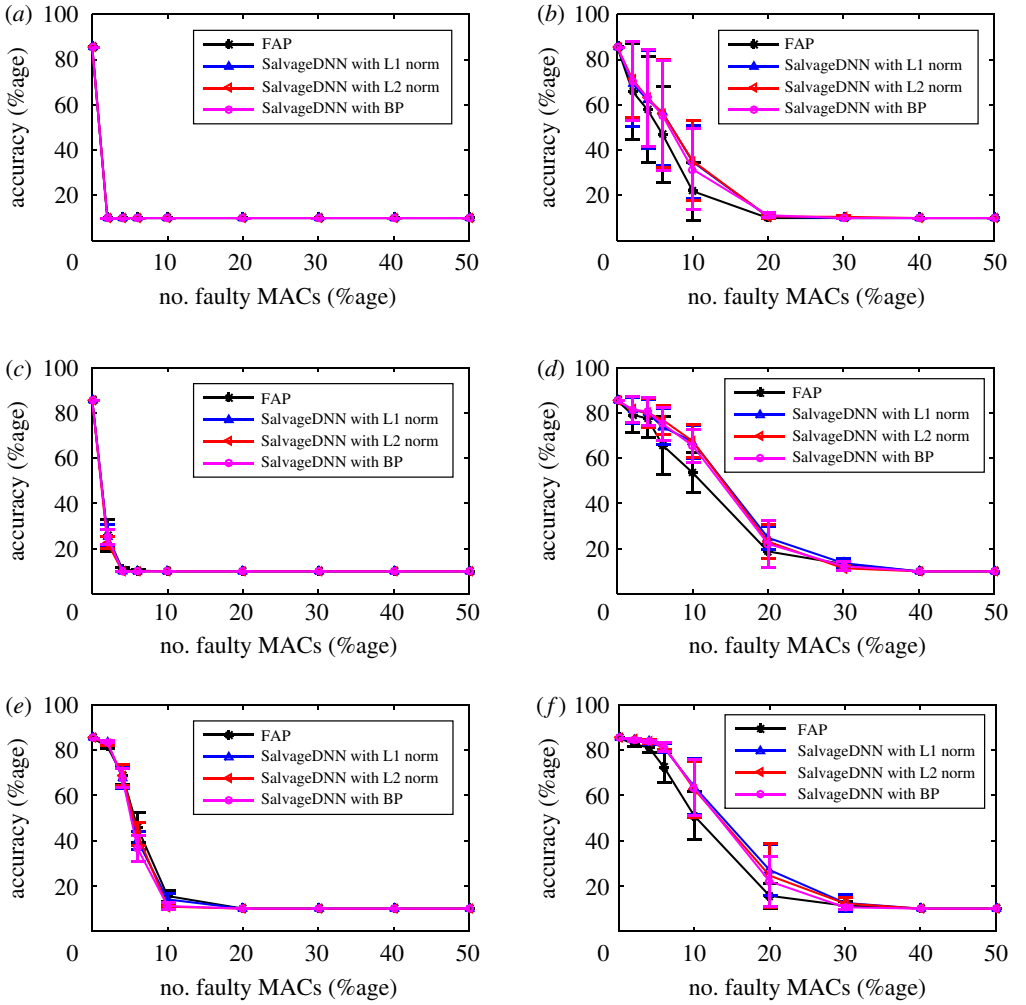


Figure 16. Impact of permanent faults in the proposed systolic array designs on the accuracy of the VGG-11 network trained on the Cifar-10 dataset. (*a*) 256 × 256 array with C_PEs, (*b*) 8 × 8 array with C_PEs, (*c*) 256 × 256 array with BNC_PEs, (*d*) 8 × 8 array with BNC_PEs, (*e*) 256 × 256 array with DBNC_PEs and (*f*) 8 × 8 array with DBNC_PEs. (Online version in colour.)

multiplexer units for injecting faults. By analysing the figure, the following observations can be made:

— SalvageDNN outperforms the state-of-the-art FAP approach in all the cases;
— the accuracy of the systolic arrays containing only C_PEs drops significantly, even at lower fault rates, as shown in figures 16*a*,*b*. This is due to the fact that a PE can get disconnected due to a fault in its downstream neighbours. Therefore, the average probability of a PE getting disconnected at a specific fault rate is significantly high in the arrays composed of *C_PEs*;
— with an increase in the number of bypass connections in a PE for fault tolerance, the average DNN accuracy at a specific fault rate increases. For example, this can be observed by analysing the DNN accuracy in figures 16*a*,*c*,*e* at where 2% of the MAC units are faulty. The *DBNC_PE* almost maintains the baseline accuracy whereas the accuracy offered by C_PE equals 10% (i.e. equivalent to a random selection); and
— the DNN accuracy for the same fault rate increases with the decrease in the size of the systolic array. This can be observed by comparing figures 16*a*, 16*c* and 16*e* with

figures 16*b*, 16*d* and 16*f*, respectively. This is also associated with the probability of disconnection of a MAC unit.

In summary, in order to mitigate the effects of permanent faults in the multiplexer units, designs with more bypass connections are better. However, they result in area, power/energy and delay overheads, which increase significantly with the number of bypass connections.

5. Conclusion

In this paper, we proposed a novel methodology, *SalvageDNN*, for mitigating permanent faults in DNN accelerators. The technique is based on saliency-driven fault-aware mapping of a DNN to a systolic array-based hardware accelerator without the need of re-training and without affecting the hardware level dataflow. At the core, the approach uses rearrangement of the network parameters at the software-level based on the given fault map of the hardware. This is achieved by finding the arrangement that minimizes the saliency of the parameters that will be pruned due to disconnection of the faulty MAC units / processing elements. We also proposed several hardware optimizations in the systolic array design and analysed their design trade-offs.

Data accessibility. This article does not contain any additional data.

Authors' contributions. Both authors have equal contributions to the concept development and paper writing. M.A.H. did the complete implementation of the idea.

Competing interests. We declare we have no competing interest.

Funding. This study was supported by Deutsche Forschungsgemeinschaft (GetSURE, SPP1500).

Acknowledgements. This Work is supported in parts by the German Research Foundation (DFG) as part of the GetSURE project in the scope of SPP-1500 (http://spp1500.itec.kit.edu) priority program, 'Dependable Embedded Systems'.

References

1. LeCun Y, Bengio Y, Hinton G. 2015 Deep learning. *Nature* **521**, 436. (doi:10.1038/nature14539)
2. Sze V, Chen YH, Yang TJ, Emer JS. 2017 Efficient processing of deep neural networks: a tutorial and survey. *Proc. IEEE* **105**, 2295–2329. (doi:10.1109/JPROC.2017.2761740)
3. Jouppi NP *et al.* 2017 In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual Int. Symp. Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017*, pp. 1–12. New York, NY: ACM.
4. Hanif MA, Putra RVW, Tanvir M, Hafiz R, Rehman S, Shafique M. 2018 Mpna: a massively-parallel neural array accelerator with dataflow optimization for convolutional neural networks. (http://arxiv.org/abs/1810.12910)
5. Chen YH, Yang TJ, Emer J, Sze V. 2019 Eyeriss v2: a flexible accelerator for emerging deep neural networks on mobile devices. *IEEE J. Emerg. Sel. Top. Circuits Syst. IEEE Micro* **9**, 292–308.
6. Constantinescu C. 2003 Trends and challenges in vlsi circuit reliability. *IEEE Micro* **23**, 14–19. (doi:10.1109/MM.2003.1225959)
7. Henkel J, Bauer L, Dutt N, Gupta P, Nassif S, Shafique M, Tahoori M, Wehn N. 2013 Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proc. 50th Annual Design Automation Conf., Austin, TX, 29 May–07 June 2013*, p. 99. New York, NY: ACM.
8. Zhang JJ, Gu T, Basu K, Garg S. 2018 Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator. In *2018 IEEE 36th VLSI Test Symp. (VTS), San Francisco, CA, 22–25 April 2018*, pp. 1–6. New York, NY: IEEE.
9. Koren I, Koren Z. 1998 Defect tolerance in vlsi circuits: techniques and yield analysis. *Proc. IEEE* **86**, 1819–1838. (doi:10.1109/5.705525)
10. Shivakumar P, Keckler SW, Moore CR, Burger D. 2003 Exploiting microarchitectural redundancy for defect tolerance. In *Proc. 21st Int. Conf. Computer Design, San Jose, CA, 13–15 October 2003*, pp. 481–488. New York, NY: IEEE.
11. Product binning. Online. https://en.wikipedia.org/wiki/Product_binning (accessed 20 October 2019).

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

Downloaded from https://royalsocietypublishing.org/ on 25 January 2022

12. Hruska J. Some amd rx 460s can be modded to unlock missing cores, additional performance. https://www.extremetech.com/gaming/240926-amd-rx-460s-can-modded-unlock-missing-cores-additional-performance (accessed 20 October 2019).
13. With cpu chips having billions of transistors, what happens if a few go bad? https://www.quora.com/With-CPU-chips-having-billions-of-transistors-what-happens-if-a-few-go-bad (accessed 20 October 2019).
14. Zhang L, Han Y, Xu Q, Li X. 2008 Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology. In *Proc. Conf. Design, Automation and Test in Europe, Munich, Germany, 10–14 March 2008*, pp. 891–896. New York, NY: ACM.
15. Markovsky Y, Wawrzynek J. 2007 On the opportunity to improve system yield with multi-core architectures. In *Proc. IEEE Workshop on Design Manufacturability & Yield, Santa Clara, CA, 25–26 October 2007*. New York, NY: IEEE.
16. Kim JH, Reddy SM. 1989 On the design of fault-tolerant two-dimensional systolic arrays for yield enhancement. *IEEE Trans. Comput.* **38**, 515–525. (doi:10.1109/12.21144)
17. Kung H, Lam MS. 1983 Fault-tolerance and two-level pipelining in vlsi systolic arrays. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.
18. Agrawal VD, Kime CR, Saluja KK. 1993 A tutorial on built-in self-test. 2. Applications. *IEEE Des. Test Comput.* **10**, 69–77. (doi:10.1109/54.211530)
19. Forlenza DO, Forlenza OP, Robbins BJ. 2016 *Logic-built-in-self-test diagnostic method for root cause identification*. US Patent 9,244,757.
20. Zmora N, Jacob G, Zlotnik L, Elharar B, Novik G. 2018 Neural network distiller. (doi:10.5281/zenodo.1297430)
21. Li H, Kadav A, Durdanovic I, Samet H, Graf HP. 2016 Pruning filters for efficient convNets (http://arxiv.org/abs/1608.08710).
22. Yu R, Li A, Chen CF, Lai JH, Morariu VI, Han X, Gao M, Lin CY, Davis LS. 2018 Nisp: Pruning networks using neuron importance score propagation. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition, Salt Lake City, UT, 18–23 June 2018*, pp. 9194–9203. New York, NY: IEEE.
23. Krizhevsky A, Sutskever I, Hinton GE. 2012 Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems, Lake Tahoe, NV, 3–6 December 2012*, pp. 1097–1105. Red Hook, NY: Curran Associates Inc.
24. Powell MD, Biswas A, Gupta S, Mukherjee SS. 2009 Architectural core salvaging in a multi-core processor for hard-error tolerance. In *ACM SIGARCH Computer Architecture News, Austin, TX, 20–24 June 2009*, vol. 37, pp. 93–104. New York, NY: ACM.
25. Nakahara S, Higeta K, Kohno M, Kawamura T, Kakitani K. 1999 Built-in self-test for GHz embedded SRAMs using flexible pattern generator and new repair algorithm. In *Int. Test Conf. 1999. Proc. (IEEE Cat. No. 99CH37034), Atlantic City, NJ, 30–30 September 1999*, pp. 301–310. New York, NY: IEEE.
26. Schuchman E, Vijaykumar T. 2005 Rescue: A microarchitecture for testability and defect tolerance. In *32nd Int. Symp. Computer Architecture (ISCA'05), Madison, WI, 4–8 June 2005*, pp. 160–171. New York, NY: IEEE.
27. Abraham JA, Banerjee P, Fuchs W, Reddy AN *et al.* 1987 Fault tolerance techniques for systolic arrays. *Computer* **20**, 65–75. (doi:10.1109/MC.1987.1663621)
28. Krizhevsky A *et al.* 2009 Learning multiple layers of features from tiny images. Technical Report, Citeseer.
29. Deng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L. 2009 Imagenet: a large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, 20–25 June 2009*, pp. 248–255. New York, NY: IEEE.

26th IEEE International Symposium on On-Line Testing and Robust System Design
(IOLTS)

Dependable Deep Learning: Towards Cost-Efficient Resilience of Deep Neural Network Accelerators against Soft Errors and Permanent Faults

Muhammad Abdullah Hanif, Muhammad Shafique
Technische Universität Wien (TU Wien), Vienna, Austria
{muhammad.hanif, muhammad.shafique}@tuwien.ac.at

***Abstract*—Deep Learning has enabled machines to learn computational models (i.e., Deep Neural Networks – DNNs) that can perform certain complex tasks with claims to be close to human-level precision. This state-of-the-art performance offered by DNNs in many Artificial Intelligence (AI) applications has paved their way to being used in several safety-critical applications where even a single failure can lead to catastrophic results. Therefore, improving the robustness of these models to hardware-induced faults (such as soft errors, aging, and manufacturing defects) is of significant importance to avoid any disastrous event. Traditional redundancy-based fault mitigation techniques cannot be employed in a wide of applications due to their high overheads, which, when coupled with the compute-intensive nature of DNNs, lead to undesirable resource consumption. In this article, we present an overview of different low-cost fault-mitigation techniques that exploit the intrinsic characteristics of DNNs to limit their overheads. We discuss how each technique can contribute to the overall resilience of a DNN-based system, and how they can be integrated together to offer resilience against multiple diverse hardware-induced reliability threats. Towards the end, we highlight several key future directions that are envisioned to help in achieving highly dependable DL-based systems.**

***Index Terms*—Reliability, Dependability, Cost, Efficiency, Deep Learning, DL, Deep Neural Networks, DNNs, Faults, Soft Errors, Aging, Permanent Faults, Manufacturing Defects, Yield, Resilience, Robustness, Systems, Architecture, Accelerator.**

I. Introduction

Deep Learning (DL) enables computational models that are composed of multiple layers to learn hierarchical features directly from raw data [1]. These models (Deep Neural Networks - DNNs) offer state-of-the-art accuracy in many Artificial Intelligence (AI) applications, such as image classification, image segmentation, object detection and speech recognition [2] [3] [4], which in some scenarios have been claimed to surpass human-level precision [5]. This has laid the foundation for these models being used in several safety-critical applications like autonomous driving [6] [7] [8] and medical diagnosis [9] [10] [11]. In these applications, the system is required to analyze the input generated by sensors in real-time. Driven by the compute-intensive nature of DNNs, a significant amount of research has been carried out in developing DNN hardware accelerators that can offer performance- and energy-efficient inference for DNN-based applications [12]. A few examples are TPU [13], MAERI [14] and Eyeriss [15].

Alongside resource-efficient execution, these systems are expected to generate highly reliable output. However, besides the input- and environment-induced errors, the hardware-induced faults (such as soft errors, device aging and manufacturing defects) lead to an increased number of false positives and false negatives, and even system failures (see Fig. 1). This leads to a significant degradation in the performance of the system and also raises various safety concerns in critical applications. For example, in the context of autonomous vehicles, detecting obstacles in the environment when nothing is actually there (false positives) can lead to abrupt braking, and reduced speed while missing the detection of an obstacle (false negative) can lead to a fatal accident. Moreover, these systems have to meet standards like ISO 26262 [16] that allow for only a limited number of errors in the entire system's lifetime. Therefore, it is vital to ensure the robustness of DNNs against a wide range of dependability threats.

Traditional fault-mitigation techniques that are based on redundancy (e.g., Dual Modular Redundancy (DMR) [17] and Triple Modular Redundancy (TMR) [18]) are not effective to be employed for DNN-based applications due to their huge overheads arising from redundant hardware/execution of compute-intensive DNNs and synchronization issues. Other techniques such as Instruction Duplication (ID) [19] [20] and use of Error-Correcting Codes (ECC) also pose similar issues

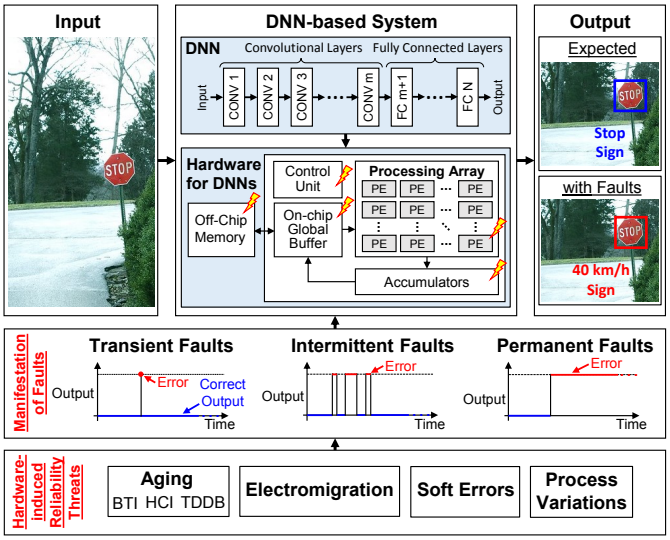


Fig. 1. Reliability threats, their manifestation, and impact on a DNN-based system's output. (used "Stop" sign picture is from the COCO dataset [21])

that lead to noticeable degradation in the system's performance and power/energy efficiency. Therefore, *alternate techniques are being developed that exploit the intrinsic characteristics of DNNs to boost their error-resilience at low cost*. These techniques are being developed with the objective that they can be integrated in the current state-of-the-art systems without drastically reducing their benefits.

Contributions of This Paper:

1) We provide an overview of different methodologies for enhancing the resilience of DNN-based systems against different reliability vulnerabilities, and also highlight their key characteristics.
2) We discuss how each technique can contribute to the overall resilience of a DNN-based system, and how can they be integrated in a cross-layer framework to offer protection against a wide spectrum of hardware-induced reliability threats while incurring low design-time and run-time overheads.
3) We highlight the key challenges and open research problems on the road ahead towards developing dependable AI systems that offer an optimal level of protection against reliability threats.

II. Hardware-induced Reliability Threats

Continuous scaling of CMOS technology, mainly for improved performance and efficiency, has raised various reliability concerns for nano-scale devices. Fig. 1 shows the main types of threats and how they manifest to affect the functionality of systems.

- **Soft Errors** are transient faults that manifest as bit flips and result in data corruption. These are mainly caused by extrinsic sources like alpha particles emitted from the impurities in packaging materials or by neutrons from cosmic radiations when they strike the chip [33]. Temperature is another factor that can result in an increase in the Soft Error Rate (SER).
- **Aging** of electronic circuits occurs due to various physical phenomena such as Bias Temperature Instability (BTI), Hot Carrier

978-1-7281-8187-5/20/$31.00 ©2020 IEEE

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on January 25,2022 at 04:01:02 UTC from IEEE Xplore. Restrictions apply.

TABLE I
AN OVERVIEW OF DIFFERENT HARDWARE (HW) AND SOFTWARE (SW) FAULT-MITIGATION TECHNIQUES FOR MACHINE LEARNING SYSTEMS.

Technique	Abstraction Layer	Related Work	Brief Description	Cost	Targeted Faults	Other Dependencies
Fault-Aware Training	SW	[22] [23]	It incorporates the information of hardware-level faults in the training process to enable the network to adapt accordingly and produce correct output using faulty hardware during inference.	Design-time: High	Permanent faults in memory	Fault map extraction
Fault-Aware Pruning	HW Architecture + SW	[24] [23] [25]	This technique leverages the inherent redundancy in DNNs and propose to drop the faulty computations to maintain the accuracy of the provided DNN. In case of higher number of faults, this technique is coupled with fault-aware training to offer optimal performance.	Run-time: Low	Permanent faults in computational array	Modifications in HW architecture + fault map extraction
Fault-Aware Mapping	HW Architecture + SW	[24]	Fault-aware mapping leverages the fact that different computations/parameters have different level of significance and pruning the least significant set of computations/parameters during fault-aware pruning can have the least impact on the accuracy of the DNN-based system. Therefore, it proposes to find the least significant set and, with the help of mapping, map them to the faulty components	Design-time: Low	Permanent faults in computational array	Fault map extraction
Range Restriction	SW	[26] [27]	In DNNs, faults in sensitive computations usually result in abnormal output at an intermediate stage in the network that can propagate to the output to produce incorrect result. Range restriction techniques propose to determine ranges of intermediate outputs and identify any output that does not fall in the defined range as faulty and map them to some in the range using some pre-defined mechanism.	Design-time: Low Run-time: Low	Transient Faults (i.e., Soft Errors)	Modifications in HW architecture in case of specialized HW
Algorithm-based Fault Tolerance	SW	[28]	Algorithm-based Fault Tolerance (ABFT) techniques that are based on checksum computation are commonly used to protect matrix-matrix multiplications. These techniques have been extended to offer cost-effective error detection and correction for convolution operations as well.	Run-time: Low	Transient faults in computational units and data corruption	No
Radiation Hardening	Circuit	[29] [30]	This approach enables us to make the hardware less prone to radiation-induced faults, using stronger cells or through designing the hardware components such that they are biased towards a specific type of faults towards which DNNs are more resilient. For example, designing SRAM cells such that, in case of radiation-induced faults, they result in a '0' more often then a '1'.	Run-time: Low	Transient Faults (i.e., Soft Errors)	No
Redundancy	HW Architecture + SW	[31] [32]	It introduces selective spatial and temporal redundancy to detect and rectify errors in critical computations during inference.	Run-time: Moderate to High	Transient, intermittent and permanent faults	No

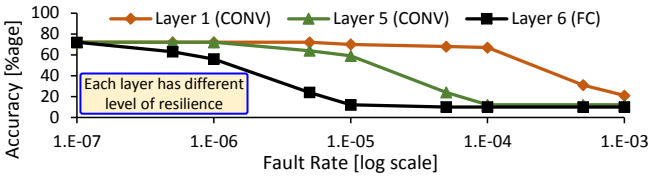


Fig. 2. Resilience analysis of different layers of the AlexNet trained for the CIFAR-10 dataset [27].

Injection (HCI), Time-Dependent Dielectric Breakdown (TDDB) and Electromigration (EM). It typically results in circuits becoming slower with time, e.g., by increasing the threshold voltage (V_{TH}) of the circuits [34], or breakdown of dielectric and wires. The aging faults manifest as timing errors during the early stages, and later can transform into permanent faults. Alongside various other factors, the rate of aging usually increases with temperature.

- **Process Variations** is an issue caused by the variations introduced during the manufacturing process. This issue arises due to the fact that it is difficult to manufacture transistors with the same properties, such as channel length, oxide thickness and doping levels, at the nano-scale. This results in inter-chip and intra-chip variations, which affect the performance (e.g., in the form of reduced operating frequency) and power efficiency, as well as the yield of the manufacturing process (in case of permanent faults).

III. COST-EFFECTIVE RESILIENCE AGAINST HARDWARE-INDUCED RELIABILITY THREATS

Table I summarizes several prominent techniques designed to improve the resilience of DNN-based systems against hardware-induced faults. The techniques are developed based on the inherent characteristics of the DNNs to limit the overhead cost. In the following subsections, we discuss the techniques used for mitigating different types of hardware-induced faults, and how they can be integrated together to offer protection against multiple types of reliability threats.

A. Soft Error Mitigation

Soft errors manifest as random bit-flips in a system that can propagate to the application layer and may lead to incorrect output or system failure. Although DNNs are considered highly resilient, these faults, when occur at critical locations, can result in undesirable results and may have severe consequences, specifically in applications like autonomous driving, banking, healthcare, and aviation. Different parts of a DNN show a different level of resilience to faults (see Fig. 2). Frameworks like BinFI [35] and Ares [36] have been proposed to compute the resilience of each part of a given DNN and identify the critical bits/computations. This analysis helps in introducing selective protection and limit the overhead cost of redundancy.

Various techniques have been proposed to mitigate the effects of soft errors. Radiation hardening of particular nodes [37] (identified using resilience analysis) can significantly boost the fault tolerance of a system at a low cost. Based on the fact that a bit-flip from 1-to-0 is relatively less critical [38], modifications have been proposed for SRAM cells that force the cells to 0s in case of radiation-induced errors [30]. However, these techniques require modifications in all/most of the SRAM cells and, therefore, can result in a high-to-moderate level of overheads. Recently, range restriction-based techniques have emerged, which locally detect the faults that can influence the output and restrict them from propagating into the network [26] [27]. The flow of these techniques is illustrated in Fig. 3. Note that, as these techniques do not require additional training or hardware, they incur low overheads. Fig. 4 presents our analysis that shows the effectiveness of these techniques when applied to the AlexNet trained for the CIFAR-10 dataset.

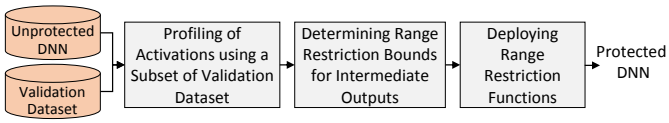


Fig. 3. Overview of range restriction-based fault-mitigation techniques

B. Permanent Fault Mitigation

Manufacturing-induced defects / permanent faults are static in nature and can be detected through post-fabrication testing. The extracted fault maps/statistics can then be used to improve the resilience of the DNNs, thereby enabling reliable execution on a faulty hardware. One of the promising techniques towards this is fault-aware training [22] [23], where the training process takes into consideration the information of faults and train the network accordingly. The flow of fault-aware training is shown in Fig. 5. Although fault-aware training offers a significant improvement in the resilience of a DNN, it cannot be used in all possible circumstances. For example, if the complete training

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on January 25,2022 at 04:01:02 UTC from IEEE Xplore. Restrictions apply.

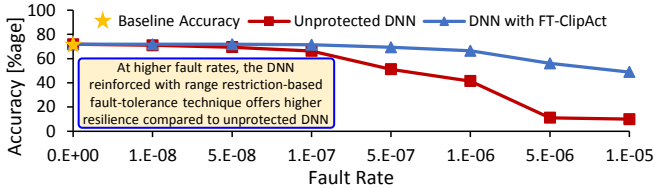


Fig. 4. Resilience of the AlexNet (trained for the CIFAR-10 dataset) with and without range restriction-based fault mitigation, based on our analysis in [27].

dataset is not available, even after training, the network will provide sub-optimal results. Moreover, in the case of a large number of faulty chips, where each chip has a different fault map, the training overhead can be significant. Techniques like fault-aware pruning [23] can be used without retraining in such scenarios. This technique is mainly designed to mitigate permanent faults in the computational units of a systolic array-based DNN hardware. Fig. 8a shows the hardware architecture of a systolic array [13], and Fig. 8b shows the design-time architectural modifications required in the processing elements to enable fault-aware pruning at run-time. These techniques offer limited improvement in the resilience of a system. Fault-aware mapping [24] has recently been proposed to overcome these limitations and to enable the system to maintain its baseline accuracy by adapting the mapping of the computations based on hardware faults. Fig. 6 shows an overview of the flow of our fault-aware mapping technique [24]. Its working principle is illustrated in Fig. 7. Fig. 7b shows the default mapping of four 2x2 filters (shown in Fig. 7a) onto an un-faulty 4x4 computational array. In case there are faulty MAC units in the array, the default mapping strategy might map critical weights (i.e., weights having higher magnitude) on those units (see Fig. 7c), which are then pruned due to fault-aware pruning. Fig. 7d shows that, by reshuffling the filters, the relatively ineffectual weights can be mapped on the fault MAC units to reduce the accuracy drop in case of fault-aware pruning. Fig. 9 shows the effectiveness of fault-aware mapping + pruning compared to a simple fault-aware pruning technique. Note that these techniques are orthogonal to the ones discussed for soft error mitigation and, therefore, can be combined with them to offer a wider coverage of reliability threats.

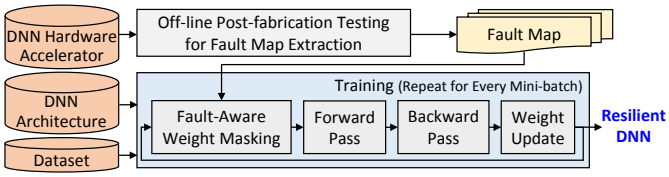


Fig. 5. Overview of the flow of fault-aware training process.

C. Timing Error Mitigation

In the earlier years, the aging of a circuit results in timing errors. Zhang et al. in [25] proposed ThunderVolt, a technique to detect and mitigate these errors in the MAC units of the processing array. The technique uses Razor flip-flops to detect the errors. In case an error is detected, it steals a cycle from the subsequent processing element by dropping its computation to discard the erroneous output and pass-on the approximately correct output. Fig. 8c shows the hardware modification required in the processing elements to realize the technique. Note that ThunderVolt is mainly developed to improve the energy efficiency of a DNN-based system through voltage scaling. However, as it mainly mitigates the effects of timing errors, it can be used to mitigate aging-induced faults.

IV. Challenges on the Road Ahead

Following are some of the key challenges and future directions that are envisioned to play a vital role in realizing dependable DL-based systems.

1) **Frameworks for effective integration of fault-mitigation techniques:** Different systems show a different level of resilience to each type of hardware-induced faults. Therefore, each system may require a different set of fault-mitigation techniques to offer reliable functionality. Towards this, frameworks need to be developed that can efficiently analyze the resilience of DNN-based systems to different types of reliability threats (preferably, in a synergistic way) and propose a sub-set of the available techniques that can be integrated together to offer optimal / cost-effective resilience while meeting the pre-defined resource constraints.

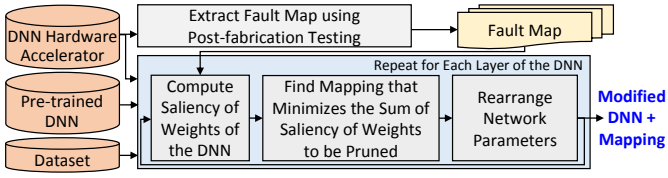


Fig. 6. Overview of the flow of our fault-aware mapping process.

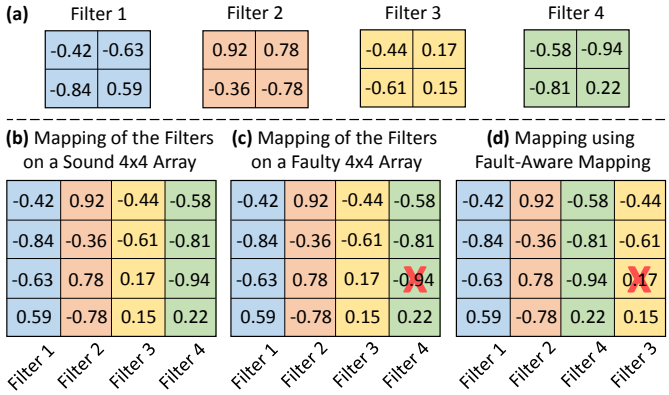


Fig. 7. An example illustrating the working principle of fault-aware mapping.

2) **Design Methods for Building Reliable DNNs:** Similar to the research in building resource-efficient DNNs, there is a need for building automated design methodologies that can generate robust DNN models. Towards this, a thorough design space exploration is required to identify the key design principles and building blocks that help in improving the resilience of DNNs. Afterward, methodologies have to be developed that can employ these principles to build reliable and resource-efficient DNNs.
3) **Test and Verification:** The increasing use of DNNs in safety-critical applications has raised the need for "complete" verification of these models. Frameworks are being developed that can formally verify their functionality. However, most of the frameworks focus on small-scale DNNs, or do approximate or incomplete verification. A considerable amount of effort is required in this direction to build frameworks that can efficiently verify the functionality of large-scale DNNs with realistic assumptions. One of the recent studies in this direction is [39], for more details see [40]. An alternate direction towards the verification of ML systems is *Interpretable Machine Learning / Explainable AI*, which focuses on developing models that can be interpreted by humans.
4) **Reliability for SNNs:** Inspired by the spiking-based event-driven information processing in the brain, the Spiking Neural Networks (SNNs) promise to offer low power consumption and high-speed inference when processed using massively parallel neuromorphic hardware [41] [42]. The underlying functions in SNNs and the need for a different type of hardware architecture make them distinct from DNNs. These distinctions ask for novel techniques to analyze and mitigate the effects of hardware-induced reliability threats in SNN-based systems.

Acknowledgement

We would like to acknowledge Intel's Grant for the project "*Cost-Effective Dependability for Deep Neural Networks and Spiking Neural Networks*".

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on January 25,2022 at 04:01:02 UTC from IEEE Xplore. Restrictions apply.

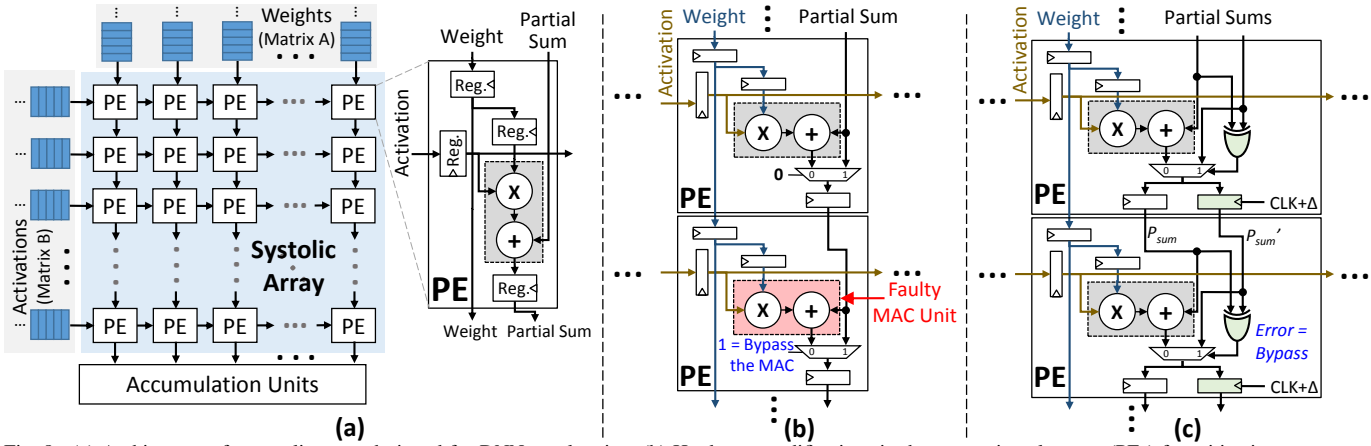


Fig. 8. (a) Architecture of a systolic array designed for DNN acceleration. (b) Hardware modifications in the processing elements (PEs) for mitigating permanent faults. (c) Hardware modifications in the PEs for mitigating timing faults. (Adapted from [23]–[25])

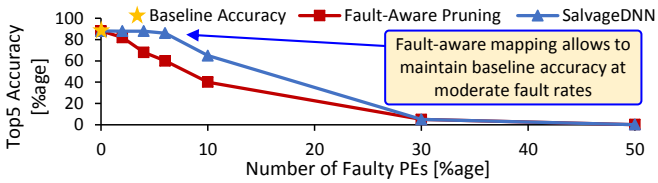


Fig. 9. Comparison between fault-aware pruning and fault-aware mapping + pruning (i.e., SalvageDNN) using VGG11 trained for the ImageNet dataset and deployed on a 256x256 faulty systolic array-based DNN hardware [24].

References

[1] Y. LeCun et al., "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[2] Y. Guo et al., "Deep learning for visual understanding: A review," *Elsevier Neurocomputing*, vol. 187, pp. 27–48, 2016.

[3] H. I. Fawaz et al., "Deep learning for time series classification: a review," *Data Mining and Knowledge Discovery*, vol. 33, no. 4, pp. 917–963, 2019.

[4] A. Marchisio et al., "Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges," in *IEEE **ISVLSI***, 2019, pp. 553–559.

[5] D. S. et al., "Mastering the game of go without human knowledge," ***Nature***, vol. 550, no. 7676, pp. 354–359, 2017.

[6] M. Al-Qizwini et al., "Deep learning algorithm for autonomous driving using googlenet," in *IEEE **IV** Symposium*, 2017, pp. 89–96.

[7] B. Huval et al., "An empirical evaluation of deep learning on highway driving," *arXiv preprint arXiv:1504.01716*, 2015.

[8] S. Mozaffari et al., "Deep learning-based vehicle behaviour prediction for autonomous driving applications: A review," *arXiv preprint arXiv:1912.11676*, 2019.

[9] R. Miotto et al., "Deep learning for healthcare: review, opportunities and challenges," *Briefings in bioinformatics*, vol. 19, no. 6, pp. 1236–1246, 2018.

[10] A. Esteva et al., "A guide to deep learning in healthcare," *Nature medicine*, vol. 25, no. 1, p. 24, 2019.

[11] B. S. Prabakaran et al., "Emap: A cloud-edge hybrid framework for eeg monitoring and cross-correlation based real-time anomaly prediction," *arXiv preprint arXiv:2004.10491*, 2020.

[12] V. Sze et al., "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[13] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *ACM/IEEE **ISCA***, 2017, pp. 1–12.

[14] H. Kwon et al., "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *ACM **ASPLOS***, 2018, pp. 461–475.

[15] Y. Chen et al., "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[16] "Road vehicles – Functional Safety," International Organization for Standardization, Geneva, CH, Standard, 2011.

[17] R. Vadlamani et al., "Multicore soft error rate stabilization using adaptive dual modular redundancy," in *IEEE **DATE***, 2010, pp. 27–32.

[18] R. E. Lyons et al., "The use of triple-modular redundancy to improve computer reliability," *IBM journal of research and development*, vol. 6, no. 2, pp. 200–209, 1962.

[19] X. Vera et al., "Selective replication: A lightweight technique for soft errors," *ACM **TOCS***, vol. 27, no. 4, pp. 1–30, 2010.

[20] M. Shafique et al., "Exploiting program-level masking and error propagation for constrained reliability optimization," in *ACM/IEEE **DAC***, 2013, pp. 1–9.

[21] T. Lin et al., "Microsoft coco: Common objects in context," in ***ECCV***. Springer, 2014, pp. 740–755.

[22] S. Kim et al., "Energy-efficient neural network acceleration in the presence of bit-level memory errors," *IEEE **TCAS-I***, vol. 65, no. 12, pp. 4285–4298, 2018.

[23] J. J. Zhang et al., "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator," in *IEEE **VTS***. IEEE, 2018, pp. 1–6.

[24] M. Hanif at al., "Salvagednn: salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2164, 2020.

[25] J. Zhang et al., "Thundervolt: enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators," in *ACM/IEEE **DAC***, 2018, pp. 1–6.

[26] Z. Chen et al., "Ranger: Boosting error resilience of deep neural networks through range restriction," *arXiv preprint arXiv:2003.13874*, 2020.

[27] L. Hoang et al., "Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation," *IEEE **DATE***, 2020.

[28] K. Zhao et al., "Algorithm-based fault tolerance for convolutional neural networks," *arXiv preprint arXiv:2003.12203*, 2020.

[29] J. Guo et al., "Novel low-power and highly reliable radiation hardened memory cell for 65 nm cmos technology," *IEEE **TCAS-I***, vol. 61, no. 7, pp. 1994–2001, 2014.

[30] A. Azizimazreah et al., "Tolerating soft errors in deep learning accelerators with reliable on-chip memory designs," in *IEEE **NAS***, 2018, pp. 1–10.

[31] L.-C. Chu et al., "Fault tolerant neural networks with hybrid redundancy," in *IEEE **IJCNN***. IEEE, 1990, pp. 639–649.

[32] S. Shankland, "Meet Tesla's self-driving car computer and its two AI brains," https://www.cnet.com/news/meet-tesla-self-driving-car-computer-and-its-two-ai-brains/.

[33] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE **T-DMR***, vol. 5, no. 3, pp. 305–316, 2005.

[34] K. Kang et al., "Nbti induced performance degradation in logic and memory circuits: How effectively can we approach a reliability solution?" in *ACM/IEEE **ASP-DAC***, 2008, pp. 726–731.

[35] Z. Chen et al., "Binfi: an efficient fault injector for safety-critical machine learning systems," in *ACM **HPCA***, 2019, p. 69.

[36] B. Reagen et al., "Ares: A framework for quantifying the resilience of deep neural networks," in *ACM/IEEE **DAC***, 2018, pp. 17:1–17:6.

[37] D. B. Limbrick et al., "Reliability-aware synthesis of combinational logic with minimal performance penalty," *IEEE Transactions on nuclear science*, vol. 60, no. 4, pp. 2776–2781, 2013.

[38] M. A. Hanif et al., "Robust machine learning systems: Reliability and security for deep neural networks," in *IEEE **IOLTS***, 2018, pp. 257–260.

[39] M. Naseer et al., "Fannet: Formal analysis of noise tolerance, training bias and input sensitivity in neural networks," *IEEE **DATE***, 2020.

[40] M. Shafique et al., "Robust machine learning systems: Challenges,current trends, perspectives, and the road ahead," *IEEE **D&T***, vol. 37, no. 2, pp. 30–57, 2020.

[41] M. Davies et al., "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE **Micro***, vol. 38, no. 1, pp. 82–99, 2018.

[42] R. Massa et al., "An efficient spiking neural network for recognizing gestures with a dvs camera on the loihi neuromorphic processor," *IEEE **IJCNN***, 2020.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on January 25,2022 at 04:01:02 UTC from IEEE Xplore. Restrictions apply.

Analyzing and Mitigating the Impact of Permanent Faults on a Systolic Array Based Neural Network Accelerator

Jeff (Jun) Zhang Tianyu Gu Kanad Basu Siddharth Garg
Department of Electrical and Computer Engineering, New York University, Brooklyn, New York, 11201
Email: {jeffjunzhang,tg1553,kb150, sg175}@nyu.edu

arXiv:1802.04657v2 [cs.LG] 17 Feb 2018

***Abstract*—Due to their growing popularity and computational cost, deep neural networks (DNNs) are being targeted for hardware acceleration. A popular architecture for DNN acceleration, adopted by the Google Tensor Processing Unit (TPU), utilizes a systolic array based matrix multiplication unit at its core. This paper deals with the design of fault-tolerant, systolic array based DNN accelerators for high defect rate technologies. To this end, we empirically show that the classification accuracy of a baseline TPU drops significantly even at extremely low fault rates (as low as 0.006%). We then propose two novel strategies, fault-aware pruning (FAP) and fault-aware pruning+retraining (FAP+T), that enable the TPU to operate at fault rates of up to 50%, with negligible drop in classification accuracy (as low as 0.1%) and no run-time performance overhead. The FAP+T does introduce a one-time retraining penalty per TPU chip before it is deployed, but we propose optimizations that reduce this one-time penalty to under 12 minutes. The penalty is then amortized over the entire lifetime of the TPU's operation.**

1. Introduction

Deep neural networks (DNN) have, in the past few years, surpassed the performance of traditional machine learning algorithms and are now considered as state-of-the-art for a range of applications, including image and video recognition, text classification [1], [2] and language translation [3]. One drawback of DNNs, however, is their complexity. State-of-the-art DNNs have millions of parameters and are computationally expensive, both to train and execute. GPU acceleration is one way to mitigate this computational burden. Nonetheless, there is a growing interest in *special purpose hardware accelerators* for DNN execution to achieve even greater performance and energy efficiency. Examples of recent special-purpose DNN accelerators include [4]–[7].

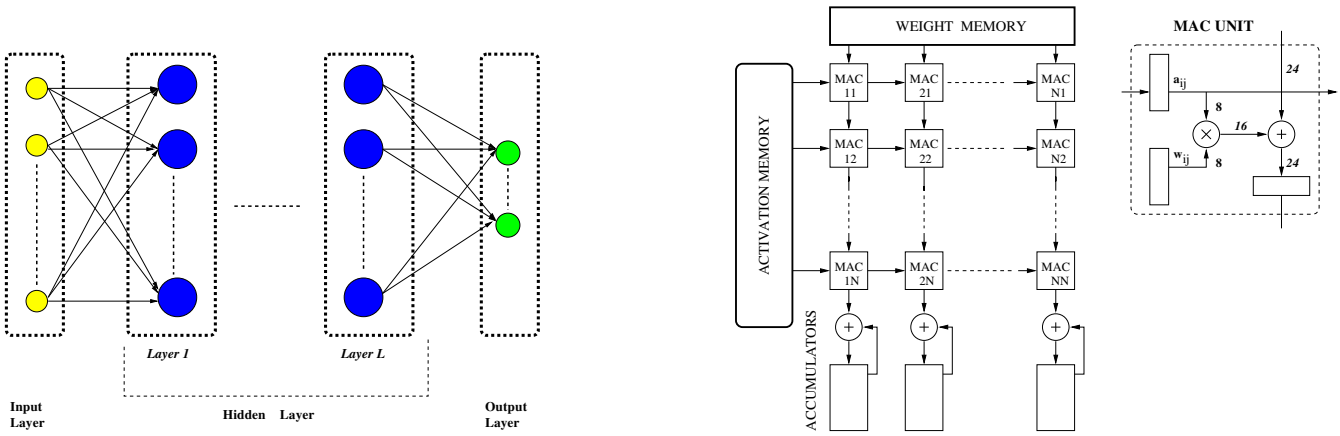
DNNs contain multiple layers of computation. Each layer comprises of a computationally expensive matrix multiplication or convolution operation, the latter for convolutional neural networks (CNN), followed by an inexpensive non-linear "activation." Work dating back to the 1980s has shown that matrix multiplications and convolutions can be efficiently implemented using *systolic arrays* [8]. A systolic array is a grid of connected processing elements (PE) that only communicate with their neighbors. For matrix multiplications and convolutions, each PE simply performs a multiply-and-accumulate (MAC) operation and passes the accumulated value downstream. Data is streamed through the array in a synchronized manner such that each PE received its inputs at just the right time, thus obviating the need for input buffering or complex routing. Further, the cost of reading inputs from memory is amortized over several compute cycles, providing high energy efficiency.

An example of a systolic array based DNN accelerator is the Google Tensor Processing Unit (TPU), that uses 256×256 grid of MAC units at its core, and provides between $30\times$ to $80\times$ times greater performance than CPU or GPU based servers [6]. The TPU is widely deployed in Google datacenters to accelerate DNN inference while reducing the total datacenter energy consumption. In this paper, we use the TPU architecture from Google as the baseline design, but our proposed techniques can apply to any systolic array based DNN accelerators. In the rest of the paper, we will use TPU to refer to the general class of systolic array based DNN accelerators.

An important challenge with future technology scaling is the increase in fault rates, including both permanent (hard errors) and temporary faults (soft errors) [9]–[12]. Temporary faults might occasionally impact the DNN's classification results, but their overall impact on classification accuracy is small, even at high soft error rates. On the other hand, permanent faults, as we show in this paper, can affect the result of *every* DNN execution and significantly reduce the classification accuracy. While permanent faults, at least those that are related to manufacturing defects, can be identified during post-fabrication testing, discarding every chip with a permanent fault reduces yield. This paper addresses the design of *fault-tolerant TPU*, with a focus on manufacturing defects or process variation induced permanent faults.

Prior work has suggested that several machine learning (ML) applications, including DNNs, are inherently resilient to errors; One potential solution for addressing permanent faults in a TPU, therefore, is to simply ignore the permanent faults in the hope that the application itself is inherently resilient to errors. Indeed, prior work has suggested that this is the case for several machine learning (ML) applications, including DNNs [13]. However, using detailed gate-level simulations of stuck-at permanent faults, we show that the accuracy of the TPU drops sharply even if as few as four out of the approximately 64K MAC units are faulty.

Next, we propose two new solutions: *fault-aware prun-*



(a) DNN Architecture (b) TPU Architecure

Figure 1: Illustration of DNN Architecture and TPU.

ing (FAP) and *fault-aware pruning plus retraining* (FAP+T) that enable the TPU to operate at high fault rates of up to 50% with a negligible impact on classification accuracy and no run-time performance overhead compared to the defect-free baseline. Our proposed solutions build on the recent work that shows that a significant fraction of a DNN's connections can be *pruned* with no (or limited) impact on accuracy. However, while the prior work using pruning to reduce DNN execution time and memory usage [14]–[18], we do so to enable *fault tolerance*. FAP prunes all connections in a DNN that map to faulty MACs using simple bypass circuitry that requires only minor modifications to the baseline TPU. FAP+T additionally *retrains* the DNN after pruning to restore classification accuracy back or close to its baseline, but comes at the expense of extra "test time" per TPU chip. To the best of our knowledge, ours is the *first* work that analyzes and proposes solutions to mitigate the impact of permanent faults on TPU (or TPU-like) DNN accelerators.

2. Related Work

Our work falls in the area of fault-tolerant digital system design. The prior work in this area has a rich history, starting with early work on addressing permanent faults in memory systems using error correcting codes (ECC) [19], and redundancy based approaches for addressing faults in logic [20]–[23] and in the routing fabric [24], [25]. However, these techniques do not directly apply to systolic arrays.

Kung et al. [26] were the first to describe fault-tolerant systolic array designs, which were later improved upon by [27]. The basic idea is to view a systolic array with faults as a smaller systolic array with *fewer* rows and columns. These solutions, in addition to requiring complex bypassing and additional registers inside PEs, can have a significant performance penalty at high-defect rates since an entire column/row is eliminated for each faulty PE. More sophisticated solutions can reduce the performance penalty, but at even higher design complexity [28], [29]. It is important to note that, as opposed to FAP and FAP+T (our proposed solutions), all of these techniques are *application agnostic*, that is, they seek to correctly execute the original task/application in the presence of faults. FAP and FAP+T, on the other hand, are application-aware, that is, we modify the underlying DNN architecture using pruning and re-training to adapt to faults in the TPU. Consequently, FAP and FAP+T have no performance penalty and negligible area overhead.

Another line of related work is on the design of crossbar-based DNN accelerators leveraging emerging technologies like phase change memory (PCM), resistive RAM (RRAM) and spin-based devices [30], [31]. Specifically, prior work has proposed techniques to cope with non-ideal device characteristics including device non-linearities, limited dynamic range and process variations. However, these techniques are specific to the devices used and to crossbar architectures, and do not apply to TPU designs [32], [33].

3. Background and Preliminaries

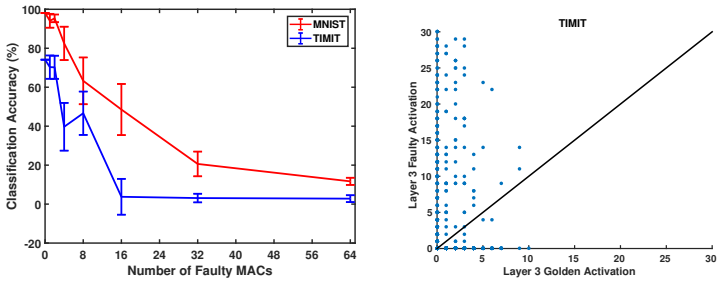
In this section, we breifly describe the relvant background on DNNs and the design of our baseline hardware accelerator for DNN execution.

3.1. Deep Neural Networks

A DNN consists of L stacked layers of computation, as shown in Figure 1a. Layer l has N^l *neurons* whose outputs are referred to as *activations*, represented by an N^l dimensional vector a^l. Each layer multiplies the vector of activations from the previous layer with a weight matrix w^l of dimensions $N^{l-1} \times N^l$ and adds constant biases represented by an N^l dimensional vector b^l, followed by an element-wise activation function ϕ. Mathematically, each layer of a DNN performs the following operation:

$$y_i^l = \sum_j w_{i,j}^l a_j^{l-1} + b_i^l, \quad a_i^l = \phi(y_i^l) \quad \forall l \in [1, L], \quad (1)$$

In our description so far, we have assumed that each neuron's activation depends on the activations of all neurons in the previous layers. Such layers are referred to as *fully connected* (FC) layers. A commonly used special case of an FC layer is a convolutional (conv) layer. The activations of a



(a) Classification Accuracy Drop Due to Stuck-at-Fault MACs.

(b) TIMIT Output Regression for Layer 3 Activation with 8 Faulty MACs.

Figure 2: Impact of TPU Stuck-at-Faults on DNN Applications

convolutional neural network are represented as 3-D *tensors* of dimensions $W^l \times H^l \times D^l$ which refer to the width, height and number of channels in layer l of the network. Also, the weights are represented as 4-D tensors of dimensions $F^l \times F^l \times D^{l-1} \times D^l$, where F^l is the filter size in layer l. While the mathematical expression of the convolutional layer is omitted for brevity; for the purposes of this paper it suffices to know that the outputs of channel $d \in D^l$ is determined by convolving the 3-D input tensor with a 3-D weight tensor obtained by setting the last dimension of the 4-D weight tensor to d.

3.2. DNN Acceleration on TPU

Figure 1b shows a block-diagram of the TPU architecture, at the heart of which is a systolic array containing $N \times N$ MAC units that is used to perform the computationally expensive matrix multiplication and convolution operations. To understand how, consider a fully-connected layer with N input neurons and N output neurons, and consequently an $N \times N$ weight matrix. The weight matrix is first loaded into the systolic array, with weight $w_{i,j}$ being loaded into the MAC in row j and column i, which we refer to as $\text{MAC}_{\text{i,j}}$.

Now consider the first column of the TPU. $\text{MAC}_{1,1}$ computes $w_{1,1}a_1$ in the first clock cycle and sends the result to $\text{MAC}_{1,2}$ which adds the product $w_{1,2}a_2$ to its input and forwards the result further downwards. In clock cycle N, the $\text{MAC}_{1,\text{N}}$ unit outputs $y_1 = \sum_{i=1}^{N} w_{1i}a_i$, which is the the first element of the output matrix.

The second column receives the same stream of inputs as the first column, but delayed by one clock cycle. This column is loaded with weights from the second row of the weight matrix, and outputs y_2, and so on. In practice, the TPU operates in *batches*, where each batch has B inputs. A batch of B inputs is multiplied by an $N \times N$ weight matrix in $2N + B$ clock cycles.

4. Motivational Analysis: Impact of Permanent Faults on TPU

To motivate the proposed fault-tolerant TPU design, we begin by empirically analyzing the impact of permanent faults on the classification accuracy of the TPU. To do so, we first synthesized an RTL description of the TPU into the gate-level netlist using the 45 nm OSU PDK, and then inserted stuck-at faults at internal nodes in the gate-level netlist. For this analysis, we focused only on faults in the data-path and ignored faults in the memory components (since they can be addressed using ECC) and the control logic since it consumes an insignificant fraction of the design.

We then mapped DNNs for two classification tasks, MNIST digit classification and TIMIT speech recognition, on the faulty TPU and plot the classification accuracy versus number of faulty MAC units in Figure 2a. For TIMIT, we observe that even with only four faulty MACs (i.e., with only $\sim 0.005\%$ MACs faulty), the classification accuracy drops from the 74.13% to 39.69%.

The reason for the drop in accuracy is that stuck-at faults frequently affect the higher order bits of the MAC output, resulting in large absolute errors in the matrix-vector product. Figures 2b scatters the golden (fault-free) activations of the final layer of the TIMIT DNNs with the corresponding faulty outputs. Observe that for TIMIT, the faulty outputs have much higher magnitudes than the golden outputs.

To mitigate the permanent faults on the TPU, one simple idea is to bypass the entire columns where the faulty MACs reside. However, with the number of permanent faults increasing, the performance penalty would be unacceptable.

These observations together motivates the fault aware design of this paper.

5. Proposed Fault-Tolerant TPU Design

From the motivational example, it is clear that TPUs with even a relatively small number of faulty MAC units cannot be used unless the fault impact is mitigated. Our proposed fault-tolerant TPU design starts with an important observation: note from the description of the TPU operation in Section 3.2 that *each weight in the DNN maps to exactly one MAC unit.* In other words, there is a static mapping between DNN weights and MAC units. We then exploit the static mapping to determine which weights to prune.

This observation can be formalized using mapping functions $r()$ and $c()$ that take as input the indices of a DNN weight and output the row and column, respectively, of the MAC unit on which the weight is mapped. Specifically, based on the discussion in Section 3.2, the mapping functions for weight $w_{i,j}$ in a fully-connected layer are $r(i,j) = j\%N$ and $c(i,j) = i\%N$, where $\%$ is the modulo operator. Note that implicit in this mapping function is the fact that weight matrices that do not fit fully in the systolic array are first *blocked* into smaller $N \times N$ sub-matrices.

Our mapping strategy for conv layers sums along input channels along rows of the TPU, and each column computes outputs for a different output channel. Consequently, the mapping functions for weight $w_{i,j,k,l}$ in a conv layer are $r(i,j) = k\%N$ and $c(i,j) = l\%N$. We now describe our two proposed fault-tolerance methodologies: FAP and FAP+T.

5.1. Fault-Aware Pruning (FAP)

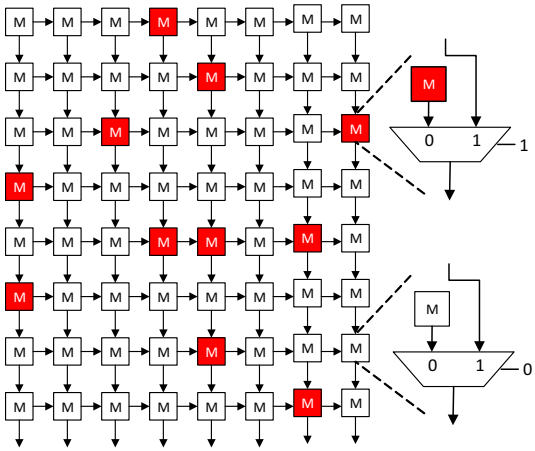


Figure 3: Diagram illustrating FAP.

The proposed FAP and FAP+T techniques both assume that standard post-fabrication tests are used on each TPU chip to determine the location of faulty MACs. Given this information, the idea behind FAP is to set (or prune) any weight that maps to a faulty MAC to zero. That is, for all pairs of (i, j) values such that $\text{MAC}_{c(i,j),r(i,j)}$ is faulty, we set the corresponding $w_{i,j} = 0$ (this is for fully connected layers, a similar strategy is used for conv layers). Note that multiple weights can map to one MAC unit; correspondingly, even a single faulty MAC can result in multiple weights being pruned.

In hardware, pruning is achieved by introducing a separate bypass path for faulty MAC units, as shown in Figure 3. With the bypass path being enabled, the faulty MAC unit's contribution to the column sum is skipped, which is equivalent to setting the faulty MAC's weight to zero. Note that actually loading a zero weight into a faulty MAC is *not* equivalent to setting it's weight to zero. The area overhead due to the new bypass path is only about 9%.

5.2. FAP+T Solution

The FAP+T approach starts with FAP based on each TPU's fault map, but additionally *retrains* the unpruned weights in the DNN while forcing all pruned weights to zero during the re-training process. The FAP+T algorithm (see Algorithm 1) returns new, optimized values for the unpruned weights that improve the classification accuracy compared to the FAP solution. One drawback, however, is that re-training needs to be performed for each TPU chip based on its unique fault map; however, this needs to be done *only once* per TPU chip and the cost of doing so is amortized over the entire lifetime of the TPU chip.

Note that the FAP+T algorithm has a parameter MAX_EPOCHS that determines the number of iterations of the re-training algorithm. Setting this parameter to zero is equivalent to FAP. As MAX_EPOCHS is increased, the re-training time increases in return for increased classification accuracy.

Algorithm 1: FAP+T Training Algorithm

Algorithm `FAP+T()`
Load the pre-trained DNN weights and TPU fault map;
Determine indices of pruned weights from TPU fault map;
Set all pruned weights to zero;
for *Training epochs* $\leq MAX_EPOCHS$ **do**
  Update weights using back-prop.;
  Set all pruned weights to zero;
end
return *Retrained model;*

Multi-Layer Perceptrons (MLPs)

Name	MNIST	TIMIT
# Neurons	784-256-256-256-10	1845-2000-2000-2000-183

AlexNet Convolutional Feature Extraction

layer	filter	stride	padding	activation
conv1	96x3x11x11	4	0	ReLU+LRN
pool1	max, 3x3	2	0	/
conv2	256x96x5x5	1	2	ReLU+LRN
pool2	max, 3x3	2	0	/
conv3	384x256x3x3	1	1	ReLU
conv4	384x384x3x3	1	1	ReLU
conv5	256x384x3x3	1	1	ReLU
pool5	max, 3x3	2	0	/

AlexNet Fully-connected Layers

layer	#neurons	activation
fc6	4096	ReLU
fc7	4096	ReLU
fc8	4096	/

TABLE 1: Benchmark DNNs architecture

6. Empirical Evaluation

In this section, we introduce our evaluation framework and discuss the results on three popular DNN applications.

6.1. Setup

Benchmarks The three benchmarks we used are: MNIST digit classification, TIMIT speech recognition [34] and AlexNet for image recognition using the PASCAL VOC2007 dataset [35]. We trained all DNNs from scratch and achieve the-state-of-art classification accuracy. Table 1 shows the detail parameters of the three DNN architectures. The MNIST and TIMIT DNNs have only fully connected layers, while AlexNet has five conv layers and three fully connected layers.

Systolic Array Parameters A prototype of the systolic array of a grid of 256×256 MACs was developed in synthesizable Verilog and synthesized with OSU FreePDK 45 nm standard cell library using the Cadence Genus synthesis tool. The systolic array is synthesized to be run at 658 MHz with a nominal supply voltage of $1.1V$ and consumes $19.7W$ of dynamic power.

Simulation Methodology We scheduled our benchmark DNNs on the systolic array using the mapping techniques described in Section 5. To analyze the impact of permanent

faults on the TPU, we use stuck-at fault injection methodology described in Section 4. We then conduct a detailed gate-level simulation in Modelsim to collect the final accuracy for the benchmarks. Each experiment is repeated 10 times with faults injected in different locations, picked uniformly at random.

6.2. Results

Figure 4 shows the classification accuracy versus the number of permanent faults using the FAP and FAP+T techniques for our three benchmark DNNs. Note that we have about 65K MAC units in the TPU, and show results for high defect rates of up to 50%, i.e., with upto half of the MAC units faulty. For MNIST and TIMIT, both FAP and FAP+T are able to go up to fault rates of 25% with negligible classification accuracy reduction. However, with 50% faulty MACs, only FAP+T is able to provide close to the baseline classification accuracy, while that of the FAP technique reduces.

Compared to MNIST/TIMIT, the classification accuracy for AlexNet as seen in Figure 4b, drops markedly with increasing fault rates with FAP. FAP+T, however, performs much better and only incurs an 8% accuracy drop even with a 50% fault rate. One reason for the accuracy drop comes from the scheduling policy for the CNNs. In the current policy, one permanent faulty MAC would lead to a whole channel of the filter to be pruned. A more sophisticated scheduling may show better results but it is out of the scope of this paper.

As mentioned before, FAP+T incurs a one-time re-training overhead per TPU chip. While the retraining overhead (about 1 hour for AlexNet in the worst case, 25 epochs) is negligible compared to the lifetime of the TPU, we show in Figure 5 that it can be reduced even further by setting the value of MAX_EPOCHS appropriately. Specifically, Figure 5 shows the classification accuracy of FAP+T with increasing number of training epochs. Observe, for instance, that for AlexNet with 25% faulty MACs, the classification accuracy after the first 5 epochs only marginally lower than the classification accuracy after 25 epochs. Consequently, the re-training time for this example of AlexNet can be reduced by $5\times$, from 1 hour to only 12 minutes.

7. Conclusion

In this paper, we address the problem of designing fault-tolerant hardware accelerators for DNN execution for high defect rate technologies. Using the systolic array based Google TPU architecture as a baseline, we first showed using detailed gate-level simulations that the TPU's classification accuracy drops significantly even in the presence of very low fault rates. Then, we proposed two novel techniques, FAP and FAP+T, that allow TPUs to operate even with fault rates as high as 50%, with negligible to tolerable drops in classification accuracy (from 0.1% drop for TIMIT to 8% drop for AlexNet). While neither technique has any run-time performance overhead, FAP+T introduces a one-time retraining overhead of about 12 minutes per TPU chip for AlexNet. However, this one-time cost is amortized over the entire TPU's lifetime. As future work, we plan to address the impact of aging-related faults on DNN accelerators. Also, we are planning to apply our proposed methodology on larger neural-networks.

References

[1] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 1097–1105, 2012.

[2] A. Karpathy *et al.*, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 1725–1732, 2014.

[3] I. Sutskever *et al.*, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 3104–3112, 2014.

[4] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[5] Z. Du *et al.*, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, 2015.

[6] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, ACM, 2017.

[7] S. Park *et al.*, "4.6 a1. 93tops/w scalable deep learning/inference processor with tetra-parallel mimd architecture for big-data applications," in *IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 1–3, 2015.

[8] H.-T. Kung, "Why systolic architectures?," *IEEE computer*, vol. 15, no. 1, pp. 37–46, 1982.

[9] S. S. Sahoo *et al.*, "Design and evaluation of reliability-oriented task re-mapping in mpsocs using time-series analysis of intermittent faults," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 798–803, IEEE, 2016.

[10] J. Zhang *et al.*, "Enabling extreme energy efficiency via timing speculation for deep neural network accelerators," 2017.

[11] S. Borkar, "Design perspectives on 22nm cmos and beyond," in *Proceedings of the 46th Annual Design Automation Conference*, pp. 93–94, ACM, 2009.

[12] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[13] A. Gebregiorgis *et al.*, "Error propagation aware timing relaxation for approximate near threshold computing," in *Proceedings of the 54th Annual Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2017.

[14] S. Han *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

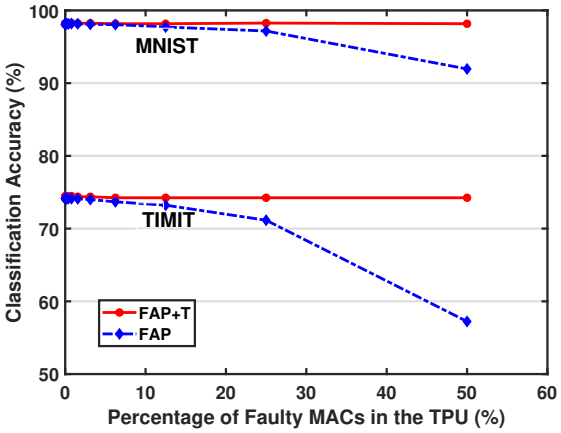
[15] J. Yu *et al.*, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 548–560, ACM, 2017.

[16] H. Li *et al.*, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

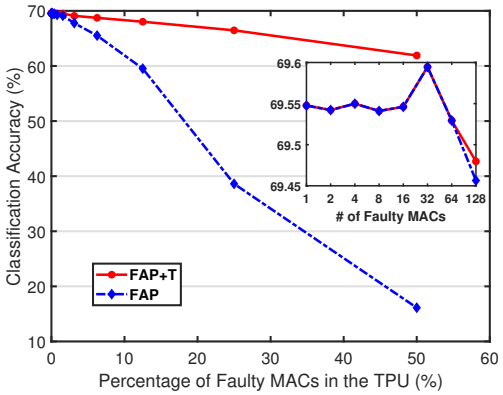
[17] S. Anwar *et al.*, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 32, 2017.

[18] P. Molchanov *et al.*, "Pruning convolutional neural networks for resource efficient inference," 2016.

[19] L. Levine *et al.*, "Special feature: Semiconductor memory reliability with error detecting and correcting codes," *Computer*, vol. 9, no. 10, pp. 43–50, 1976.

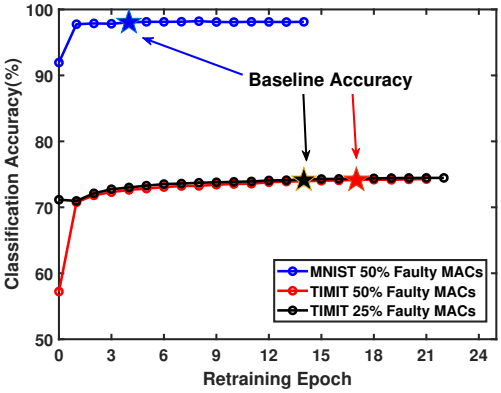


(a) MNIST and TIMIT.

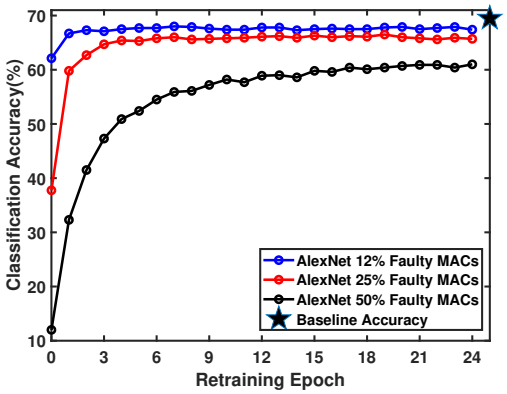


(b) AlexNet.

Figure 4: Classification accuracy vs. Percentage of Faulty MACs using FAP and FAP+T for (a) MNIST and TIMIT and (b) AlexNet.



(a) MNIST and TIMIT.



(b) AlexNet.

Figure 5: Classification accuracy vs. MAX_EPOCHS for (a) MNIST and TIMIT, and (b) AlexNet.

[20] J. H. Patel *et al.*, "Concurrent error detection in alu's by recomputing with shifted operands," *IEEE Transactions on Computers*, vol. 31, no. 7, pp. 589–595, 1982.

[21] N. Oh *et al.*, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[22] A. Meixner *et al.*, "Argus: Low-cost, comprehensive error detection in simple cores," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 210–222, 2007.

[23] M. Zhang *et al.*, "Sequential element design with built-in soft error resilience," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 12, pp. 1368–1378, 2006.

[24] Y.-C. Chang *et al.*, "On the design and analysis of fault tolerant noc architecture using spare routers," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference (ASPDAC)*, pp. 431–436, 2011.

[25] W.-C. Tsai *et al.*, "A fault-tolerant noc scheme using bidirectional channel," in *Proceedings of the 48th Design Automation Conference (DAC)*, pp. 918–923, ACM, 2011.

[26] H. Kung *et al.*, "Fault-tolerance and two-level pipelining in vlsi systolic arrays," tech. rep., Carnegie-Mellon UNIV, 1983.

[27] J. H. Kim *et al.*, "On the design of fault-tolerant two-dimensional systolic arrays for yield enhancement," *IEEE Transactions on Computers*, vol. 38, no. 4, pp. 515–525, 1989.

[28] M. Esonu *et al.*, "Fault-tolerant design methodology for systolic array architectures," *IEE Proceedings-Computers and Digital Techniques*, vol. 141, no. 1, pp. 17–28, 1994.

[29] H. F. Li *et al.*, "Restructuring for fault-tolerant systolic arrays," *IEEE Transactions on Computers*, vol. 38, no. 2, pp. 307–311, 1989.

[30] G. W. Burr *et al.*, "Neuromorphic computing using non-volatile memory," *Advances in Physics: X*, vol. 2, no. 1, pp. 89–124, 2017.

[31] C. Liu *et al.*, "Rescuing memristor-based neuromorphic design with high defects," in *Proceedings of the 54th Annual Design Automation Conference (DAC)*, pp. 87:1–87:6, ACM, 2017.

[32] G. W. Burr *et al.*, "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element," *IEEE Transactions on Electron Devices*, vol. 62, no. 11, pp. 3498–3507, 2015.

[33] D. Chabi *et al.*, "Robust neural logic block (nlb) based on memristor crossbar array," in *Proceedings of the IEEE/ACM International Symposium on Nanoscale Architectures*, pp. 137–143, 2011.

[34] J. Ba *et al.*, "Do deep nets really need to be deep?," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 2654–2662, 2014.

[35] M. Everingham *et al.*, "The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results." http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

Efficient Image Processing with Deep Neural Networks

Vivienne Sze, Tien-Ju Yang

Massachusetts Institute of Technology

In collaboration with

Yu-Hsin Chen, Joel Emer

Contact Info
email: sze@mit.edu
website: www.rle.mit.edu/eems

Follow @eems_mit

RESEARCH LABORATORY OF ELECTRONICS AT MIT

MTL microsystems technology laboratories massachusetts institute of technology

Contributors

Vivienne Sze

Professor

MIT

Tien-Ju Yang

PhD Candidate

MIT

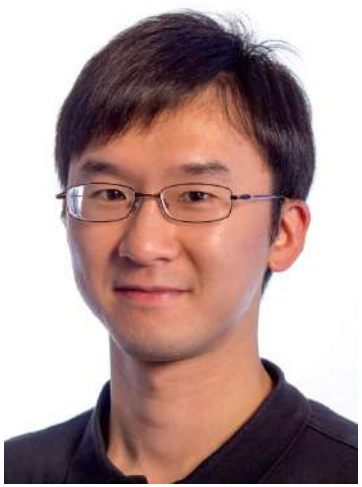
Joel Emer

Senior Distinguished Research Scientist

NVIDIA

Professor

MIT



Yu-Hsin Chen

PhD Graduate

MIT

Outline of Tutorial

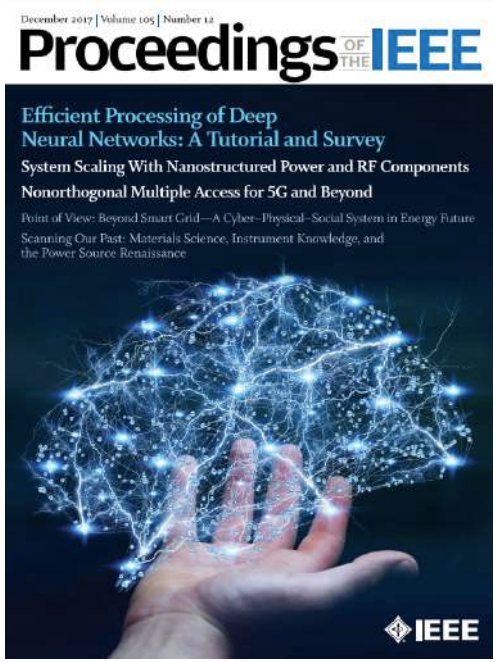
- Brief overview of Deep Neural Networks (DNN)
- **Part 1: Hardware Platforms for DNNs** (e.g., CPU, GPU, FPGA, ASIC) and **metrics for evaluating the efficiency of DNNs**
- **Part 2: Co-design algorithms and hardware** for efficient DNNs (e.g., precision, sparsity, network architecture design, network architecture search, designing networks with hardware in the loop)
- **Part 3: Application of efficient DNNs** on a wide range of image processing and computer vision tasks (e.g., image classification, depth estimation, image segmentation, super-resolution)

Additional Resources

Overview Paper

V. Sze, Y.-H. Chen, T-J. Yang, J. Emer, "*Efficient Processing of Deep Neural Networks: A Tutorial and Survey,*" **Proceedings of the IEEE**, Dec. 2017

Book Coming Soon!



More info about **Tutorial on DNN Architectures**
http://eyeriss.mit.edu/tutorial.html

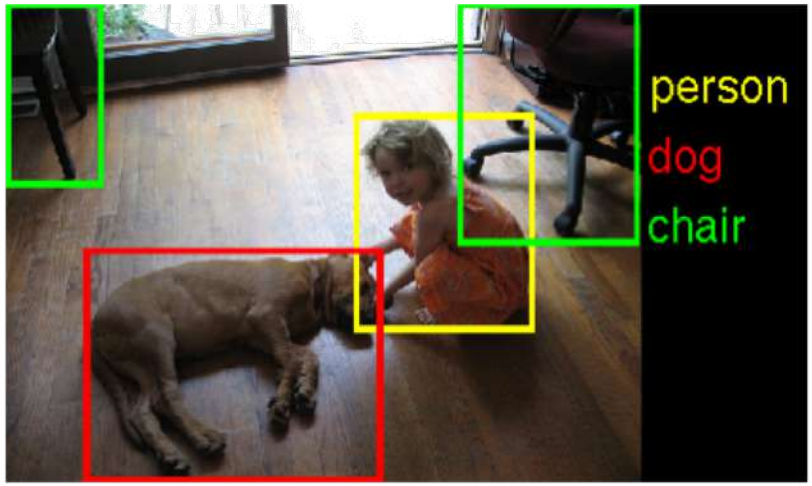
MIT Professional Education Course on
"Designing Efficient Deep Learning Systems"
http://professional-education.mit.edu/deeplearning

For updates Follow @eems_mit

http://mailman.mit.edu/mailman/listinfo/eems-news

Example Applications of Deep Learning

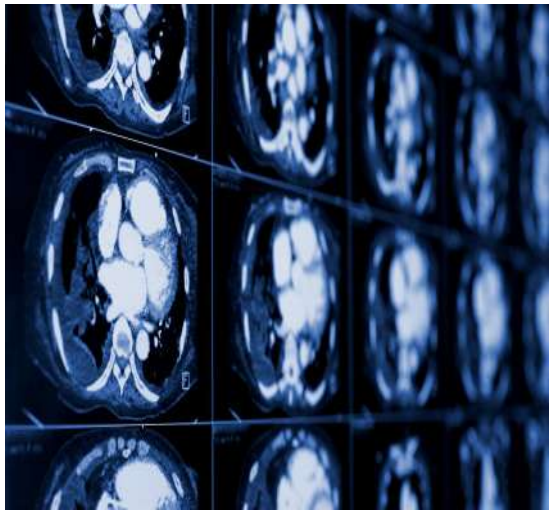
Computer Vision



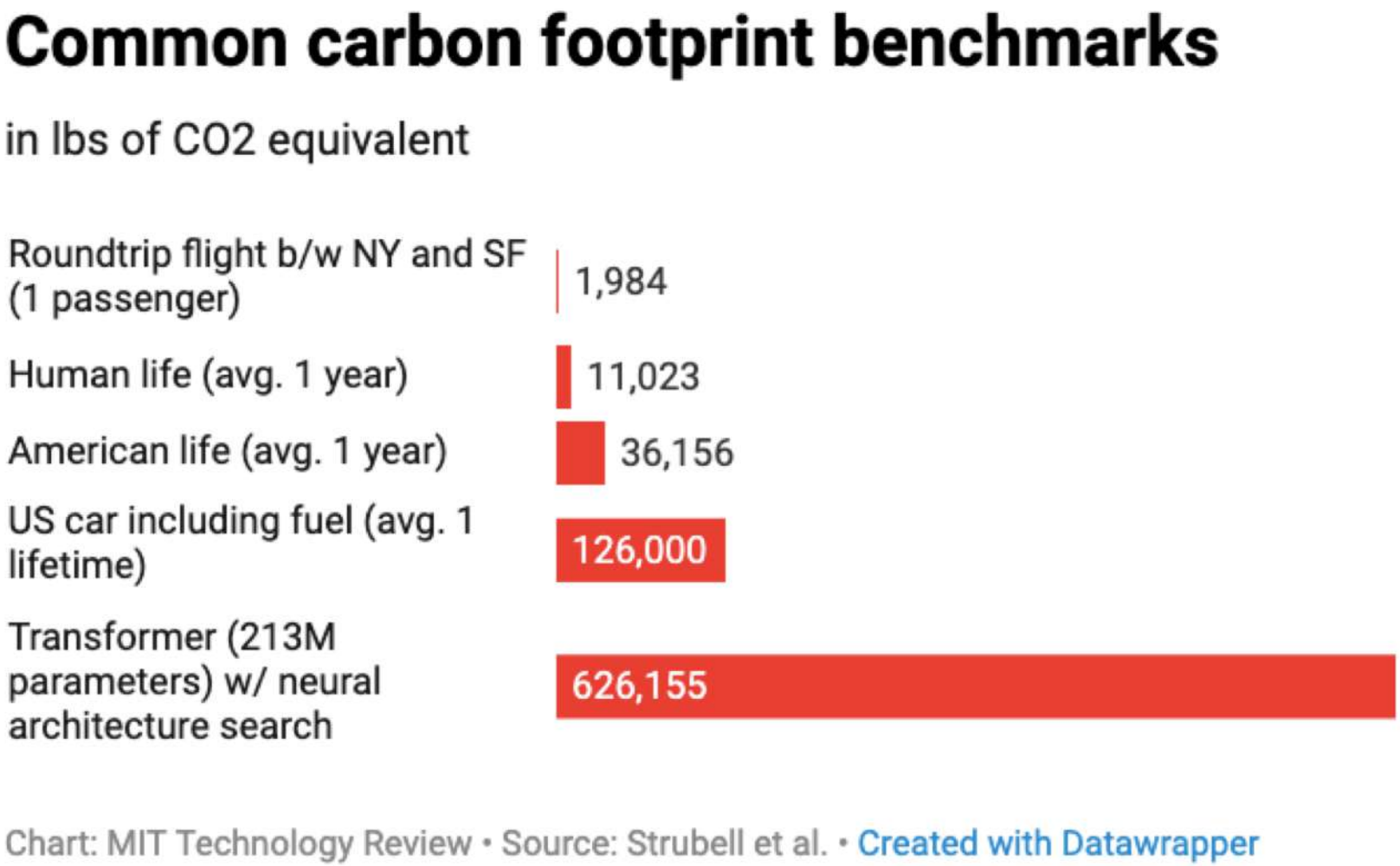
Speech Recognition

Game Play

Medical

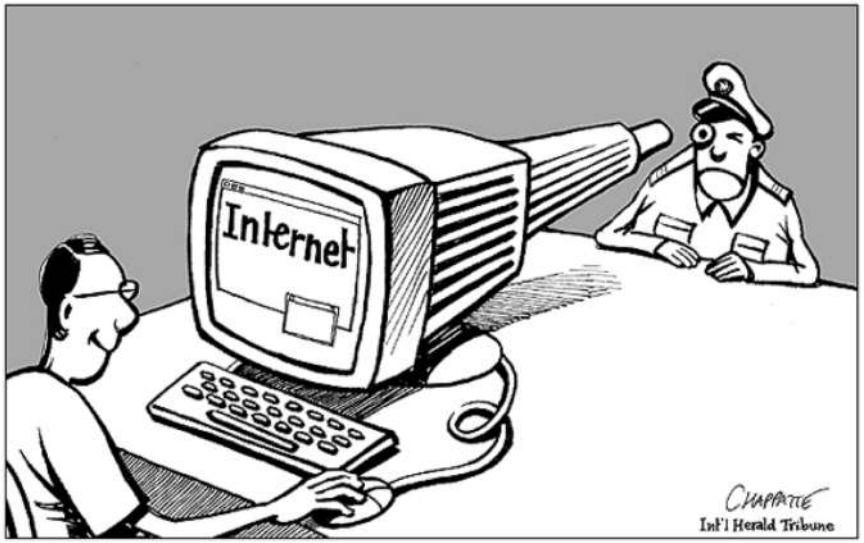


Compute Demands for Deep Learning



Processing at "Edge" instead of the "Cloud"

Privacy



Communication

Image source: www.theregister.co.uk

Latency

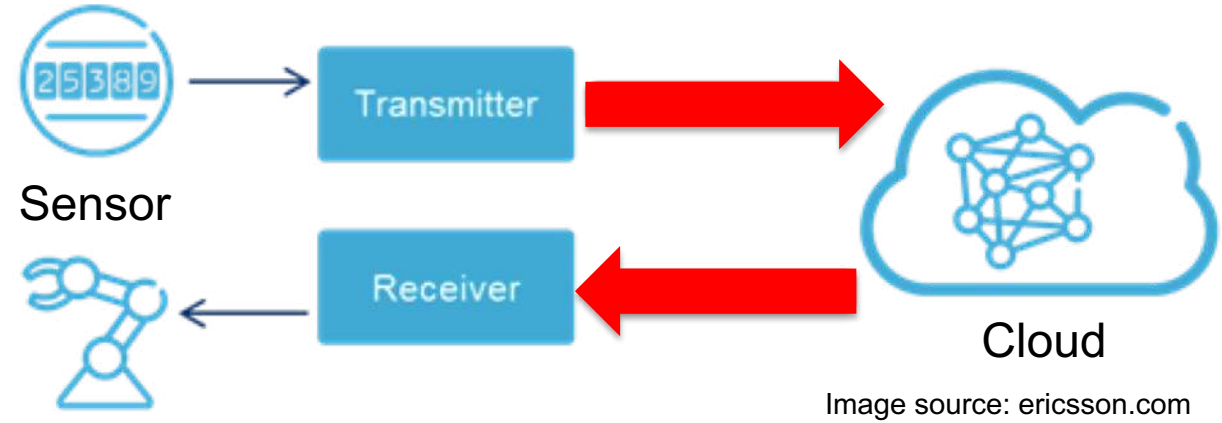


Image source: ericsson.com

Deep Learning for Self-Driving Cars

JACK STEWART TRANSPORTATION 02.06.18 08:00 AM

SELF-DRIVING CARS USE CRAZY AMOUNTS OF POWER, AND IT'S BECOMING A PROBLEM

Shelley, a self-driving Audi TT developed by Stanford University, uses the brains in the trunk to speed around a racetrack autonomously.

NIKKI KAHN/THE WASHINGTON POST/GETTY IMAGES

WIRED

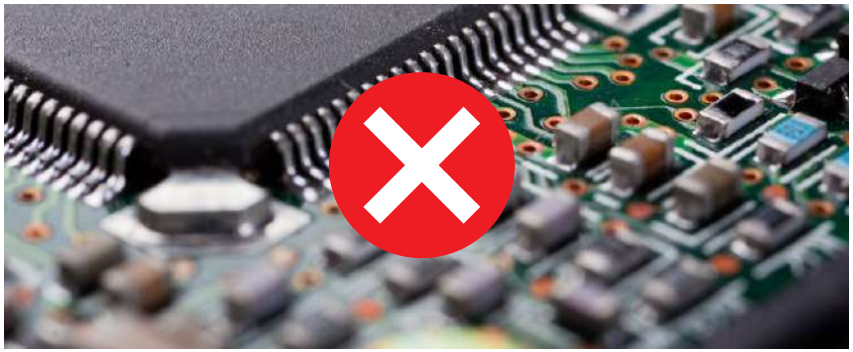
(Feb 2018)

Cameras and radar generate ~6 gigabytes of data every 30 seconds.

Prototypes use around 2,500 Watts. Generates wasted heat and some prototypes need water-cooling!

Existing Processors Consume Too Much Power

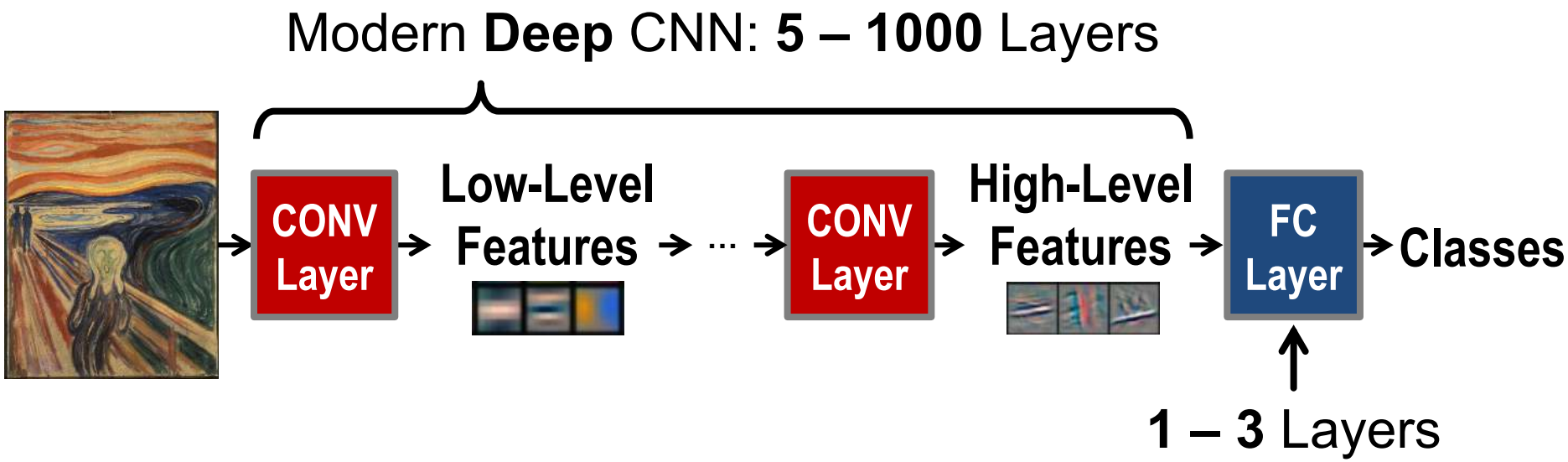
< 1 Watt



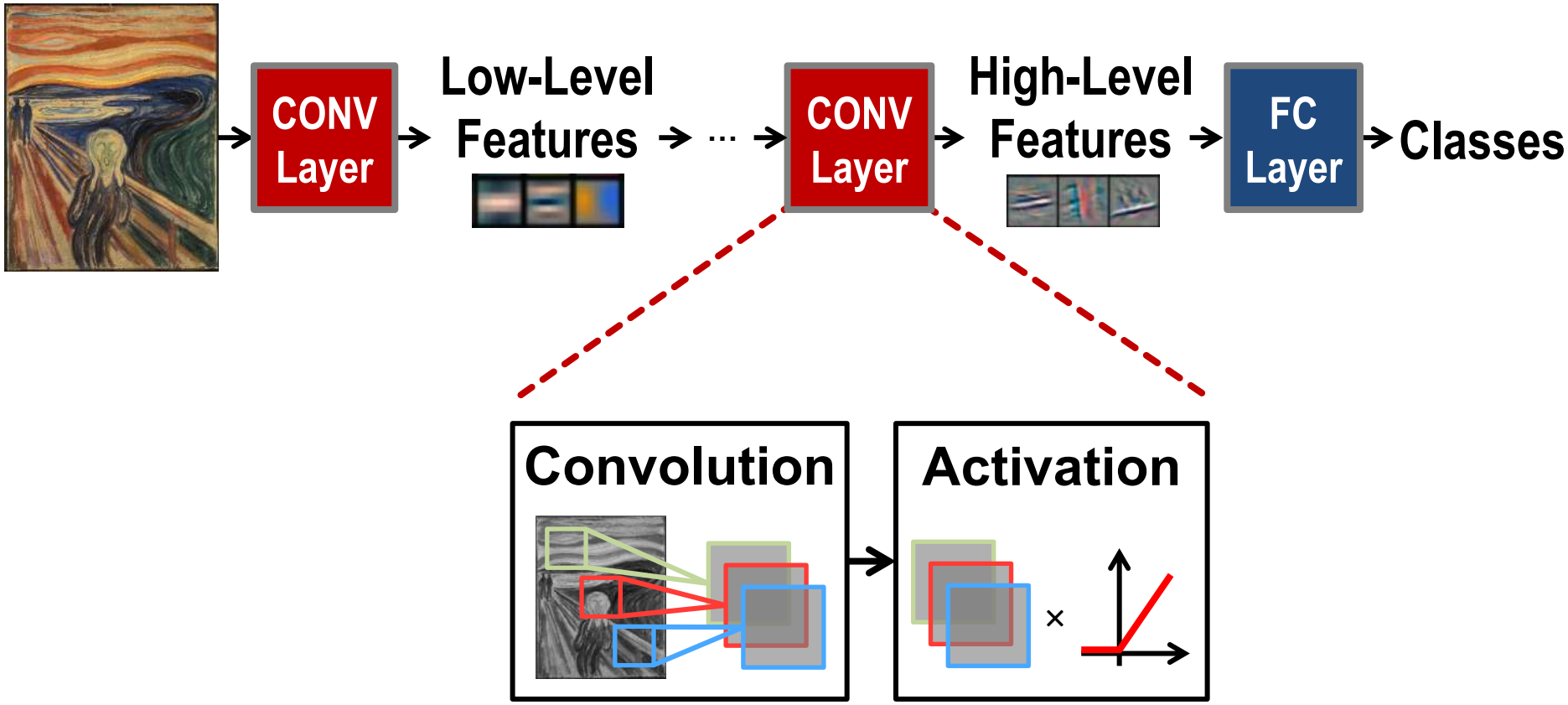
> 10 Watts

Overview of Deep Neural Networks

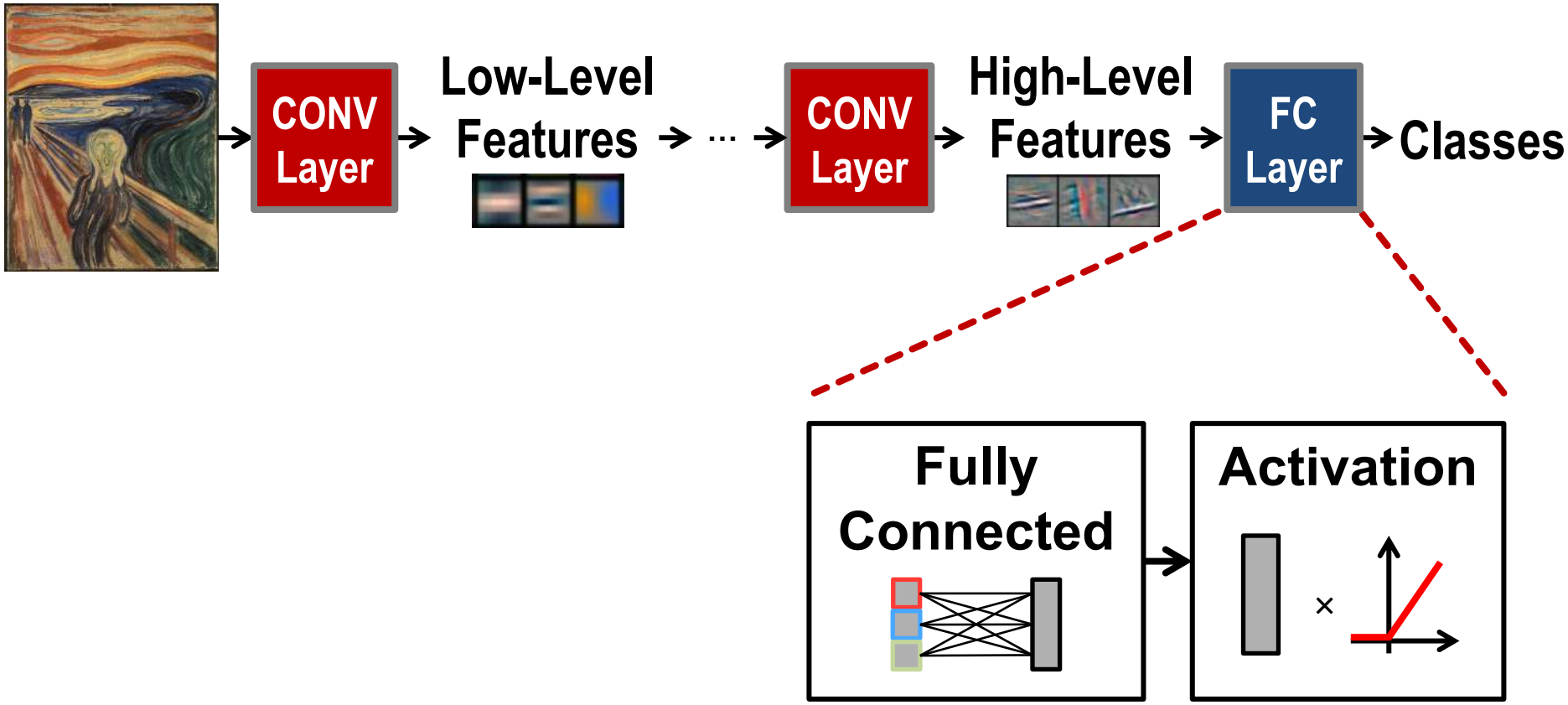
Deep Convolutional Neural Networks



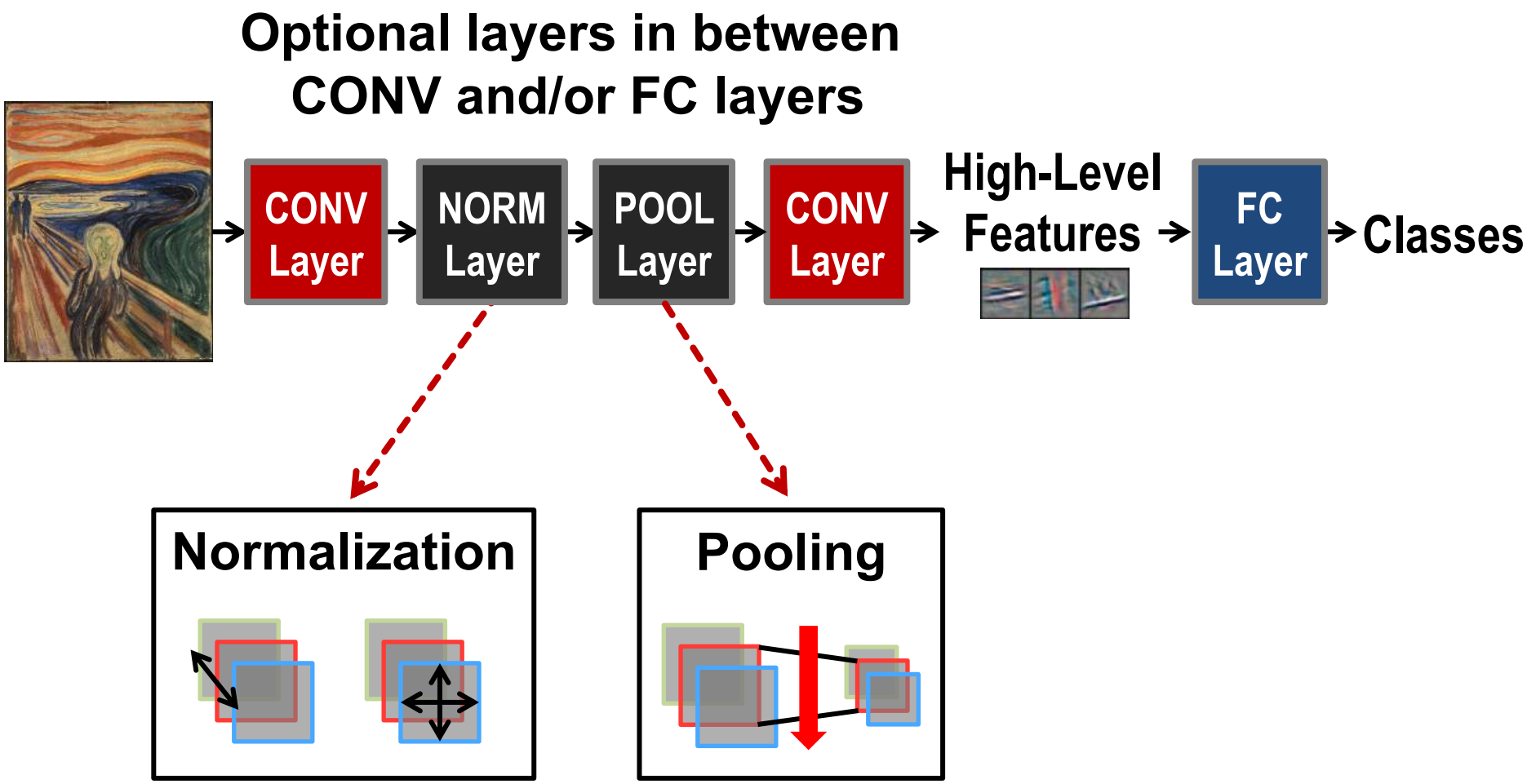
Deep Convolutional Neural Networks



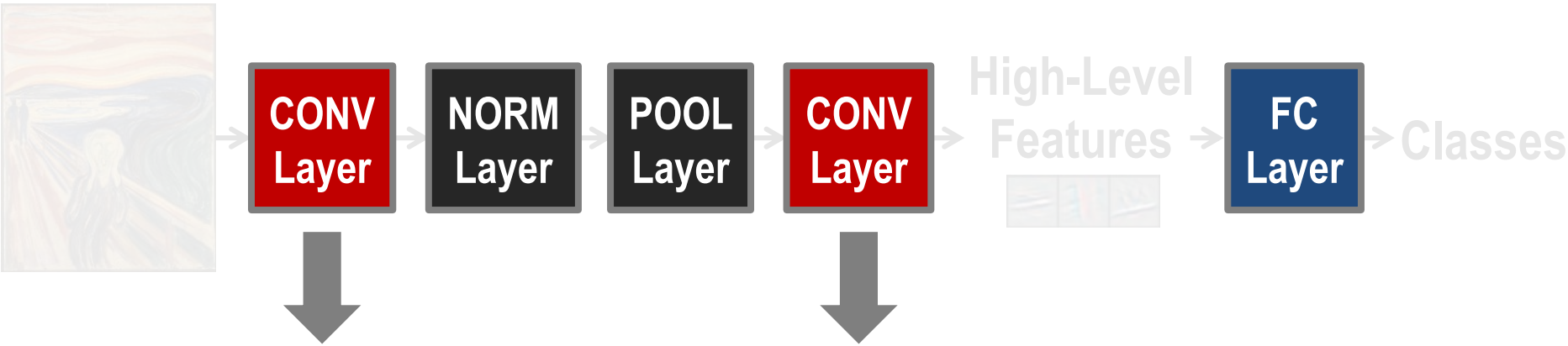
Deep Convolutional Neural Networks

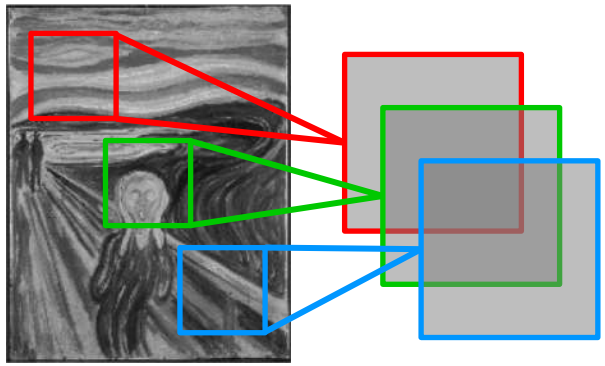


Deep Convolutional Neural Networks



Deep Convolutional Neural Networks



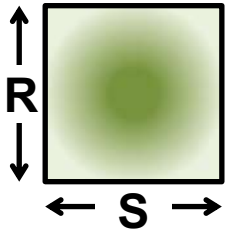


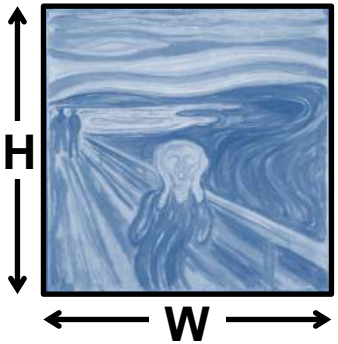
Convolutions account for more than 90% of overall computation, dominating **runtime** and **energy consumption**

Convolution (CONV) Layer

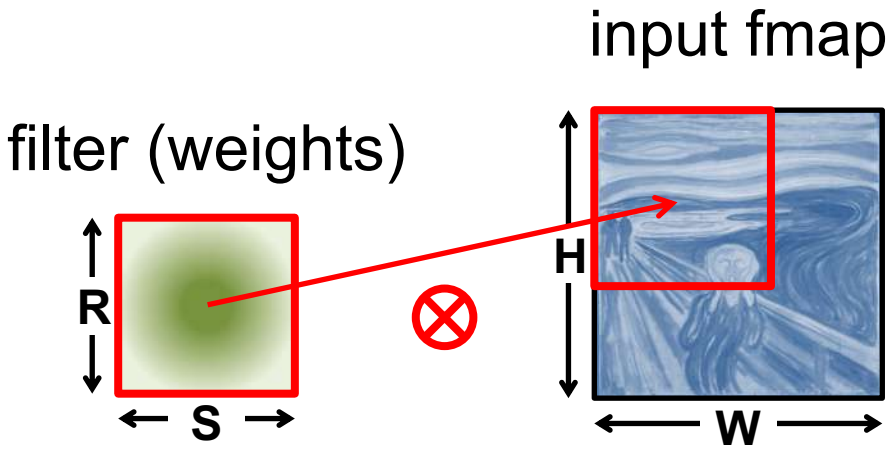
a plane of input activations
a.k.a. **input feature map (fmap)**

filter (weights)



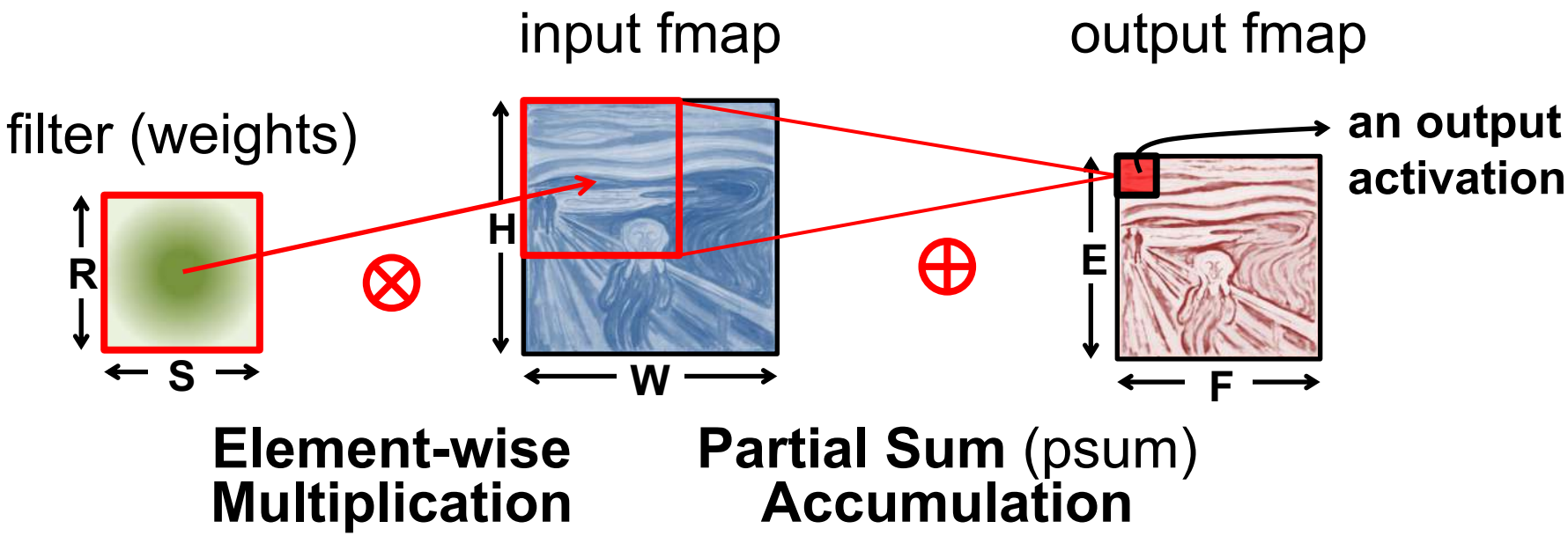


Convolution (CONV) Layer

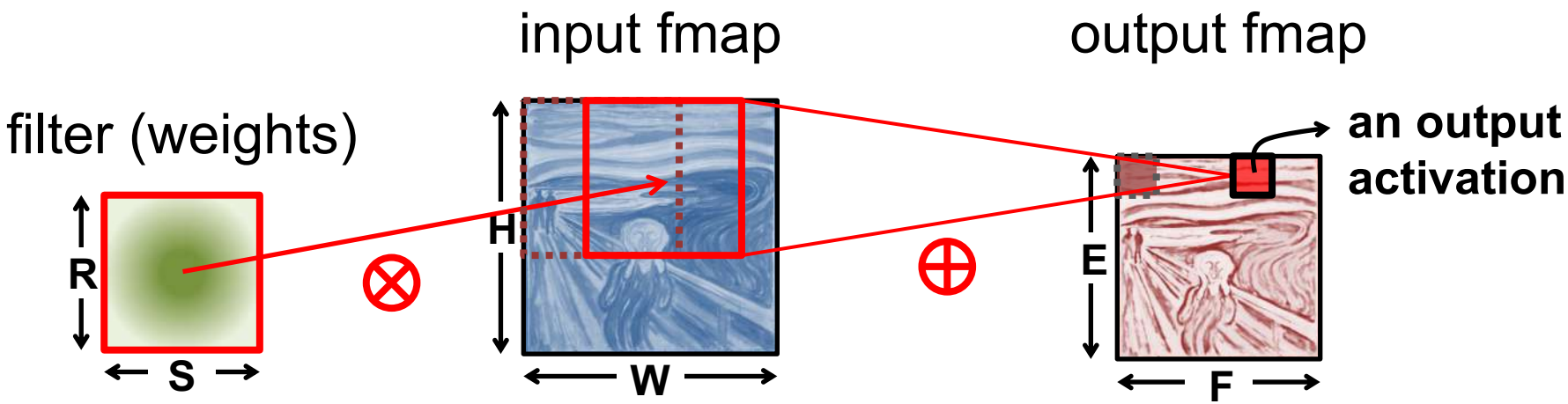


Element-wise Multiplication

Convolution (CONV) Layer

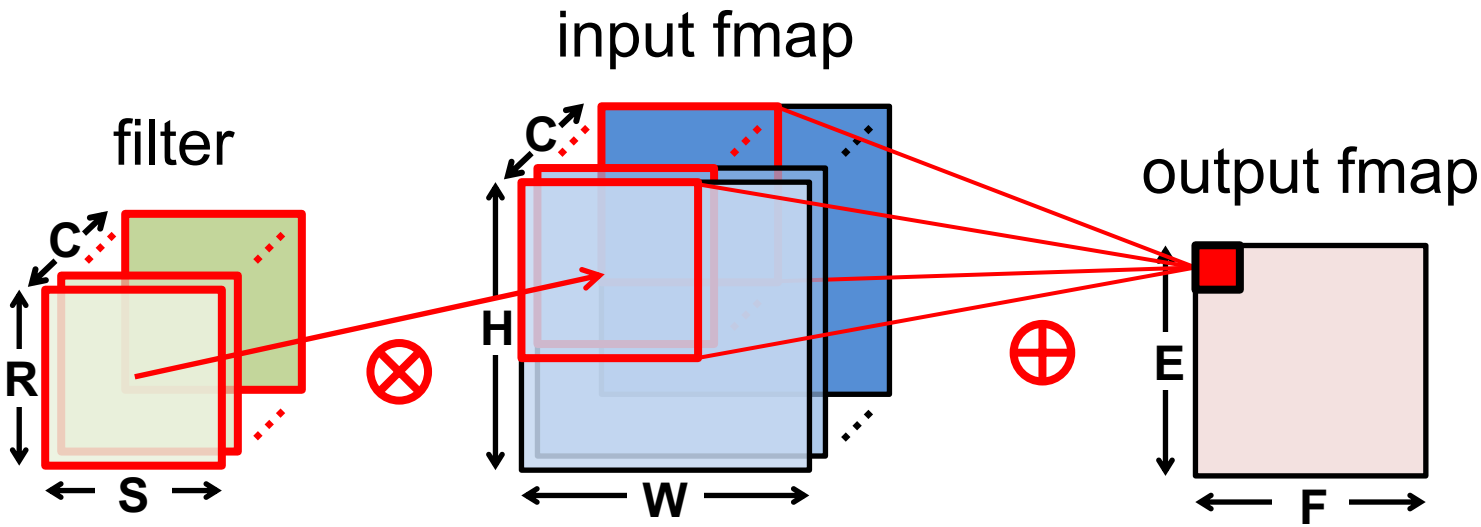


Convolution (CONV) Layer



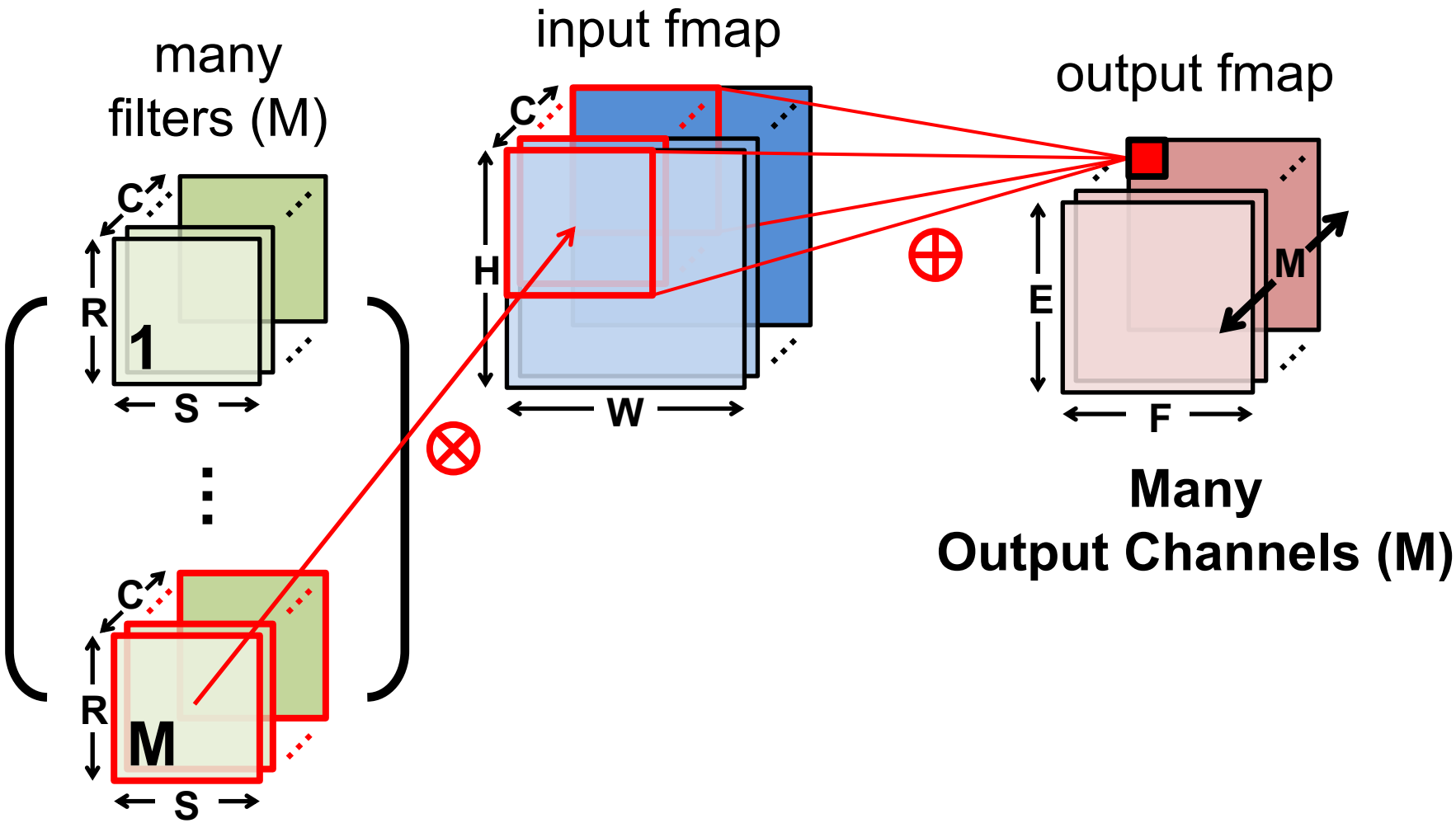
Sliding Window Processing

Convolution (CONV) Layer

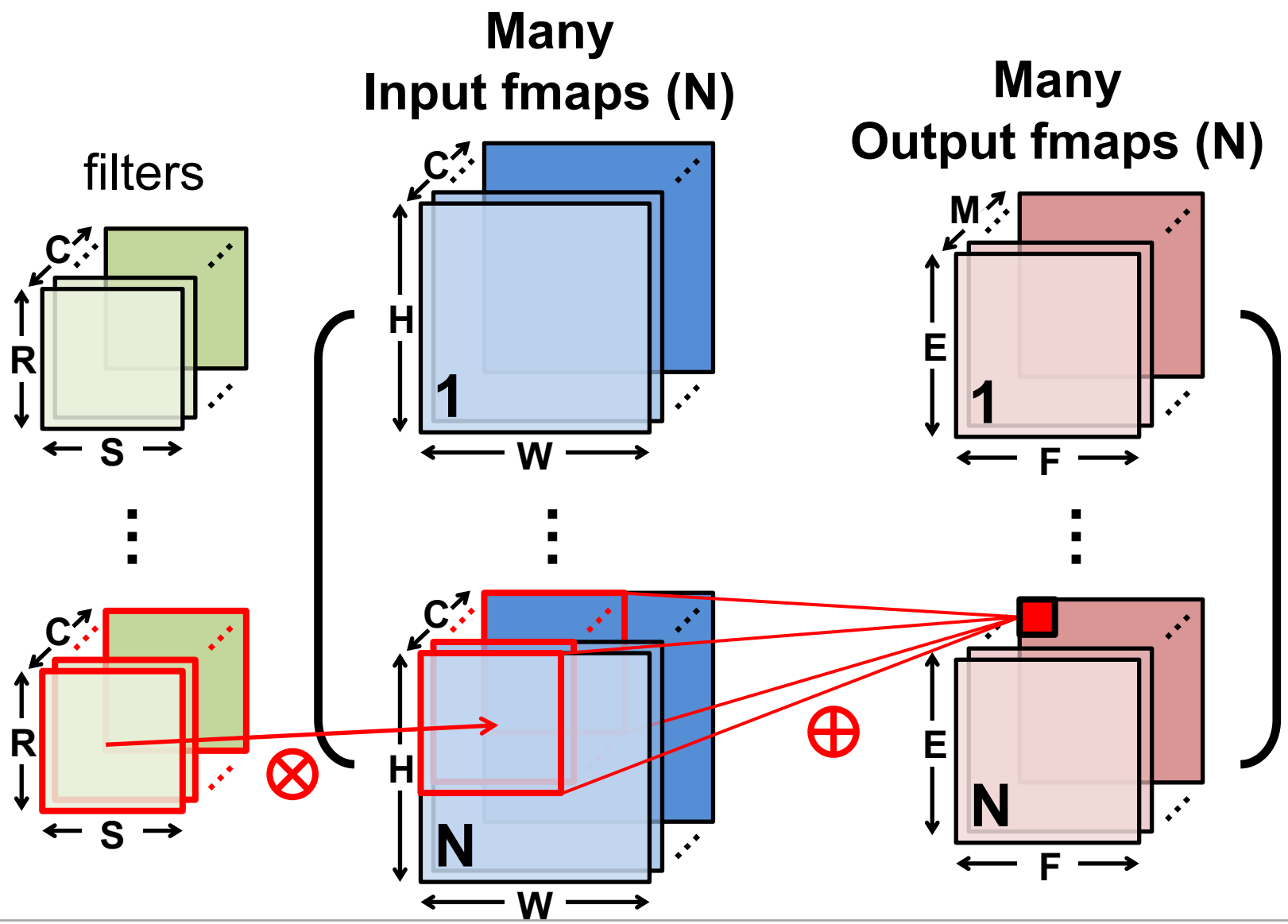


Many Input Channels (C)

Convolution (CONV) Layer



Convolution (CONV) Layer

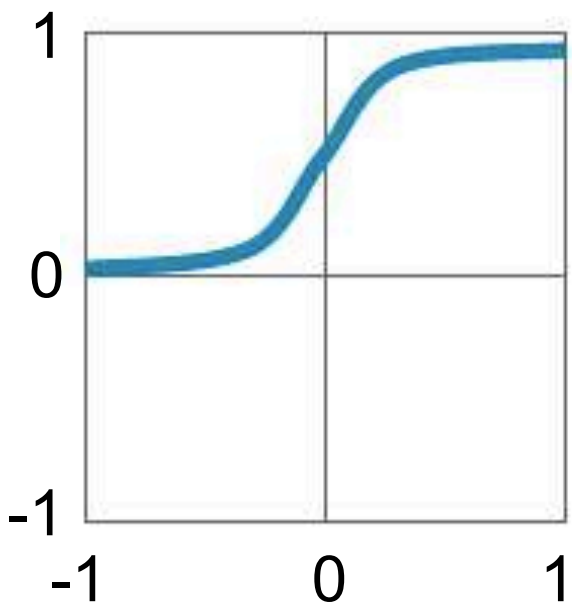


CNN Decoder Ring

- N – Number of input fmaps/output fmaps (batch size)
- C – Number of 2-D input fmaps /filters (channels)
- H – Height of input fmap (activations)
- W – Width of input fmap (activations)
- R – Height of 2-D filter (weights)
- S – Width of 2-D filter (weights)
- M – Number of 2-D output fmaps (channels)
- E – Height of output fmap (activations)
- F – Width of output fmap (activations)

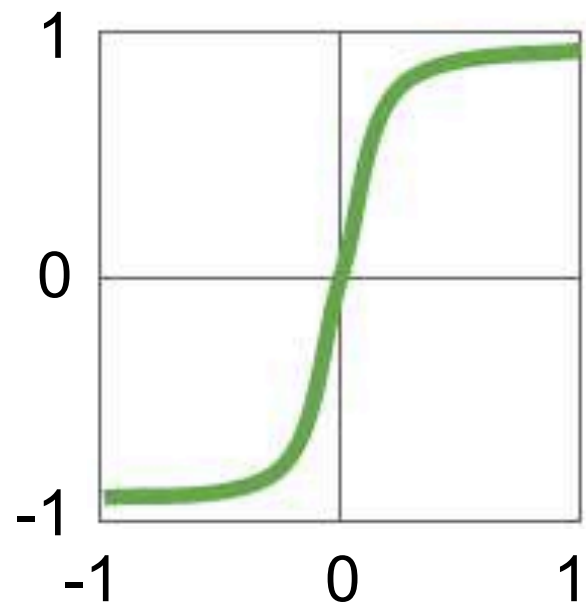
Traditional Activation Functions

Sigmoid



$$y=1/(1+e^{-x})$$

Hyperbolic Tangent

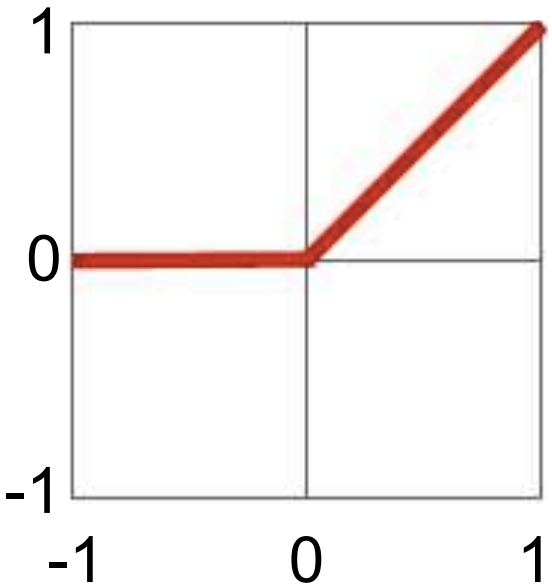


$$y=(e^{x}-e^{-x})/(e^{x}+e^{-x})$$

Image Source: Caffe Tutorial

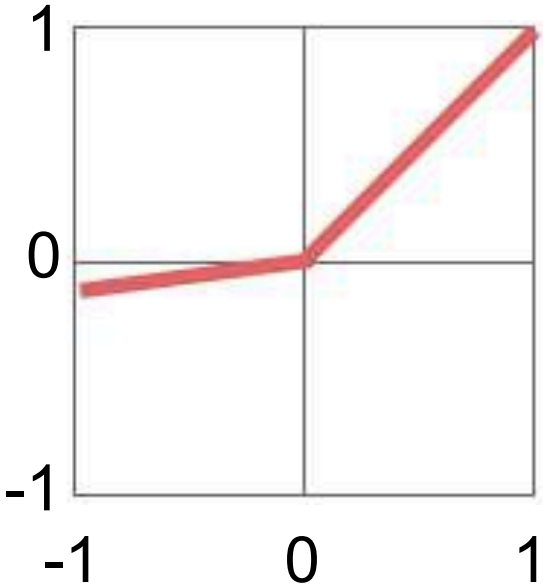
Modern Activation Functions

Rectified Linear Unit (ReLU)



$$y=\max(0,x)$$

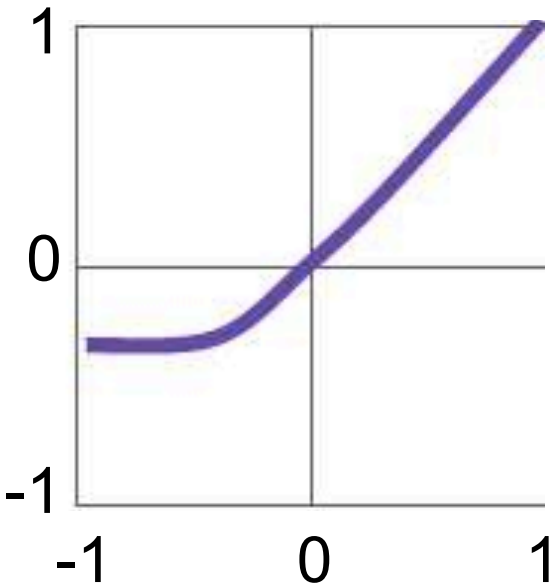
Leaky ReLU



$$y=\max(\alpha x,x)$$

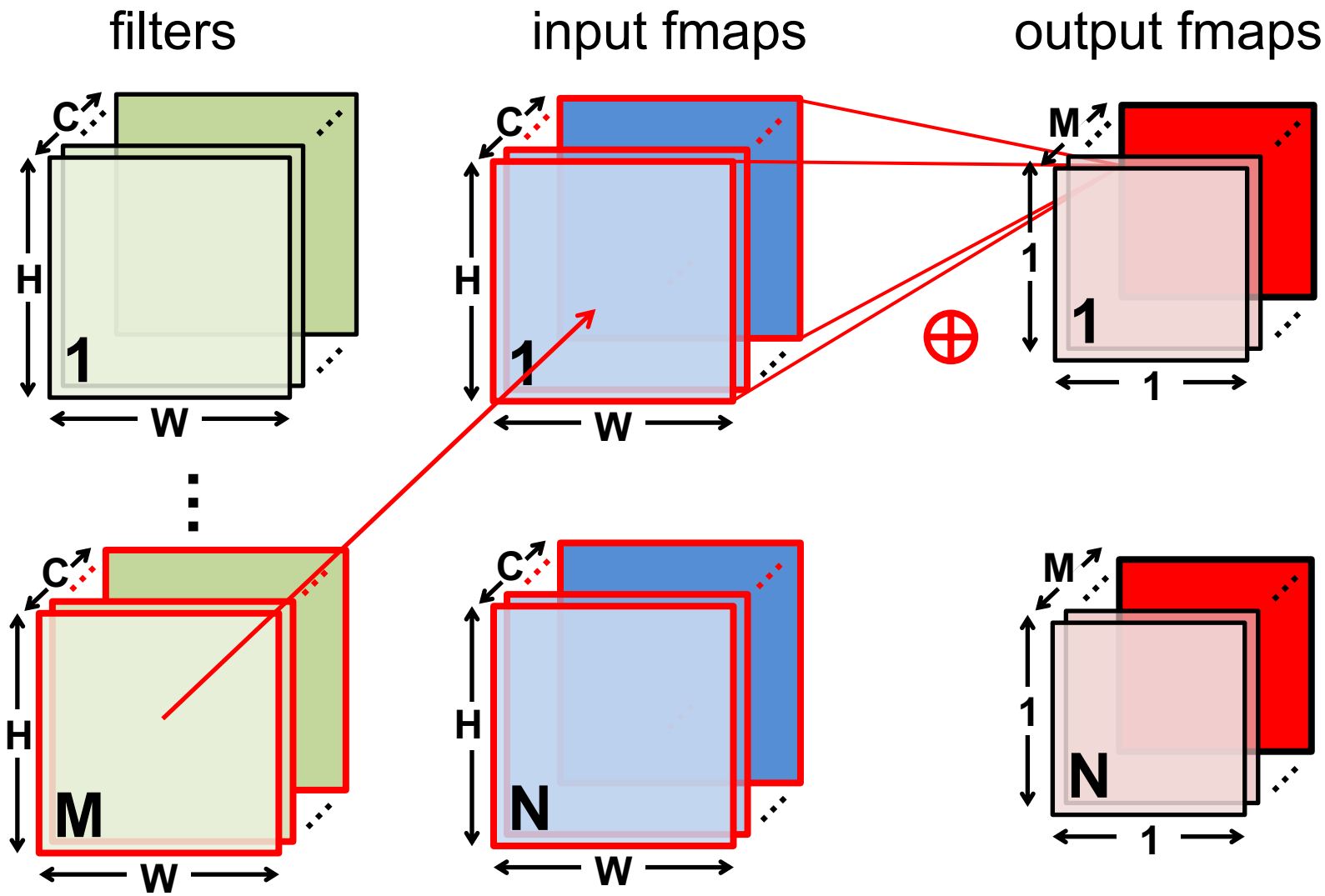
α = small const. (e.g. 0.1)

Exponential LU



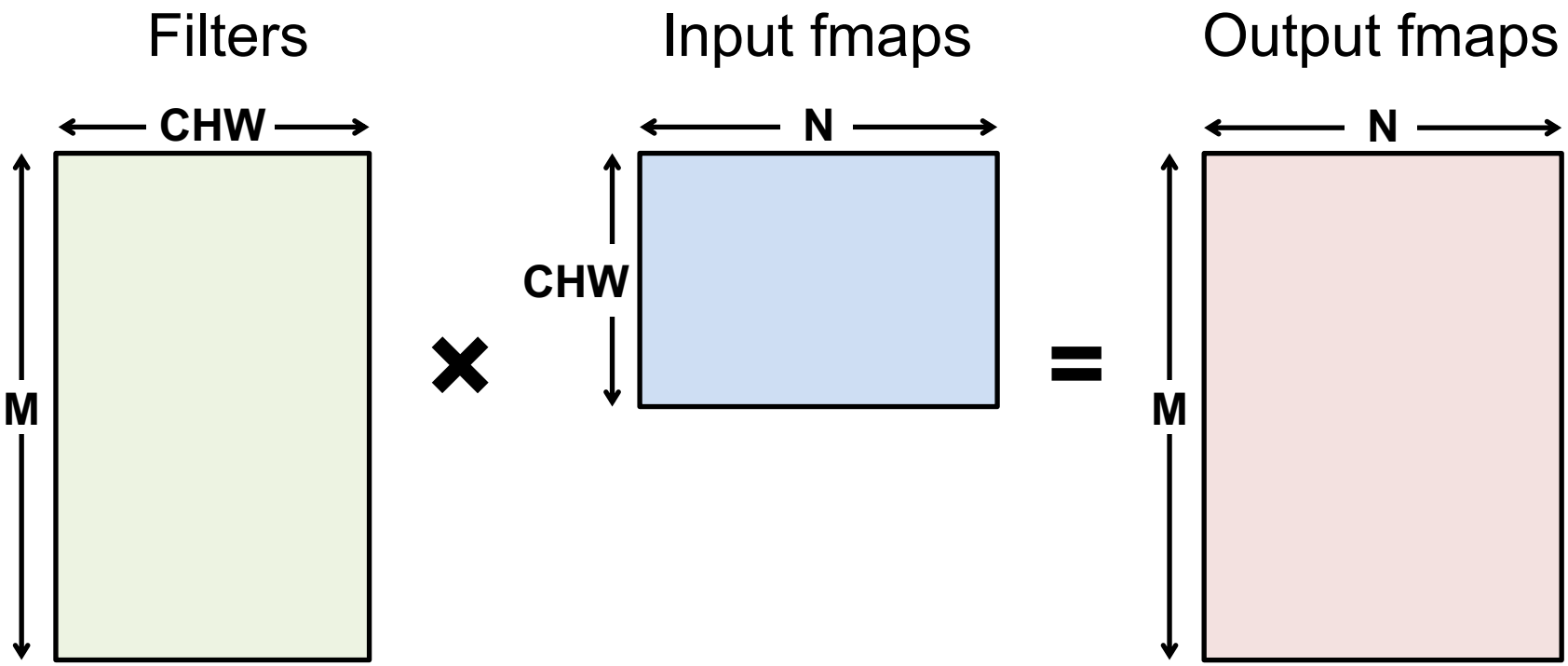
$$y=\begin{cases} x, & x\geq 0 \\ \alpha(e^x-1), & x<0 \end{cases}$$

FC Layer – from CONV Layer POV



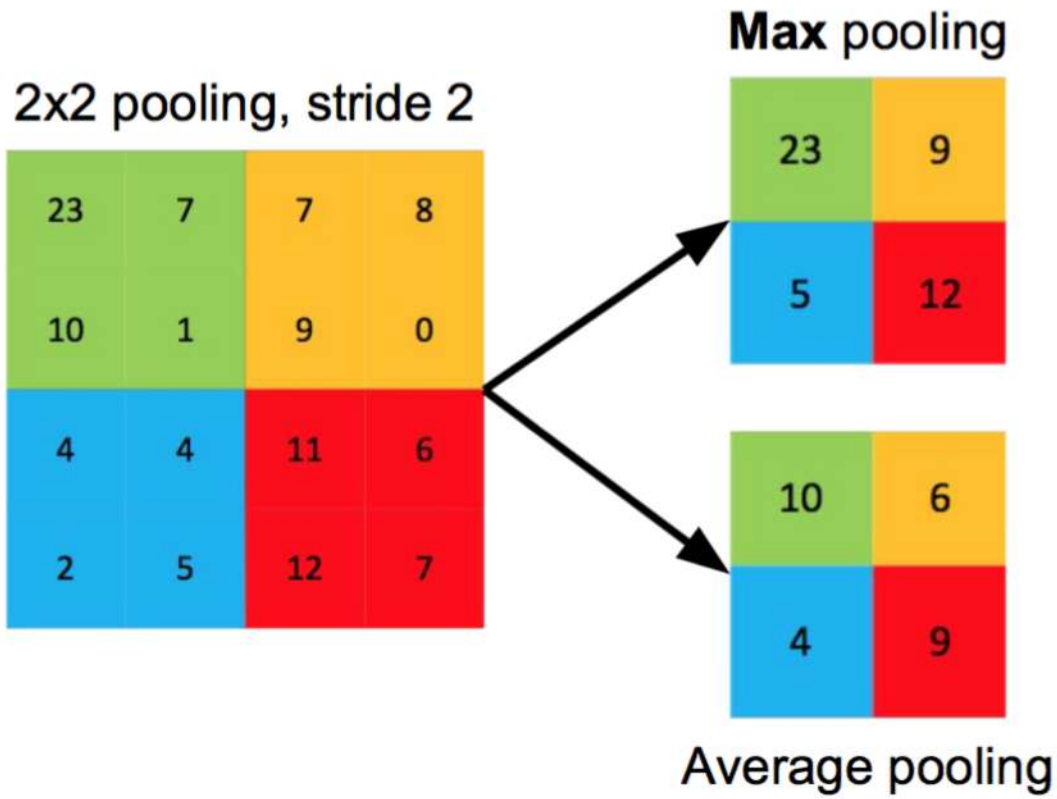
Fully-Connected (FC) Layer

- Height and width of output fmaps are 1 (E = F = 1)
- Filters as large as input fmaps (R = H, S = W)
- Implementation: **Matrix Multiplication**



Pooling (POOL) Layer

- Reduce resolution of each channel independently
- Overlapping or non-overlapping → depending on stride

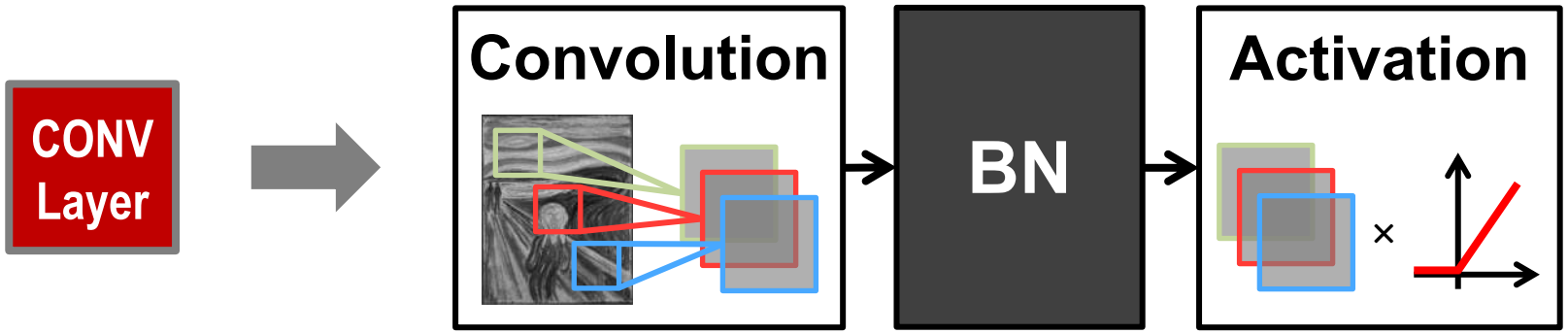


Increases translation-invariance and noise-resilience

Image Source: Caffe Tutorial

Normalization (NORM) Layer

- **Batch Normalization (BN)**
  - Normalize activations towards mean=0 and std. dev.=1 based on the statistics of the training dataset
  - put **in between** **CONV/FC** and **Activation function**



Believed to be key to getting high accuracy and faster training on very deep neural networks.

[Ioffe et al., ICML 2015]

BN Layer Implementation

- The normalized value is further scaled and shifted, the parameters of which are learned from training

data mean

learned scale factor

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \gamma + \beta$$

learned shift factor

data std. dev.

small const. to avoid numerical problems

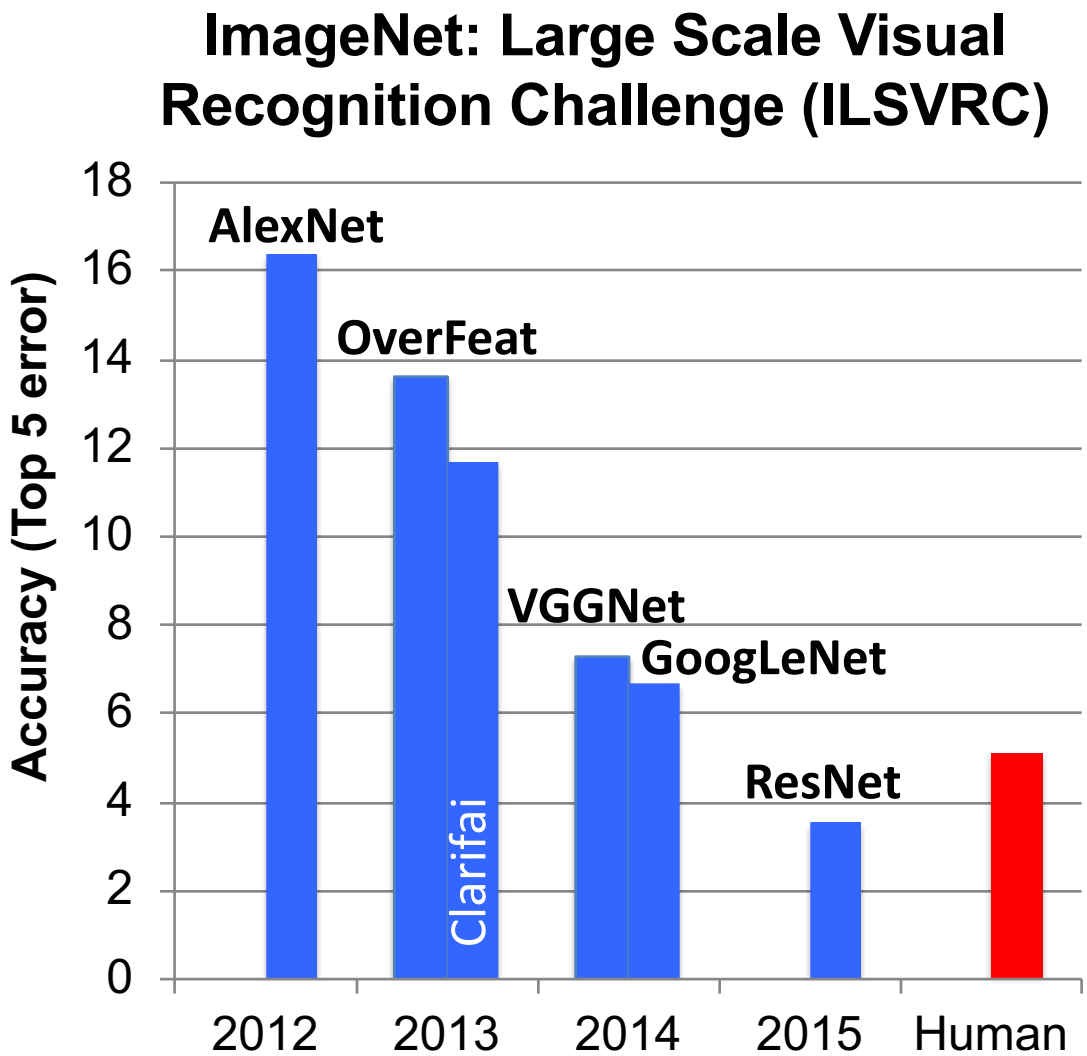
Relevant Components for this Tutorial

- Typical operations that we will discuss:
  - Convolution (CONV)
  - Fully-Connected (FC)
  - Max Pooling
  - ReLU

Popular DNN Models

Popular DNNs

- LeNet (1998)
- AlexNet (2012)
- OverFeat (2013)
- VGGNet (2014)
- GoogleNet (2014)
- ResNet (2015)



[O. Russakovsky et al., IJCV 2015]

ImageNet

Image Classification
~256x256 pixels (color)
1000 Classes
1.3M Training
100,000 Testing (50,000 Validation)

For **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)**
accuracy of classification task reported based on top-1 and top-5 error

Image Source: http://karpathy.github.io/

http://www.image-net.org/challenges/LSVRC/

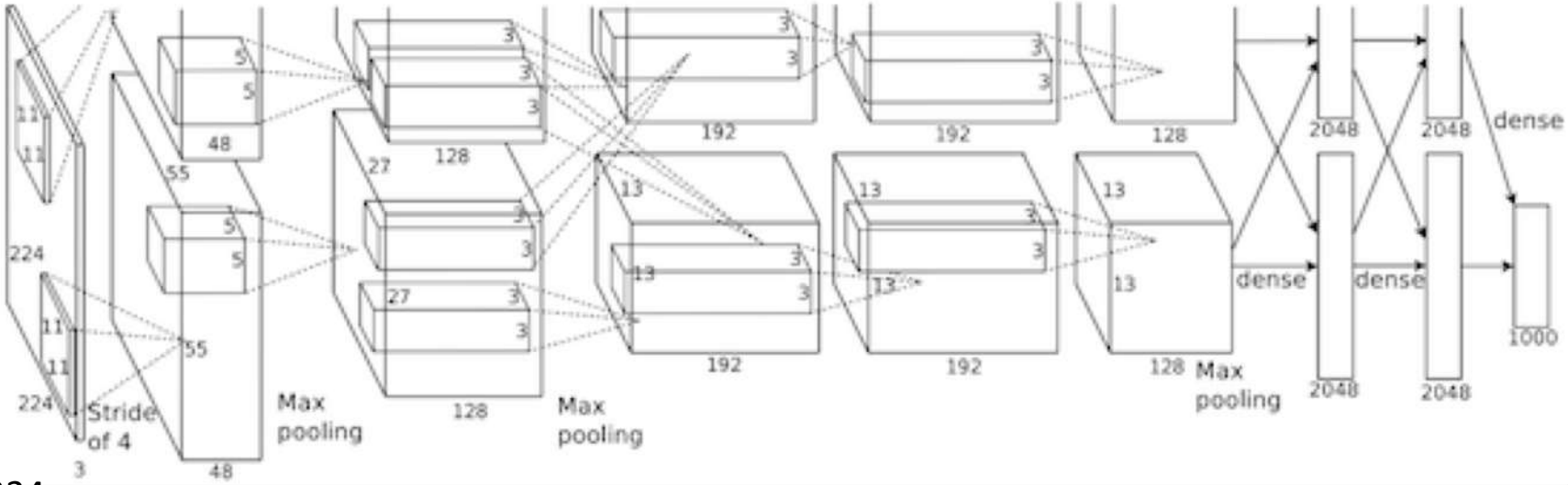
AlexNet

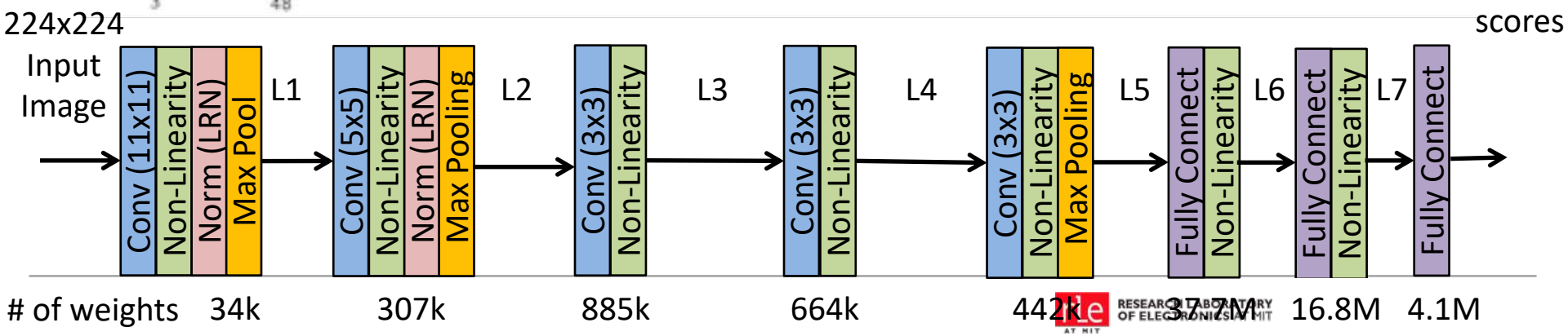
CONV Layers: 5
Fully Connected Layers: 3
Weights: 61M
MACs: 724M
ReLU used for non-linearity

ILSCVR12 Winner

Uses Local Response Normalization (LRN)

[Krizhevsky et al., NeurIPS 2012]



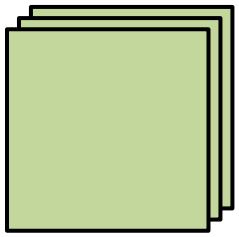


Large Sizes with Varying Shapes

AlexNet Convolutional Layer Configurations

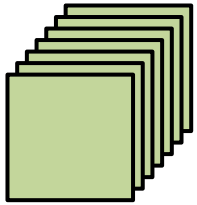
Layer	Filter Size (RxS)	# Filters (M)	# Channels (C)	Stride
1	11x11	96	3	4
2	5x5	256	48	1
3	3x3	384	256	1
4	3x3	384	192	1
5	3x3	256	192	1

Layer 1



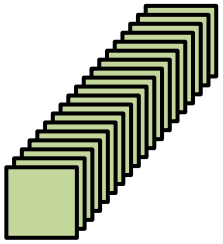
34k Params
105M MACs

Layer 2



307k Params
224M MACs

Layer 3



885k Params
150M MACs

[Krizhevsky et al., NeurIPS 2012]

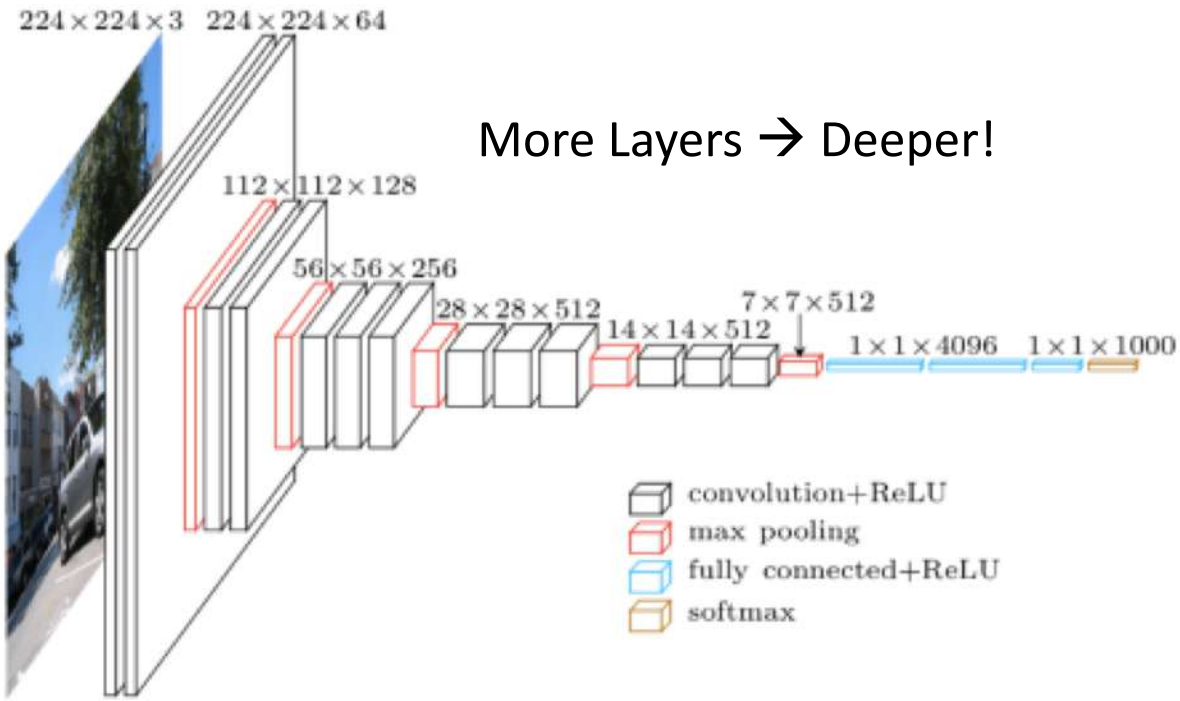
VGG-16

CONV Layers: 13
Fully Connected Layers: 3
Weights: 138M
MACs: 15.5G

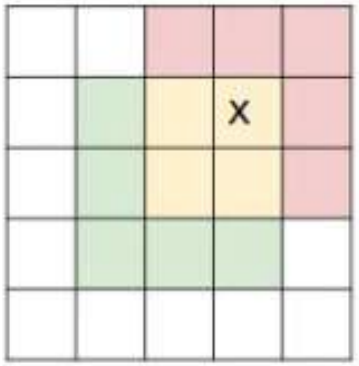
Also, 19 layer version

Reduce # of weights

More Layers → Deeper!



stack 2
3x3 conv



for a 5x5
receptive field

[figure credit A. Karpathy]

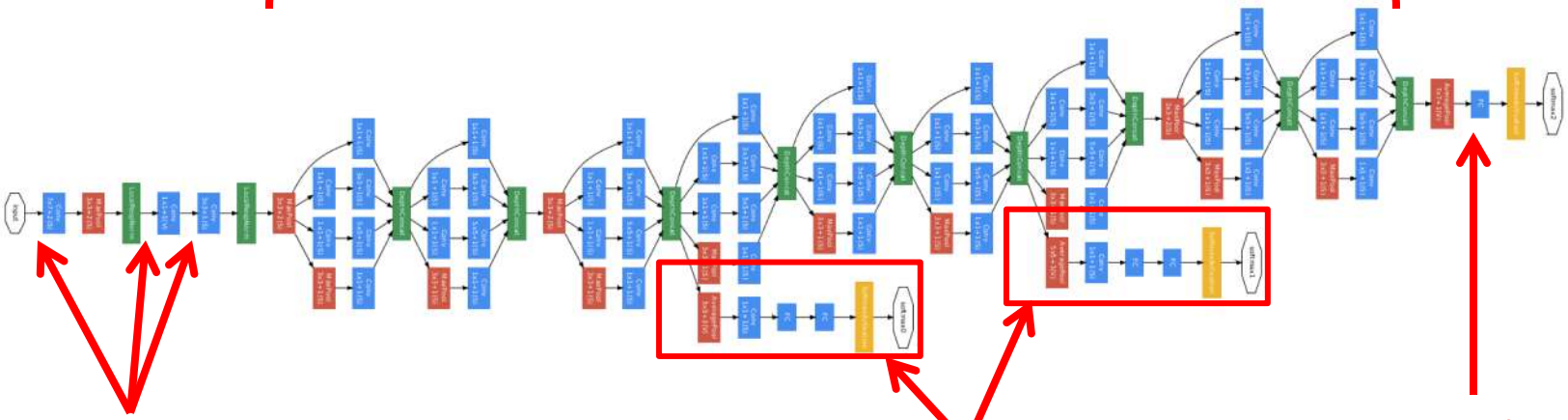
Image Source: http://www.cs.toronto.edu/~frossard/post/vgg16/

GoogLeNet/Inception (v1)

CONV Layers: 21 (depth), 57 (total)
Fully Connected Layers: 1
Weights: 7.0M
MACs: 1.43G

Also, v2, v3 and v4
ILSVRC14 Winner

9 Inception Layers



3 CONV layers

Auxiliary Classifiers
(helps with training,
not used during inference)

1 FC layer
(reduced from 3)

[Szegedy et al., arXiv 2014, CVPR 2015]

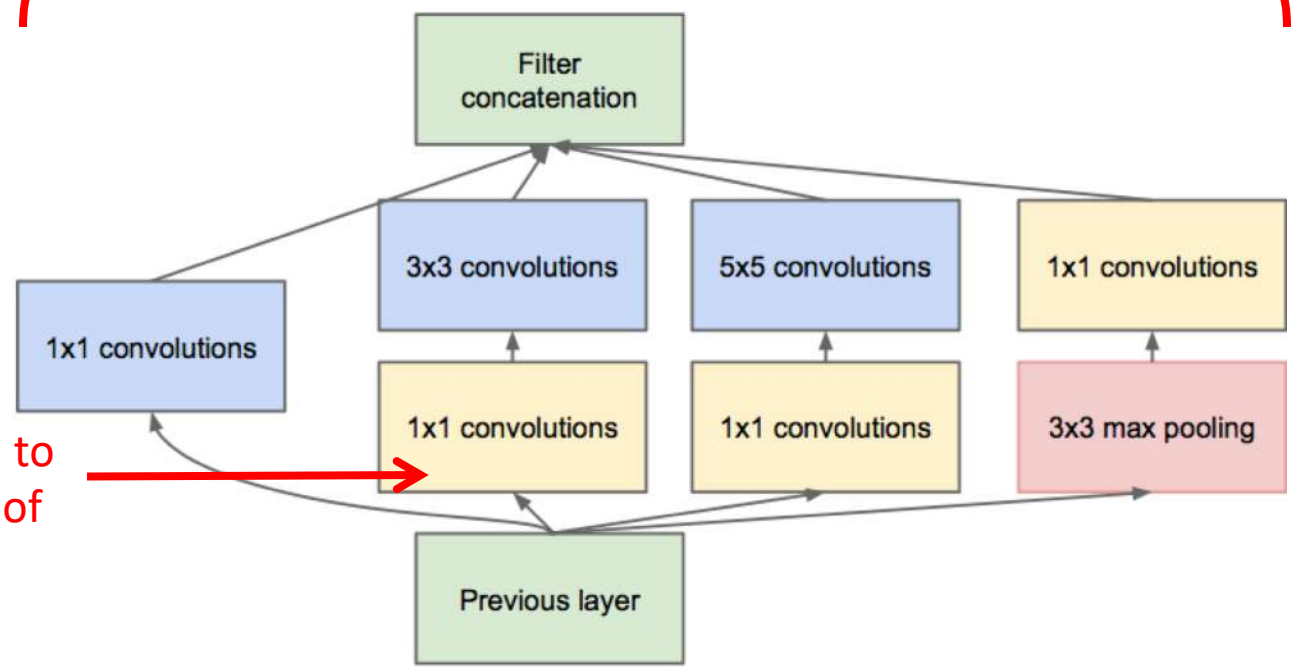
GoogLeNet/Inception (v1)

CONV Layers: 21 (depth), 57 (total)
Fully Connected Layers: 1
Weights: 7.0M
MACs: 1.43G

Also, v2, v3 and v4
ILSVRC14 Winner

parallel filters of different size have the effect of processing image at different scales

Inception Module



1x1 'bottleneck' to reduce number of weights and multiplications

[Szegedy et al., arXiv 2014, CVPR 2015]

ResNet

ILSVRC15 Winner
(better than human level accuracy!)

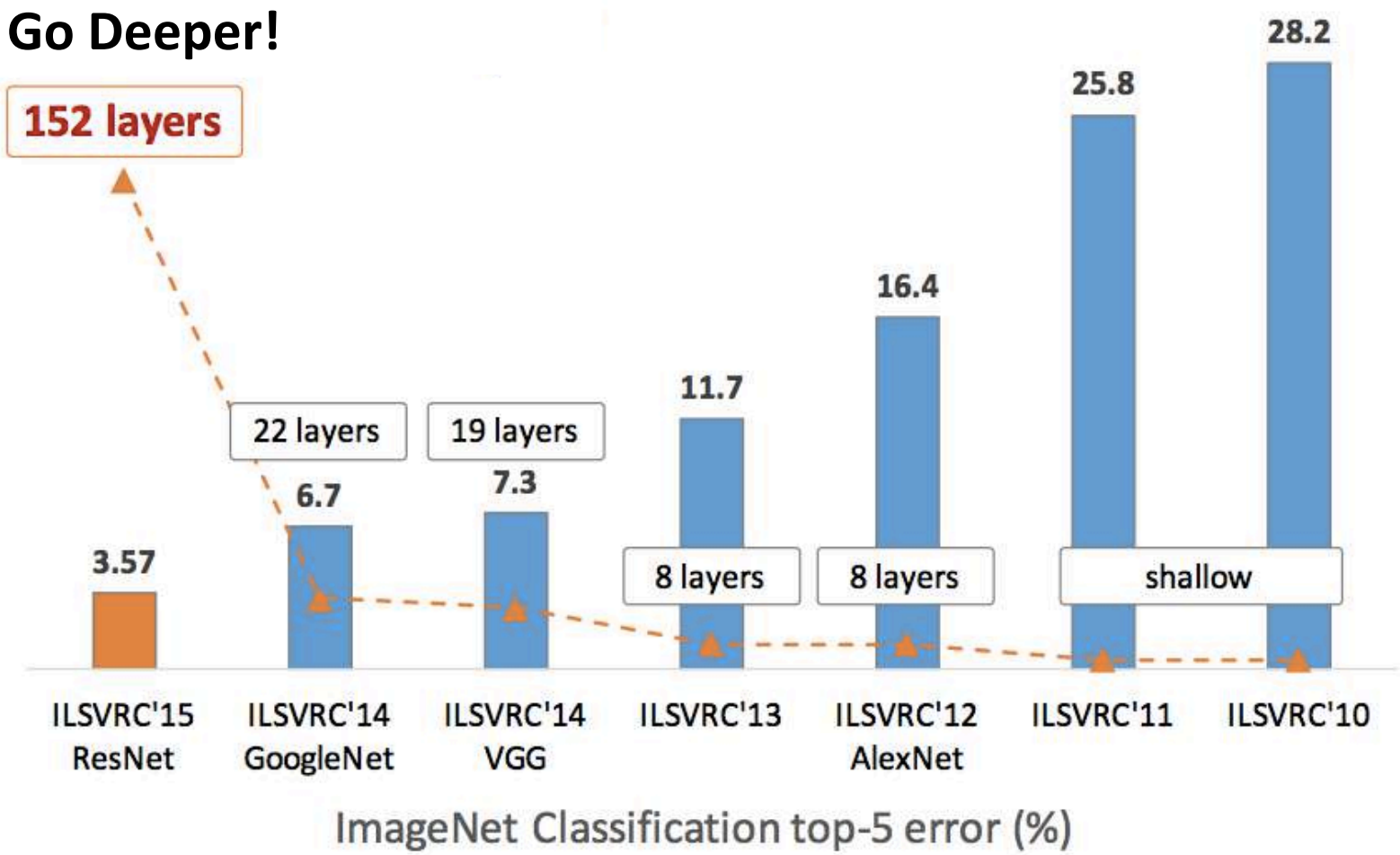


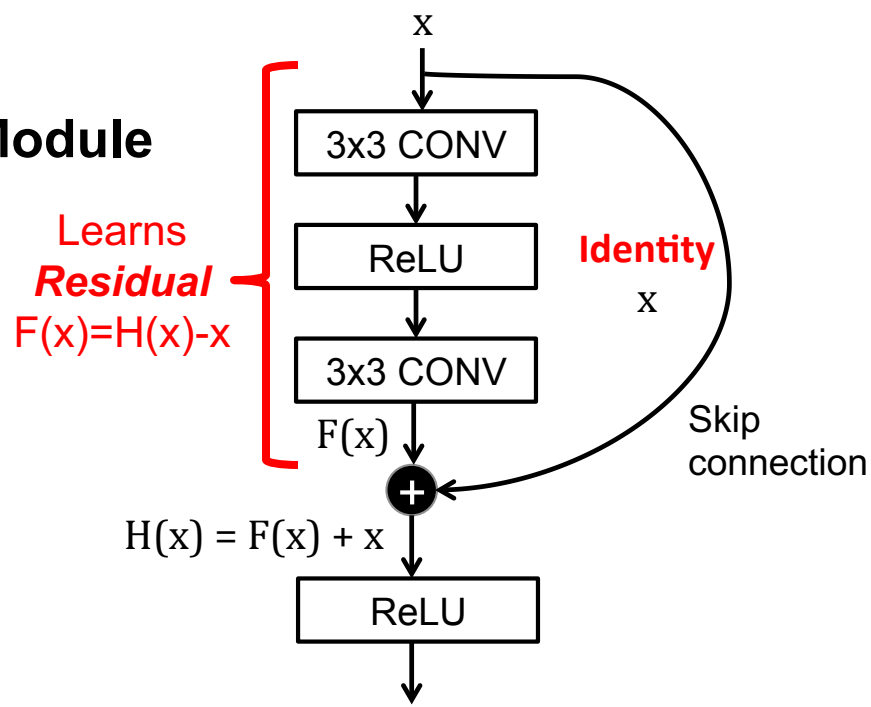
Image Source: http://icml.cc/2016/tutorials/icml2016_tutorial_deep_residual_networks_kaiminghe.pdf

ResNet-50

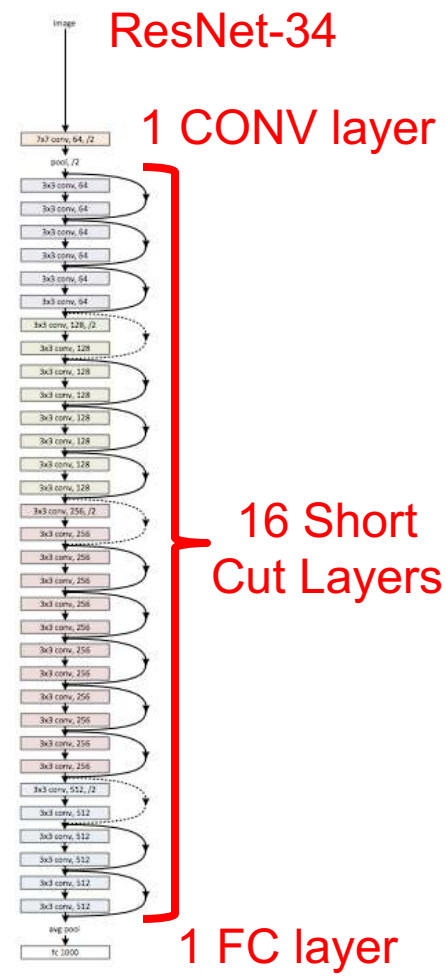
CONV Layers: 49
Fully Connected Layers: 1
Weights: 25.5M
MACs: 3.9G

Also, 34,**152** and 1202 layer versions
ILSVRC15 Winner

Short Cut Module



Helps address the vanishing gradient challenge for training very deep networks



Summary of Popular CNNs

Metrics	LeNet-5	AlexNet	VGG-16	GoogLeNet (v1)	ResNet-50
Top-5 error	n/a	16.4	7.4	6.7	5.3
Input Size	28x28	227x227	224x224	224x224	224x224
# of CONV Layers	**2**	**5**	**16**	**21 (depth)**	**49**
Filter Sizes	5	3, 5,11	3	1, 3 , 5, 7	1, 3, 7
# of Channels	1, 6	3 - 256	3 - 512	3 - 1024	3 - 2048
# of Filters	6, 16	96 - 384	64 - 512	64 - 384	64 - 2048
Stride	1	1, 4	1	1, 2	1, 2
# of Weights	2.6k	2.3M	14.7M	6.0M	23.5M
# of MACs	283k	666M	15.3G	1.43G	3.86G
# of FC layers	**2**	**3**	**3**	**1**	**1**
# of Weights	58k	58.6M	124M	1M	2M
# of MACs	58k	58.6M	124M	1M	2M
Total Weights	**60k**	**61M**	**138M**	**7M**	**25.5M**
Total MACs	**341k**	**724M**	**15.5G**	**1.43G**	**3.9G**

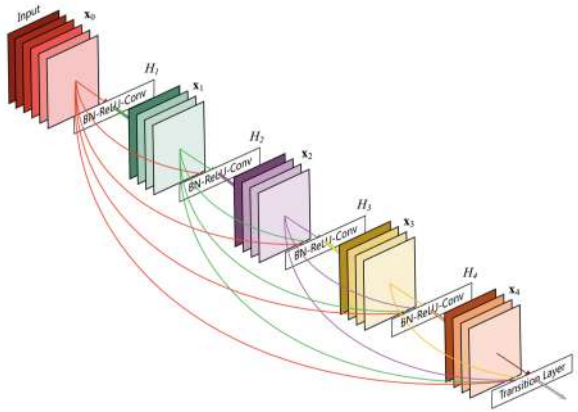
CONV Layers increasingly important!

Summary of Popular CNNs

- **AlexNet**
  - First CNN Winner of ILSVRC
  - Uses LRN (deprecated after this)
- **VGG-16**
  - Goes Deeper (16+ layers)
  - Uses only 3x3 filters (stack for larger filters)
- **GoogLeNet (v1)**
  - Reduces weights with Inception and only one FC layer
  - Inception: 1x1 and DAG (parallel connections)
  - Batch Normalization
- **ResNet**
  - Goes Deeper (24+ layers)
  - Shortcut connections

Beyond ResNet

DenseNet



[Huang et al., CVPR 2017]

Wide ResNet

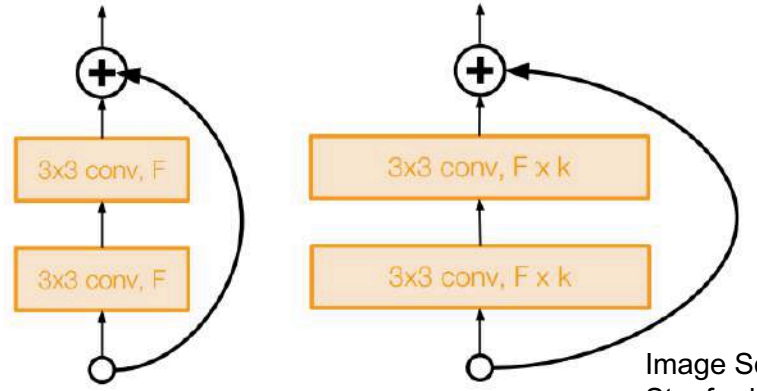
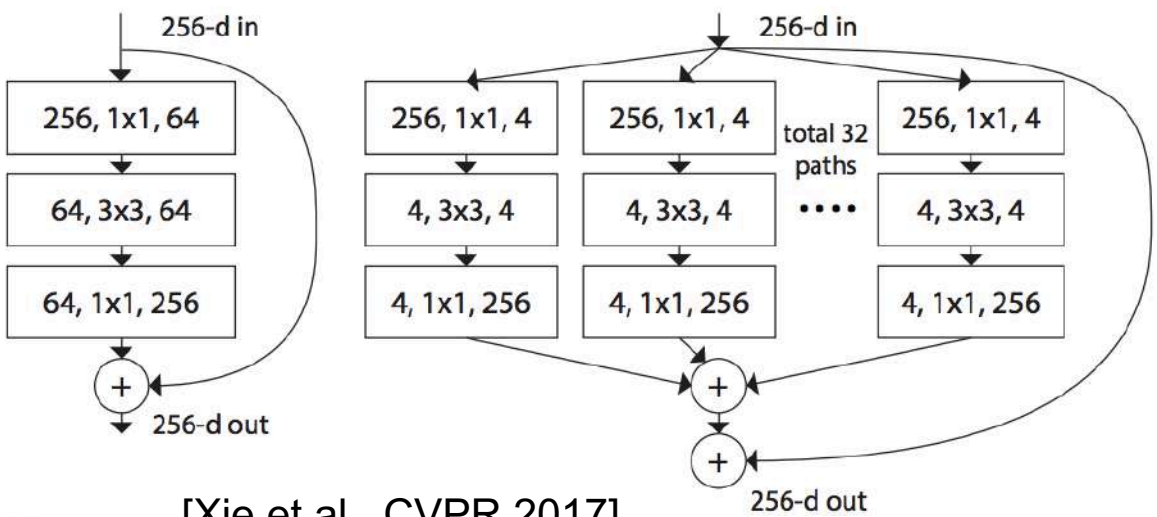


Image Source: Stanford cs231n

Basic residual block Wide residual block

[Zagoruyko et al., BMVC 2016]

ResNeXt



[Xie et al., CVPR 2017]

Increase accuracy ***without*** going deeper!

Part 1: Hardware Platforms for DNN Processing

GPUs and CPUs Targeting Deep Learning

Intel Xeon Scalable CPU (2019)

Nvidia's V100 GPU (2018)

Use **matrix multiplication libraries** on CPUs and GPUs

Matrix Multiplication Libraries

- Implementation: **Matrix Multiplication (GEMM)**
  - **CPU:** OpenBLAS, Intel MKL, etc
  - **GPU:** cuBLAS, cuDNN, etc
- Library will note shape of the matrix multiply and select implementation optimized for that shape.
- Optimization usually involves proper tiling to storage hierarchy

Map DNN to a Matrix Multiplication

Convolution:

Filter

1	2
3	4

*

Input Fmap

1	2	3
4	5	6
7	8	9

=

Output Fmap

1	2
3	4

Matrix Mult:

Toeplitz Matrix
(w/ redundant data)

1	2	3	4

×

1	2	4	5
2	3	5	6
4	5	7	8
5	6	8	9

=

1	2	3	4

Data is repeated

Goal: Reduced number of operations to **increase throughput**

Analogy: Gauss's Multiplication Algorithm

$$(a+bi)(c+di) = (ac-bd) + (bc+ad)i.$$

4 multiplications + 3 additions

$$k_1 = c \cdot (a+b)$$

$$k_2 = a \cdot (d-c)$$

$$k_3 = b \cdot (c+d)$$

$$\text{Real part} = k_1 - k_3$$

$$\text{Imaginary part} = k_1 + k_2.$$

3 multiplications + 5 additions

Reduce number of multiplications, but **increase** number of additions

Reduce Operations in Matrix Multiplication

- **Fast Fourier Transform** [Mathieu, ICLR 2014]
  - **Pro:** Direct convolution $O(N_o^2 N_f^2)$ to $O(N_o^2 \log_2 N_o)$
  - **Con:** Increase storage requirements
- **Strassen** [Cong, ICANN 2014]
  - **Pro:** $O(N^3)$ to $(N^{2.807})$
  - **Con:** Numerical stability
- **Winograd** [Lavin, CVPR 2016]
  - **Pro:** 2.25x speed up for 3x3 filter
  - **Con:** Specialized processing depending on filter size

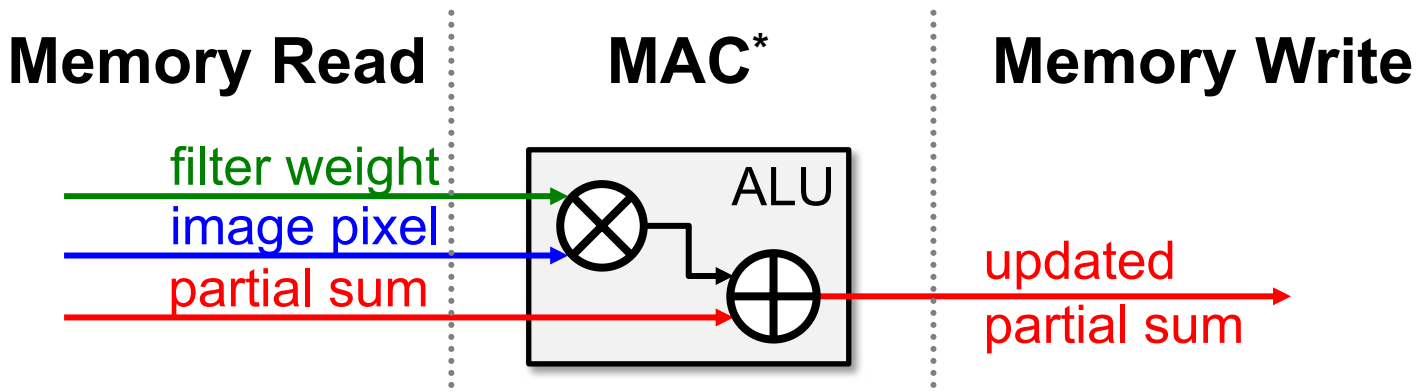
Specialized Hardware (Accelerators)

Properties We Can Leverage

- Operations exhibit **high parallelism**
  → **high throughput** possible
- Memory Access is the Bottleneck

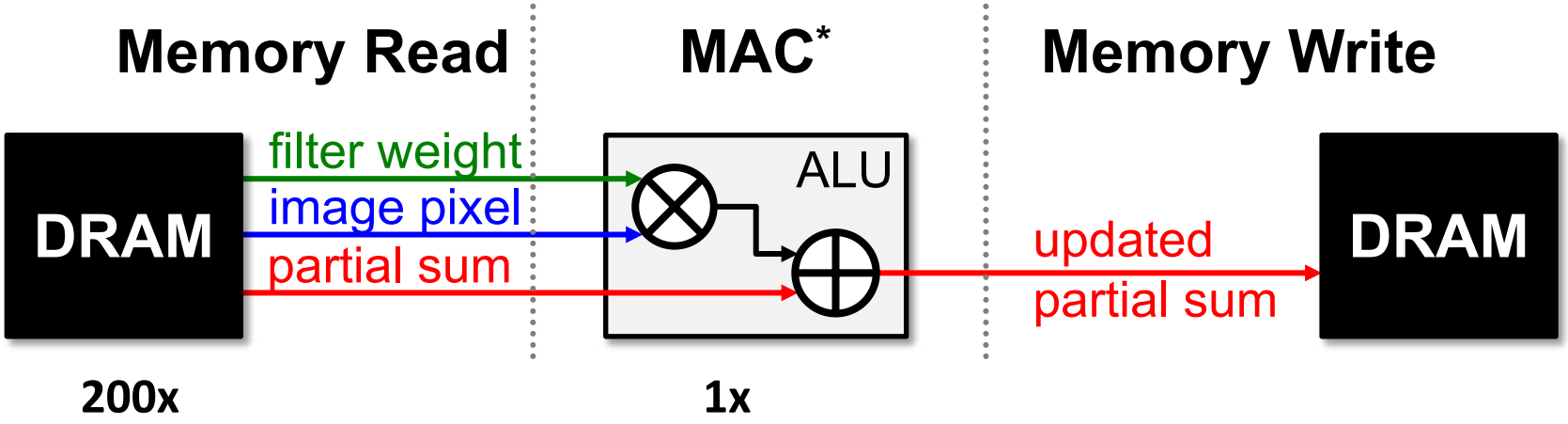
Properties We Can Leverage

- Operations exhibit **high parallelism**
  → **high throughput** possible
- Memory Access is the Bottleneck



Properties We Can Leverage

- Operations exhibit **high parallelism**
  → **high throughput** possible
- Memory Access is the Bottleneck

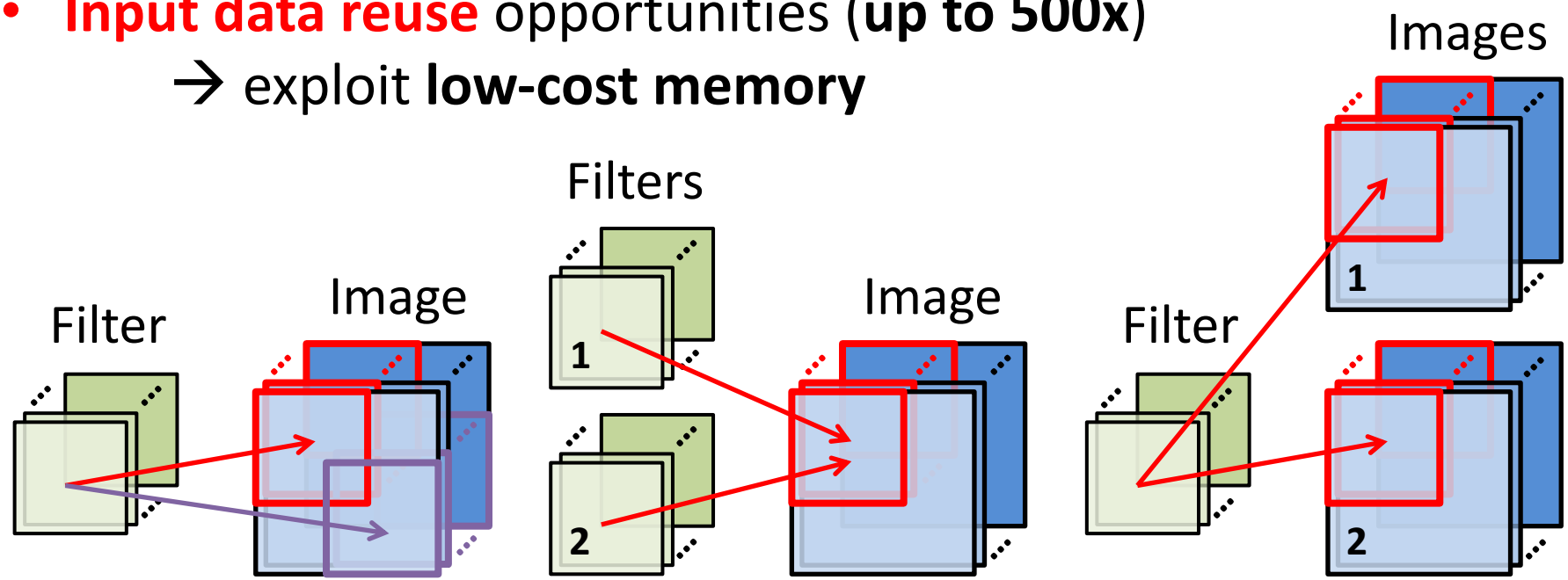


<u>Worst Case</u>: all memory R/W are **DRAM** accesses

- Example: AlexNet [NeurIPS 2012] has **724M** MACs
  → **2896M** DRAM accesses required

Properties We Can Leverage

- Operations exhibit **high parallelism**
  → **high throughput** possible
- **Input data reuse** opportunities (**up to 500x**)
  → exploit **low-cost memory**



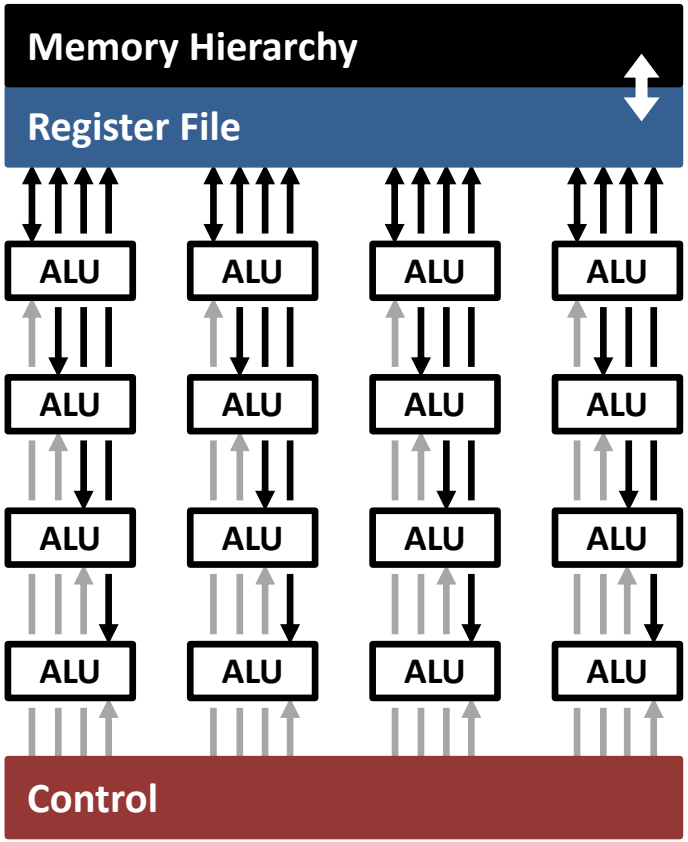
Convolutional Reuse
(pixels, weights)

Image Reuse
(pixels)

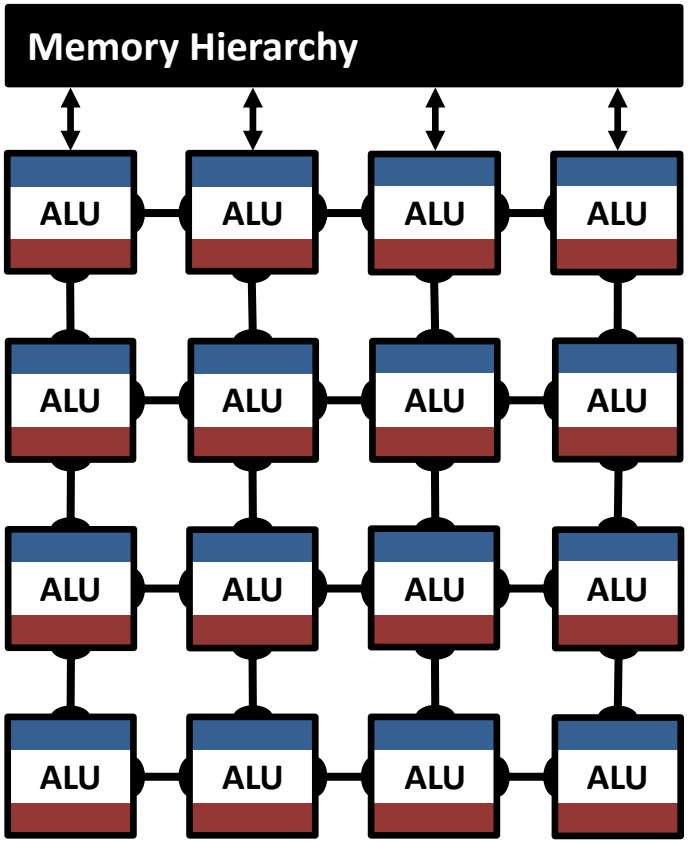
Filter Reuse
(weights)

Highly-Parallel Compute Paradigms

Temporal Architecture (SIMD/SIMT)



Spatial Architecture (Dataflow Processing)



Advantages of Spatial Architecture

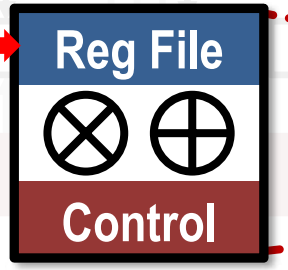
Temporal Architecture
(SIMD/SIMT)

Efficient Data Reuse
Distributed local storage (RF)

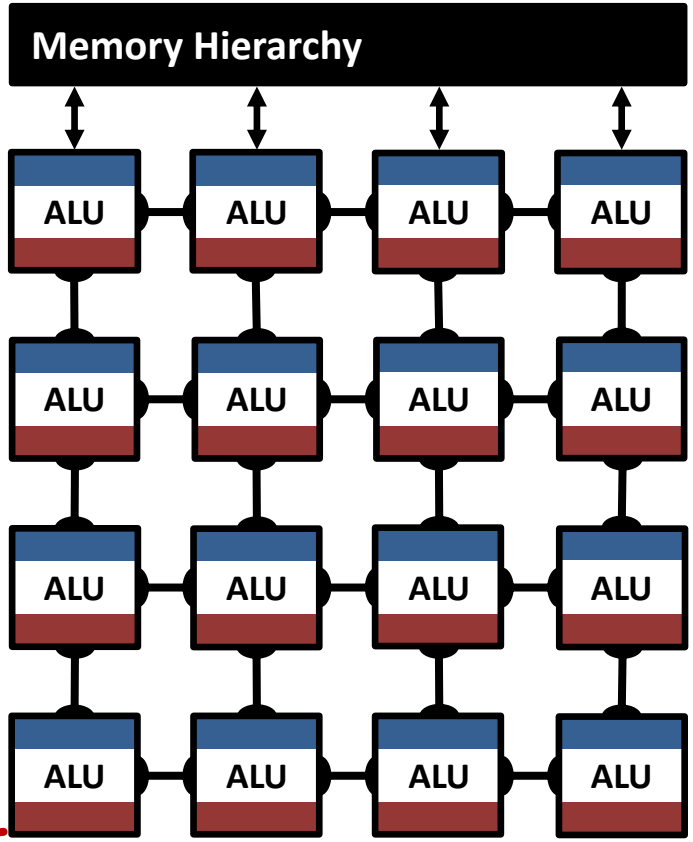
Inter-PE Communication
Sharing among regions of PEs

Processing Element (PE)

0.5 – 1.0 kB

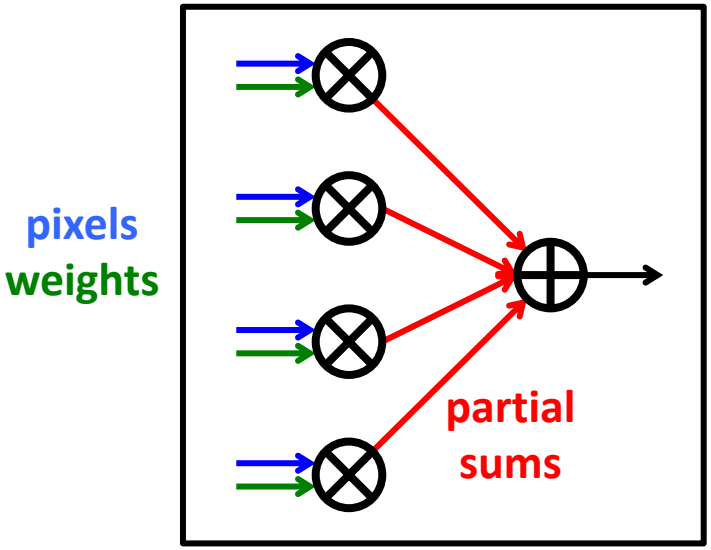


Spatial Architecture (Dataflow Processing)



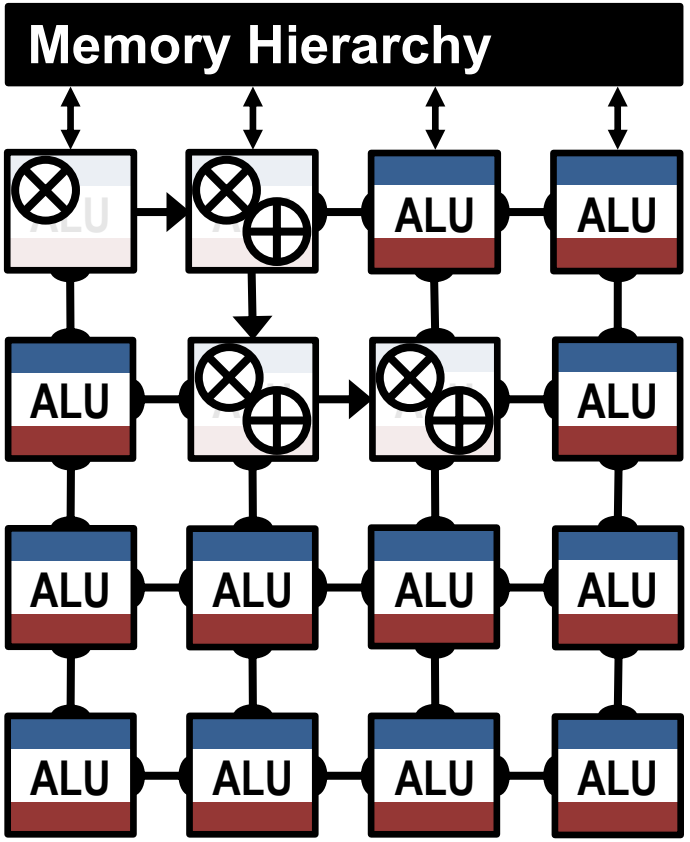
How to Map the Dataflow?

CNN Convolution



?

Spatial Architecture (Dataflow Processing)

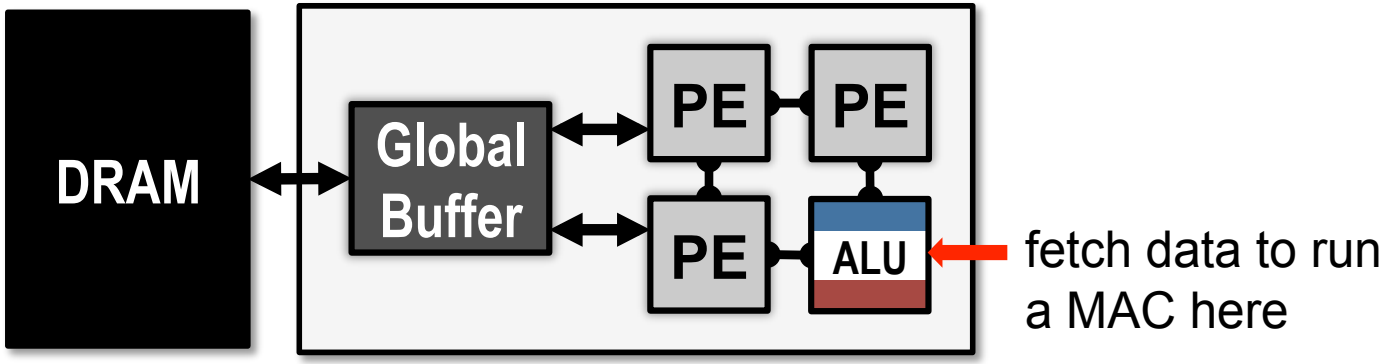


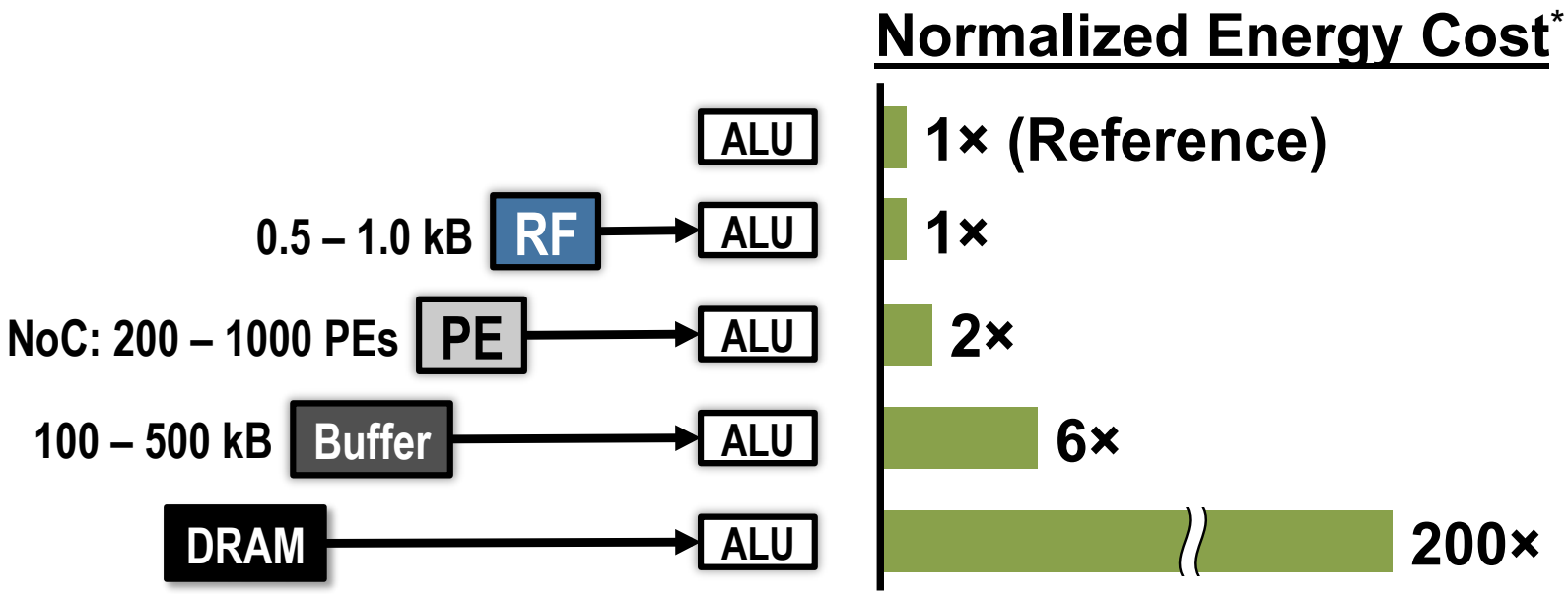
Goal: Increase reuse of input data (**weights** and **pixels**) and local **partial sums** accumulation

Energy-Efficient Dataflow

Y.-H. Chen, J. Emer, V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," ISCA 2016

Data Movement is Expensive

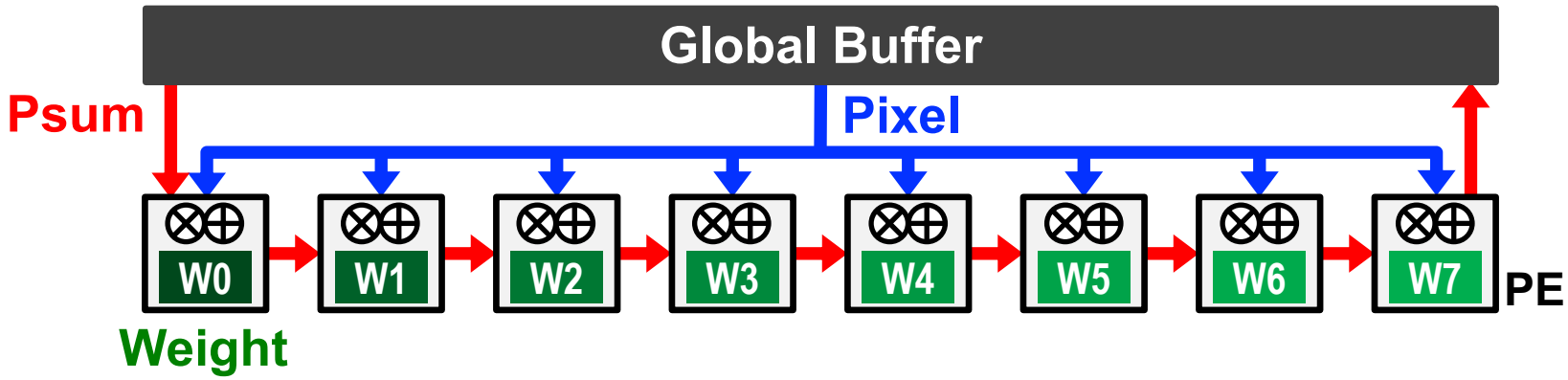




* measured from a commercial 65nm process

Maximize data reuse at low cost levels of hierarchy

Weight Stationary (WS)

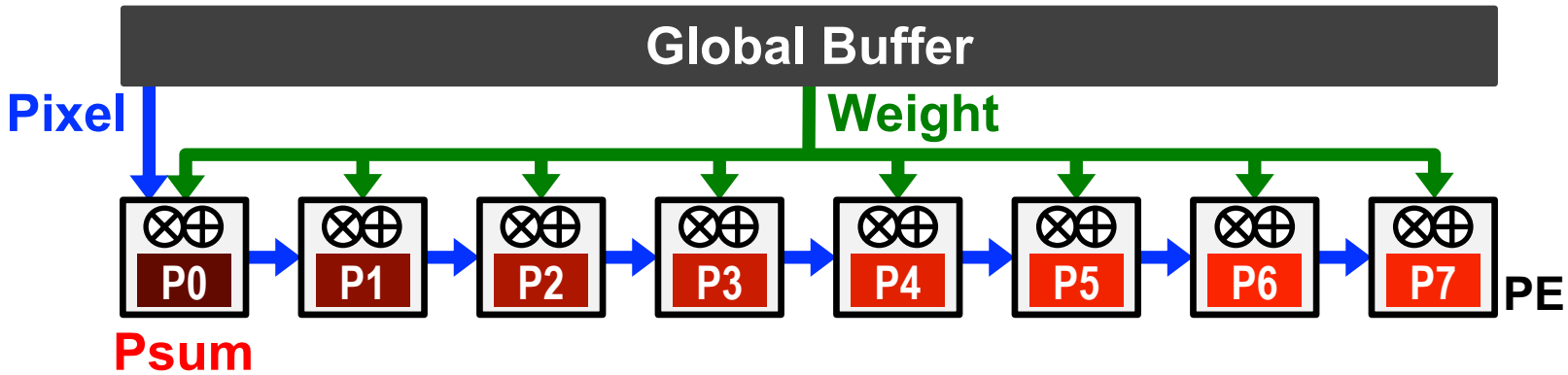


- **Minimize weight** read energy consumption
  - maximize convolutional and filter reuse of weights
- **Examples:**

[**Chakradhar**, *ISCA* 2010] [**nn-X (NeuFlow)**, *CVPRW* 2014]

[**Park**, *ISSCC* 2015] [**Origami**, *GLSVLSI* 2015]

Output Stationary (OS)

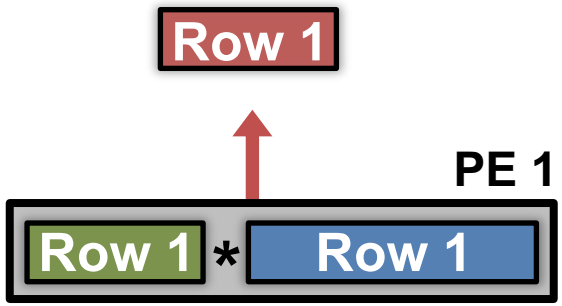


- **Minimize partial sum** R/W energy consumption
  - maximize local accumulation
- **Examples:**

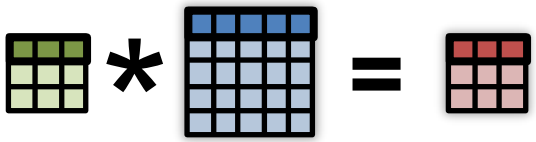
  [**Gupta**, *ICML* 2015] [**ShiDianNao**, *ISCA* 2015]

  [**Peemen**, *ICCD* 2013]

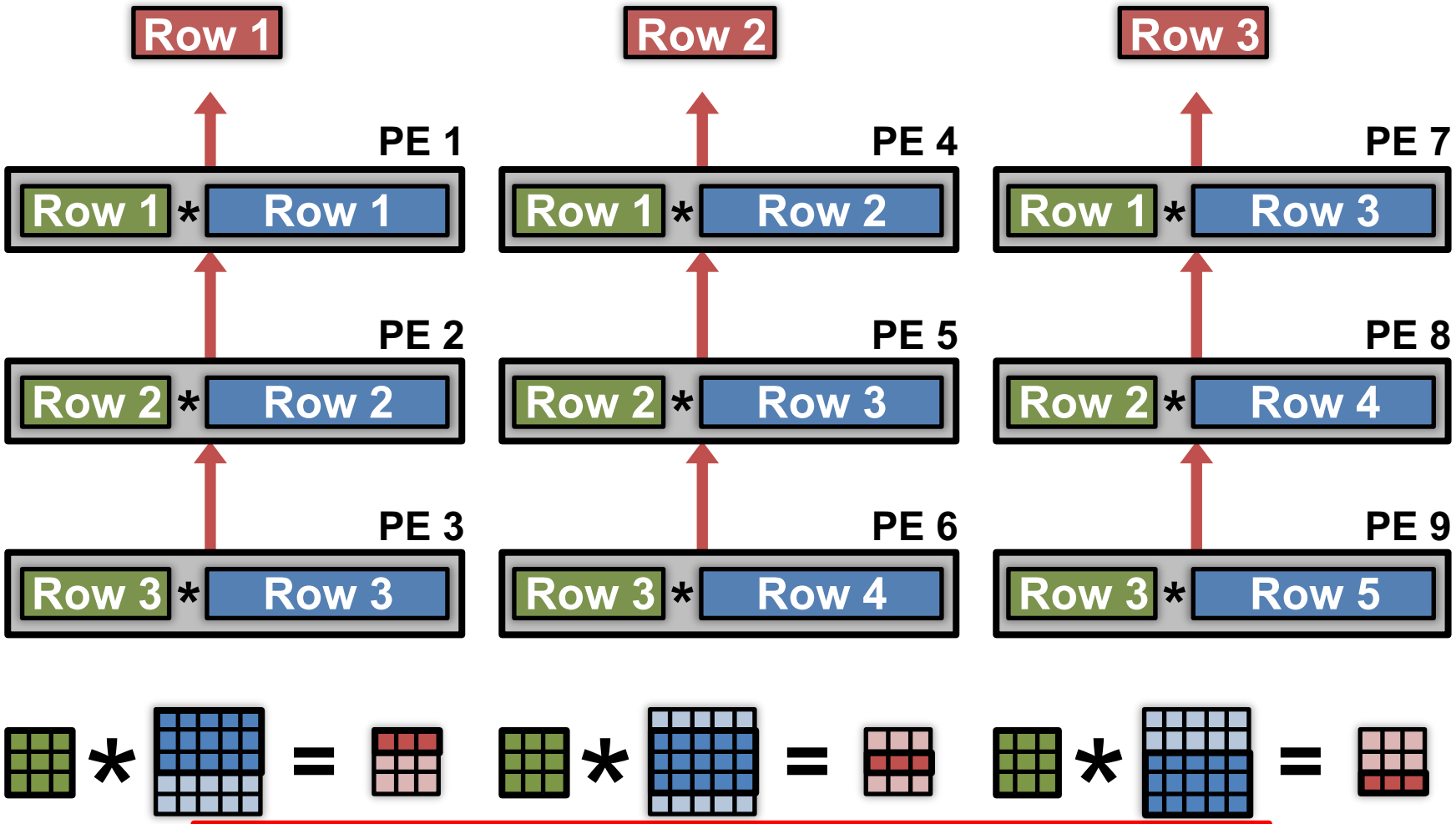
Row Stationary Dataflow



- Maximize row **convolutional reuse** in RF
  - Keep a **filter** row and **fmap** sliding window in RF
- Maximize row **psum accumulation** in RF



Row Stationary Dataflow



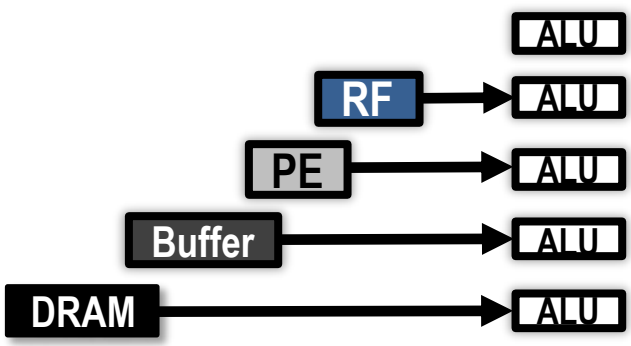
Optimize for **overall energy efficiency** instead for only a certain data type

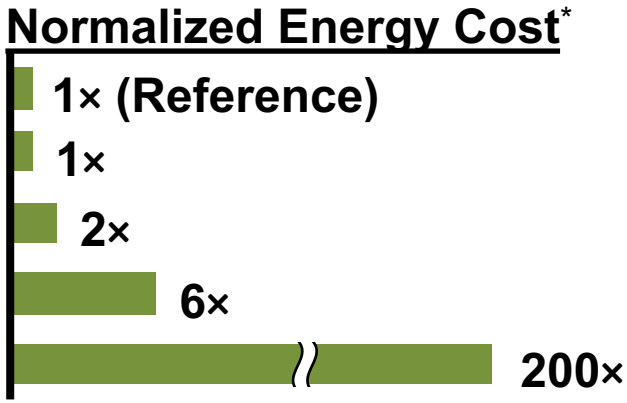
Evaluate Reuse in Different Dataflows

- **Weight Stationary**
  - Minimize movement of filter weights
- **Output Stationary**
  - Minimize movement of partial sums
- **No Local Reuse**
  - Don't use any local PE storage. Maximize global buffer size.
- **Row Stationary**

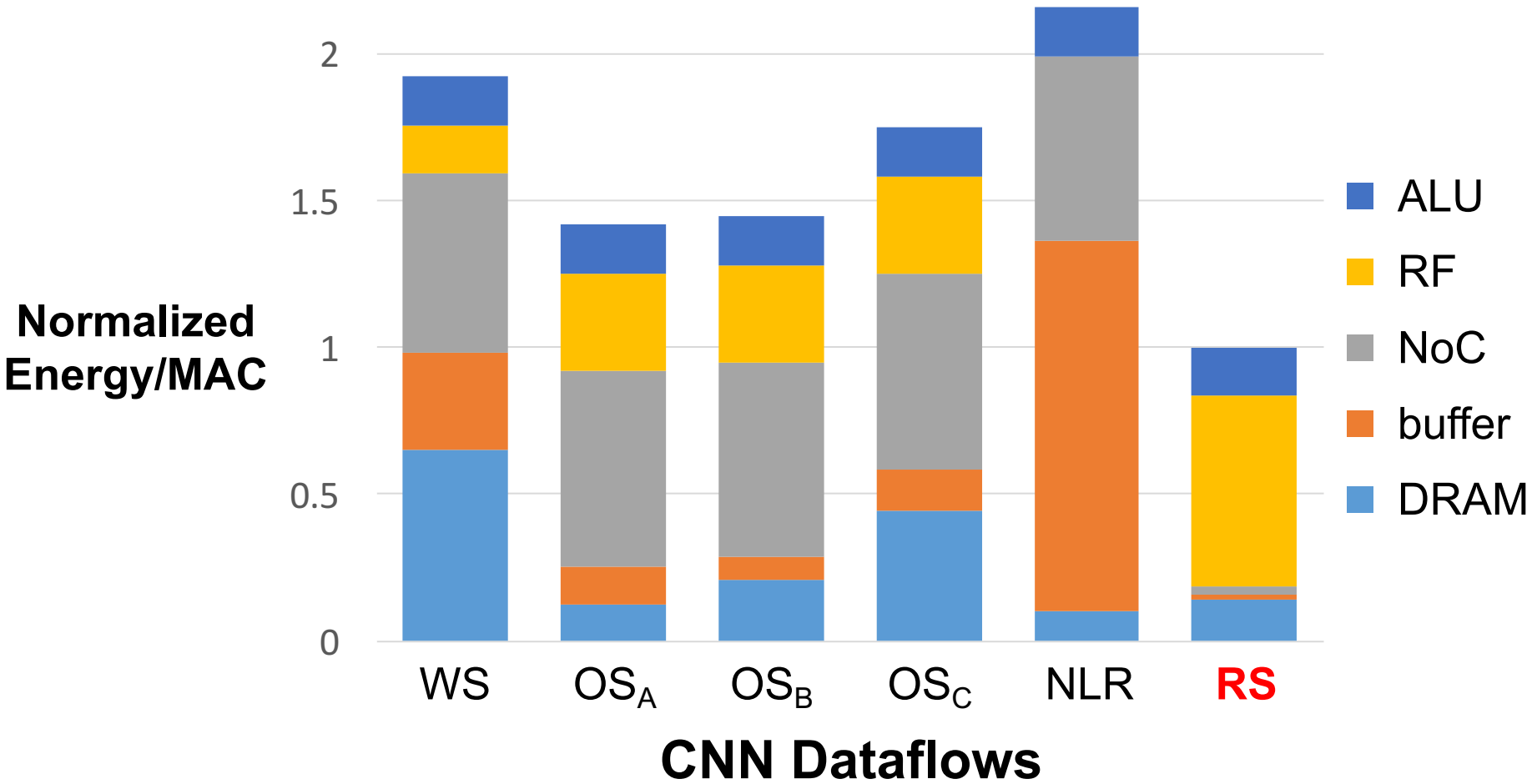
Evaluation Setup

- Same Total Area
- AlexNet
- 256 PEs
- Batch size = 16





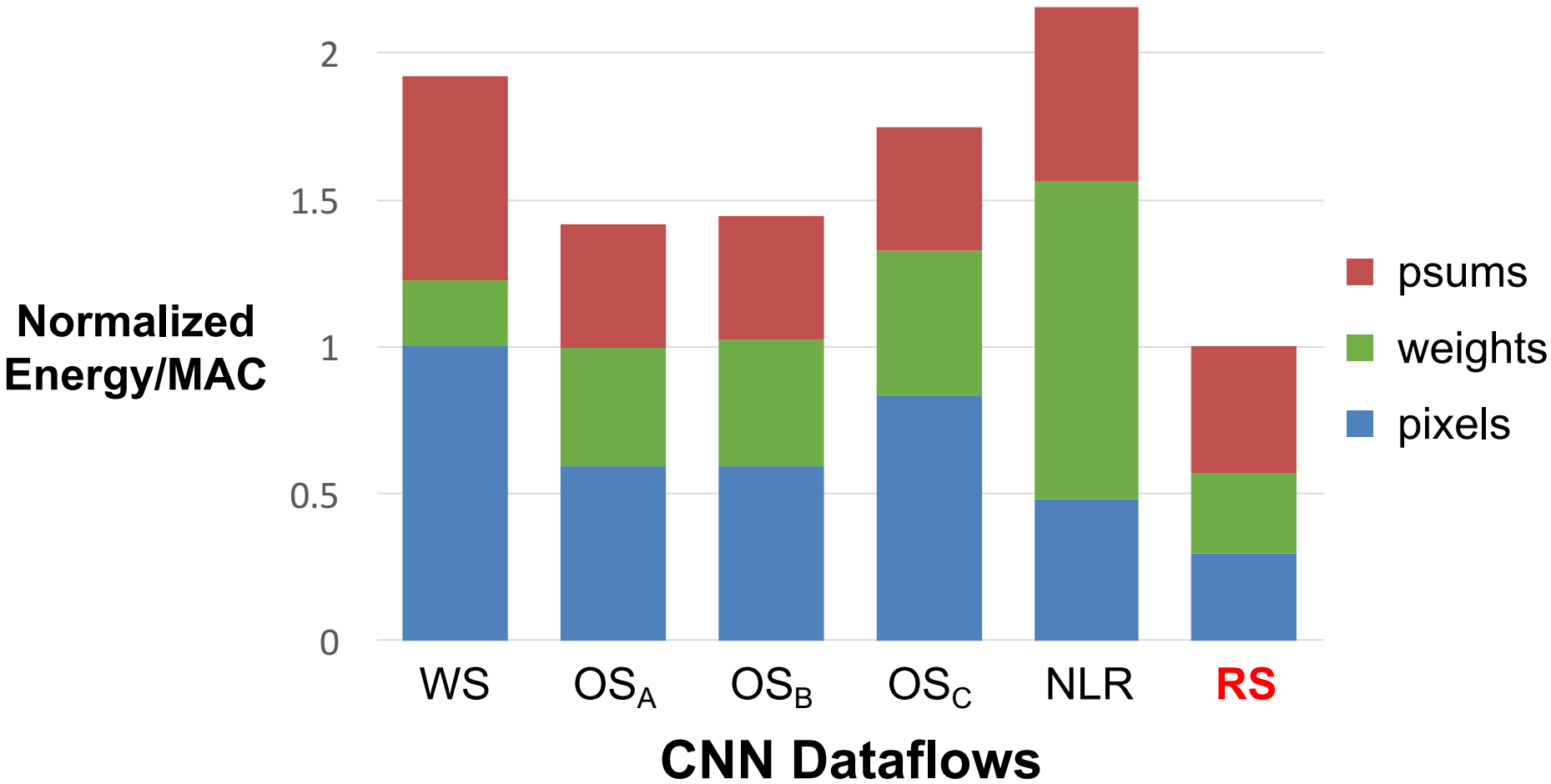
Dataflow Comparison: CONV Layers



RS uses **1.4× – 2.5× lower** energy than other dataflows

[Chen et al., ISCA 2016]

Dataflow Comparison: CONV Layers

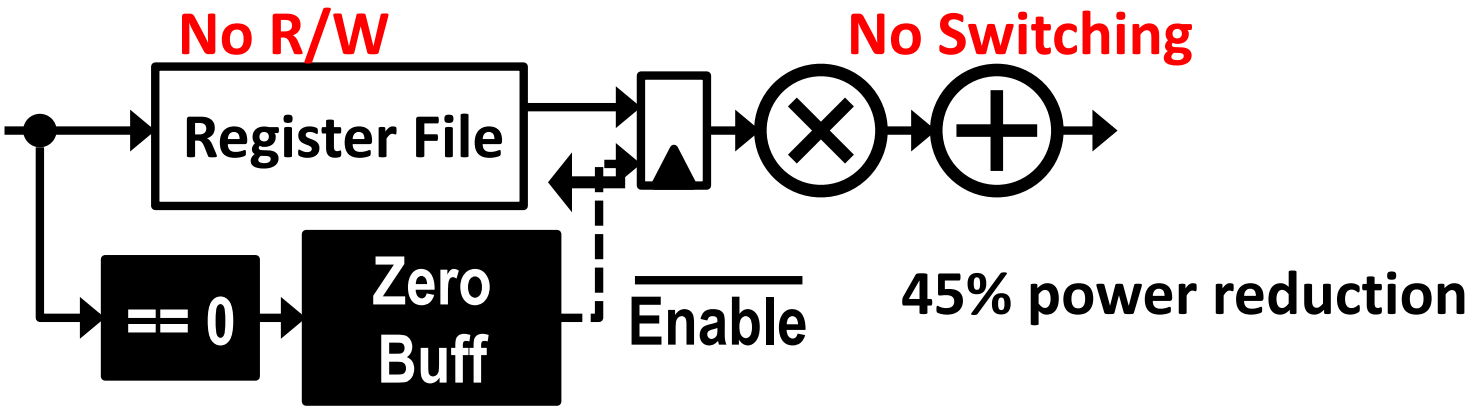


RS optimizes for the best **overall** energy efficiency

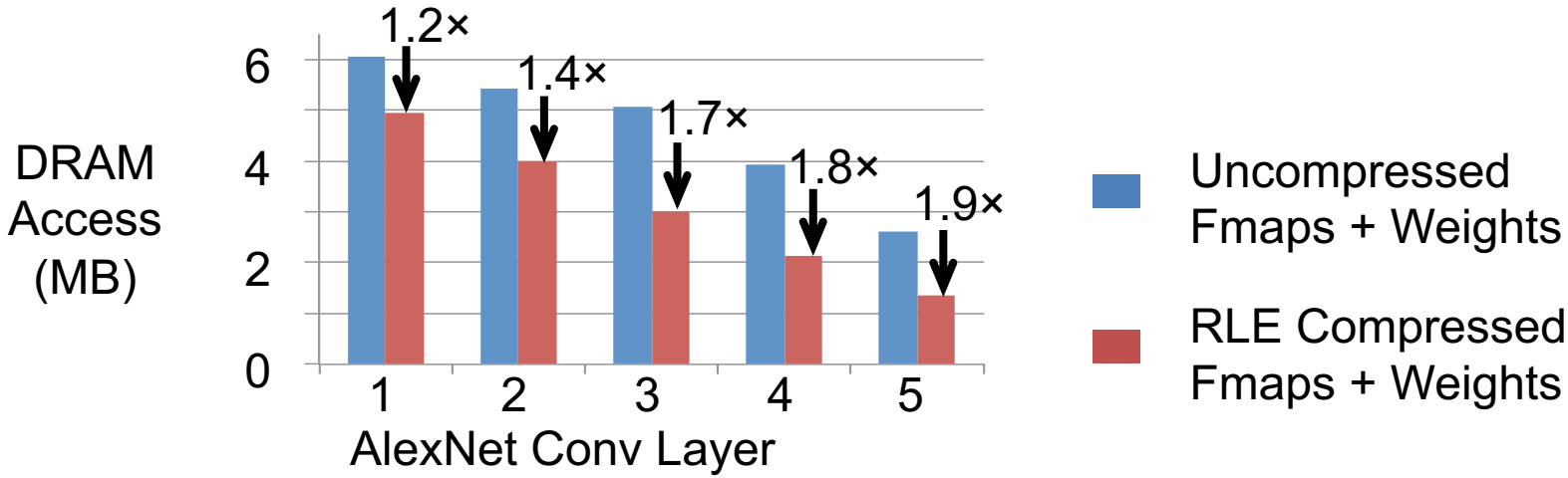
[Chen et al., ISCA 2016]

Exploit Sparsity

Method 1. Skip memory access and computation

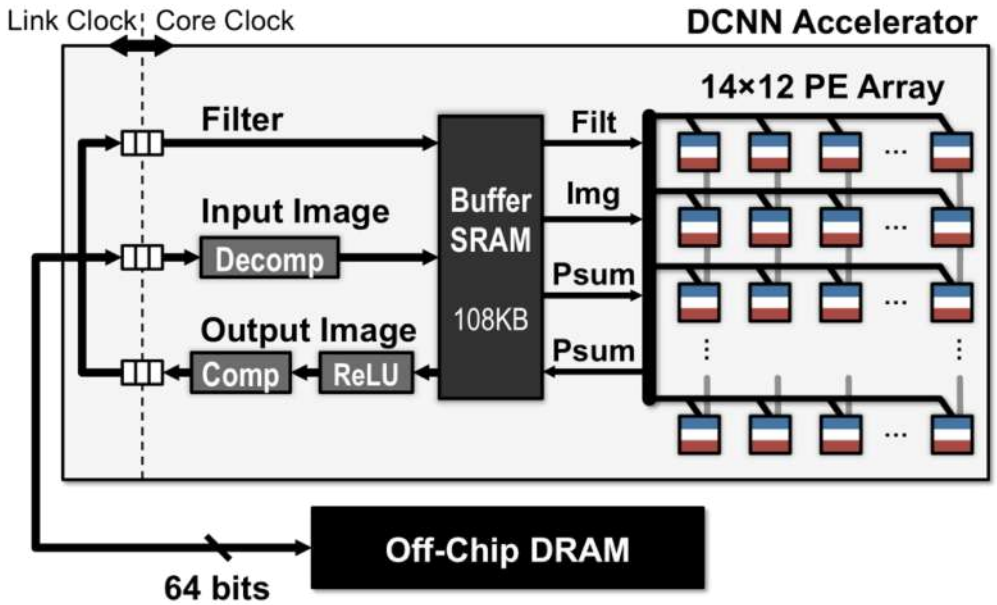


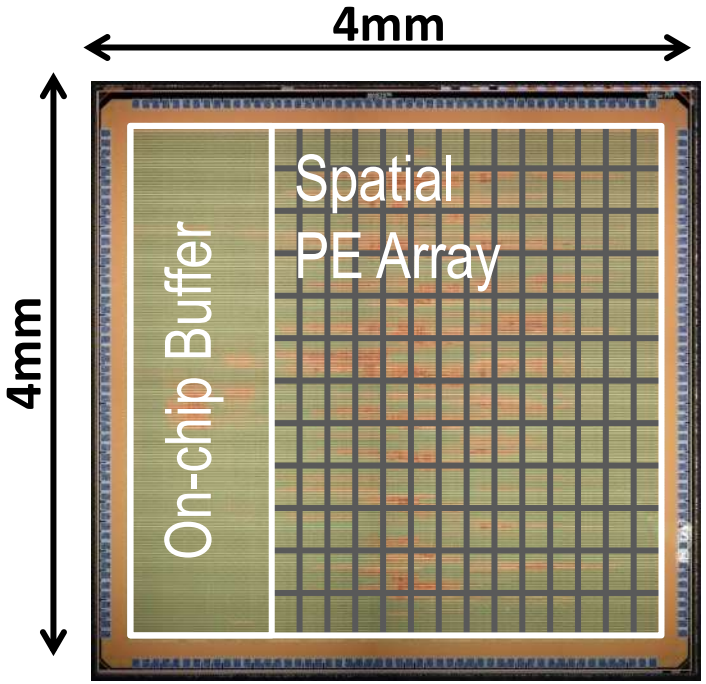
Method 2. Compress data to reduce storage and data movement



[Chen et al., ISSCC 2016]

Eyeriss: Deep Neural Network Accelerator





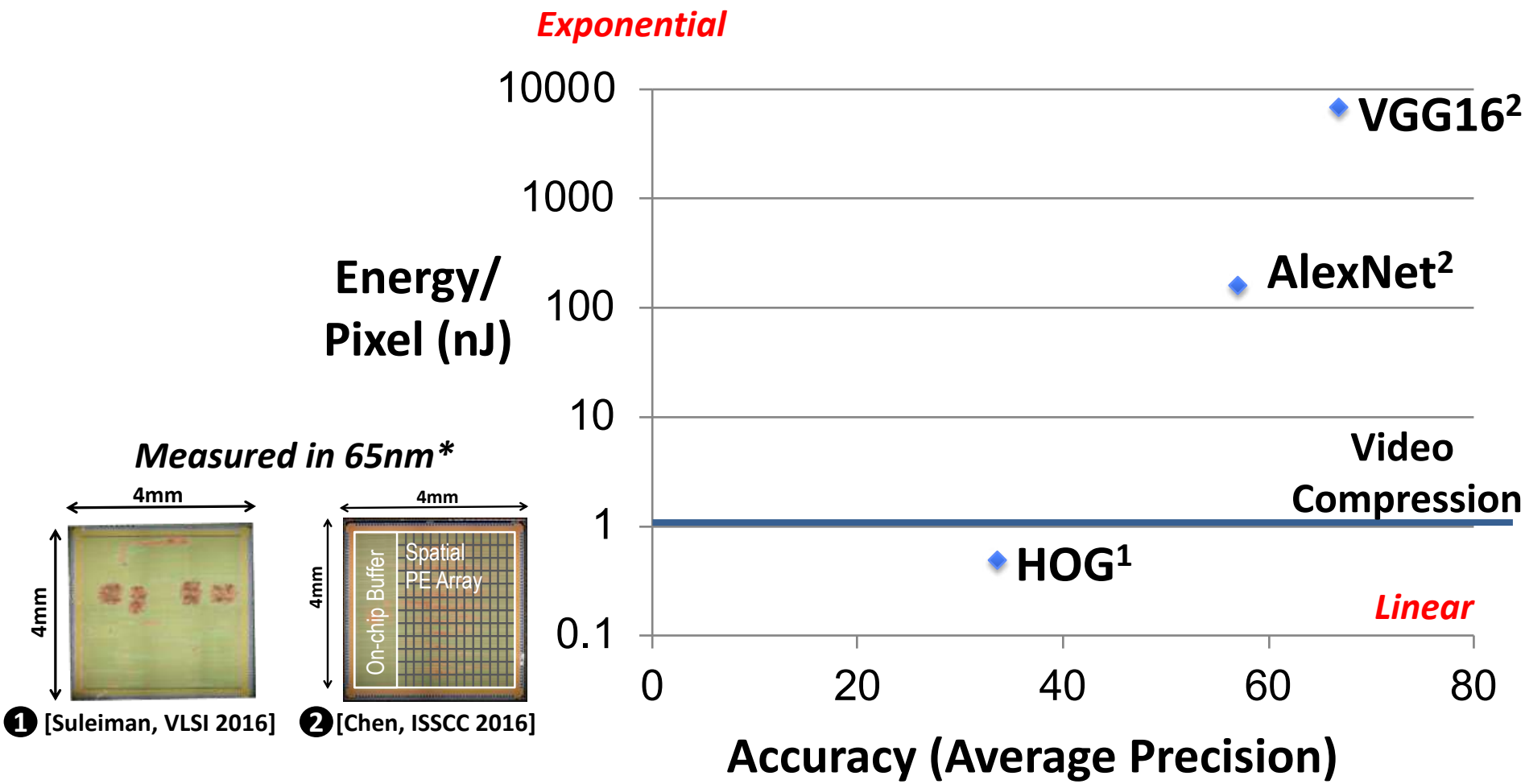
[Chen et al., ISSCC 2016, ISCA 2016]

Exploits data reuse for **100x** reduction in memory accesses from global buffer and **1400x** reduction in memory accesses from off-chip DRAM

Overall **>10x energy reduction** compared to a mobile GPU (Nvidia TK1)

Results for AlexNet

Features: Energy vs. Accuracy



** Only feature extraction. Does not include data, classification energy, augmentation and ensemble, etc.*

Measured in on VOC 2007 Dataset

1. DPM v5 [Girshick, 2012]
2. Fast R-CNN [Girshick, CVPR 2015]

Benchmarking Metrics for DNN Hardware

How can we compare designs?

V. Sze, Y.-H. Chen, T-J. Yang, J. Emer,
"***Efficient Processing of Deep Neural Networks: A Tutorial and Survey***,"
Proceedings of the IEEE, Dec. 2017

Metrics for DNN Hardware

- **Accuracy**
  - Quality of result for a given task
- **Throughput**
  - Analytics on high volume data
  - Real-time performance (e.g., video at 30 fps)
- **Latency**
  - For interactive applications (e.g., autonomous navigation)
- **Energy and Power**
  - Edge and embedded devices have limited battery capacity
  - Data centers have stringent power ceilings due to cooling costs
- **Hardware Cost**
  - $$$

Specifications to Evaluate Metrics

- **Accuracy**
  - Difficulty of dataset and/or task should be considered
- **Throughput**
  - Number of cores (include utilization along with peak performance)
  - Runtime for running specific DNN models
- **Latency**
  - Include batch size used in evaluation
- **Energy and Power**
  - Power consumption for running specific DNN models
  - Include external memory access
- **Hardware Cost**
  - On-chip storage, number of cores, chip area + process technology

Example: Metrics of Eyeriss Chip

ASIC Specs	Input
Process Technology	65nm LP TSMC (1.0V)
Total Core Area (mm^2)	12.25
Total On-Chip Memory (kB)	192
Number of Multipliers	168
Clock Frequency (MHz)	200
Core area (mm^2) /multiplier	0.073
On-Chip memory (kB) / multiplier	1.14
Measured or Simulated	Measured

Metric	Units	Input
Name of CNN Model	Text	AlexNet
Top-5 error classification on ImageNet	#	19.8
Supported Layers		All CONV
Bits per weight	#	16
Bits per input activation	#	16
Batch Size	#	4
Runtime	ms	115.3
Power	mW	278
Off-chip Access per Image Inference	MBytes	3.85
Number of Images Tested	#	100

Comprehensive Coverage

- **All metrics** should be reported for fair evaluation of design tradeoffs
- Examples of what can happen if certain metric is omitted:
  - **Without the accuracy given for a specific dataset and task**, one could run a simple DNN and claim low power, high throughput, and low cost – however, the processor might not be usable for a meaningful task
  - **Without reporting the off-chip bandwidth**, one could build a processor with only multipliers and claim low cost, high throughput, high accuracy, and low chip power – however, when evaluating system power, the off-chip memory access would be substantial
- Are results measured or simulated? On what test data?

Evaluation Process

The evaluation process for whether a DNN system is a viable solution for a given application might go as follows:

1. **Accuracy** determines if it can perform the given task
2. **Latency and throughput** determine if it can run fast enough and in real-time
3. **Energy and power consumption** will primarily dictate the form factor of the device where the processing can operate
4. **Cost**, which is primarily dictated by the chip area, determines how much one would pay for this solution

Part 2: Co-Design of Algorithms and Hardware for DNNs

Approaches

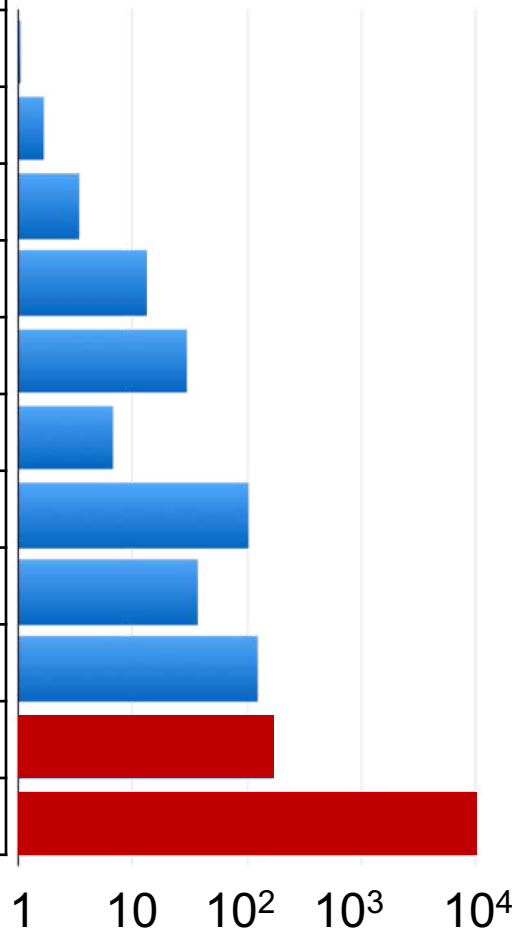
- **<u>Reduce size</u> of operands for storage/compute**
  - Floating point → Fixed point
  - Bit-width reduction
  - Non-linear quantization

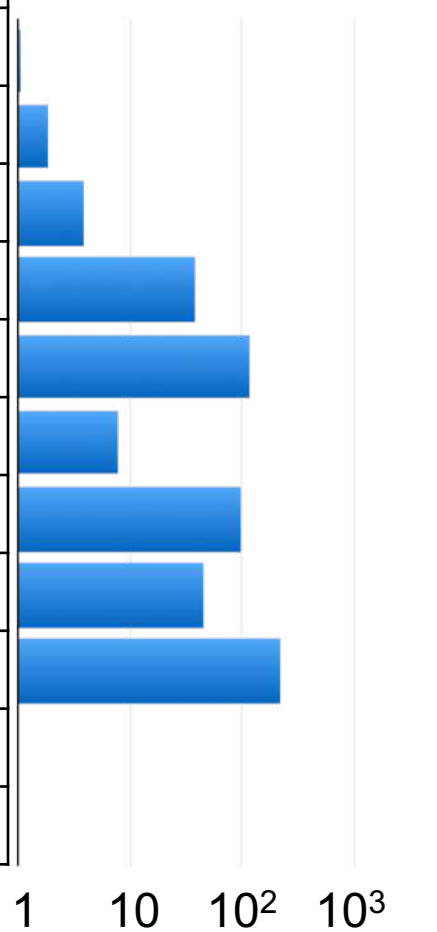
- **<u>Reduce number</u> of operations for storage/compute**
  - Exploit Activation Statistics (Compression)
  - Network Pruning
  - Compact Network Architectures

Reduced Precision

Cost Per Operation

Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A





[Horowitz, ISSCC 2014]

Floating Point → Fixed Point

Mantissa (m): number of levels
Exponent (e): scale to a target range
Sign (s): indicates if number is positive or negative

Floating Point $(-1)^s \times m \times 2^{(e-127)}$

sign | exponent (8-bits) | mantissa (23-bits)

32-bit float

1	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0

-1.42122425 x 10^-13 ***s*** = 1 ***e*** = 70 ***m*** = 20482

Fixed Point $(-1)^s \times m$

sign | mantissa (7-bits)

8-bit fixed

0	1	1	0	0	1	1	0

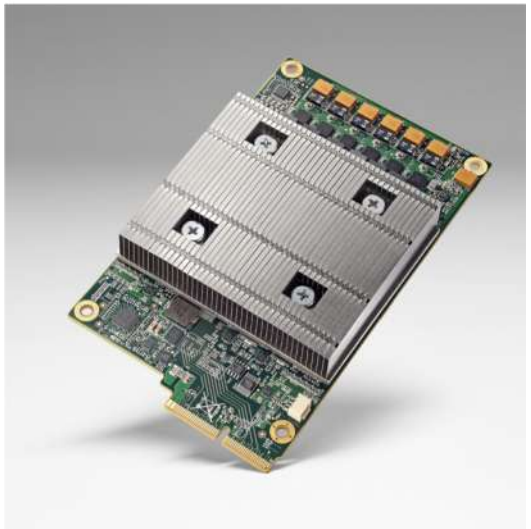
12.75

integer (4-bits) | fractional (3-bits)

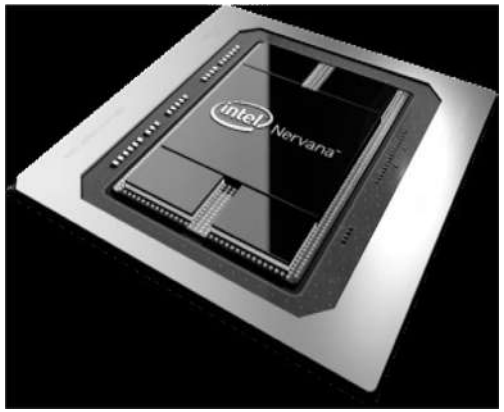
s = 0 ***m***=*102*

Commercial Products Support Reduced Precision

Nvidia's Pascal (2016)

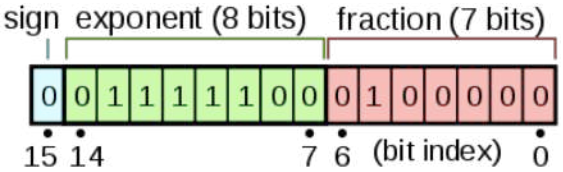


Google's TPU (2016)



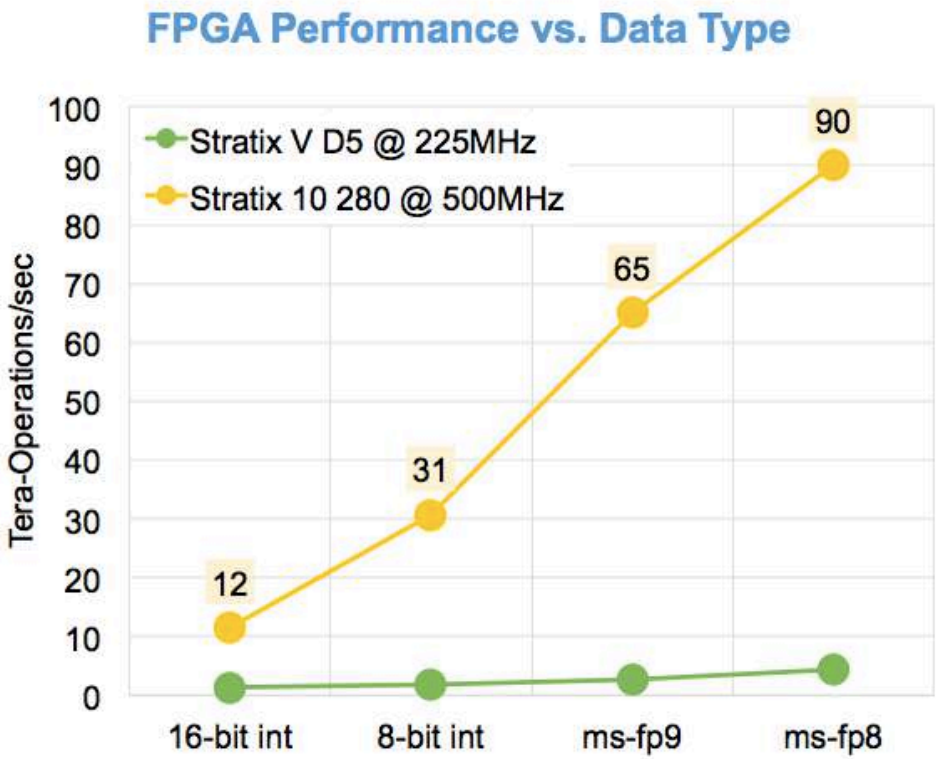
Intel's NNP-L (2019)

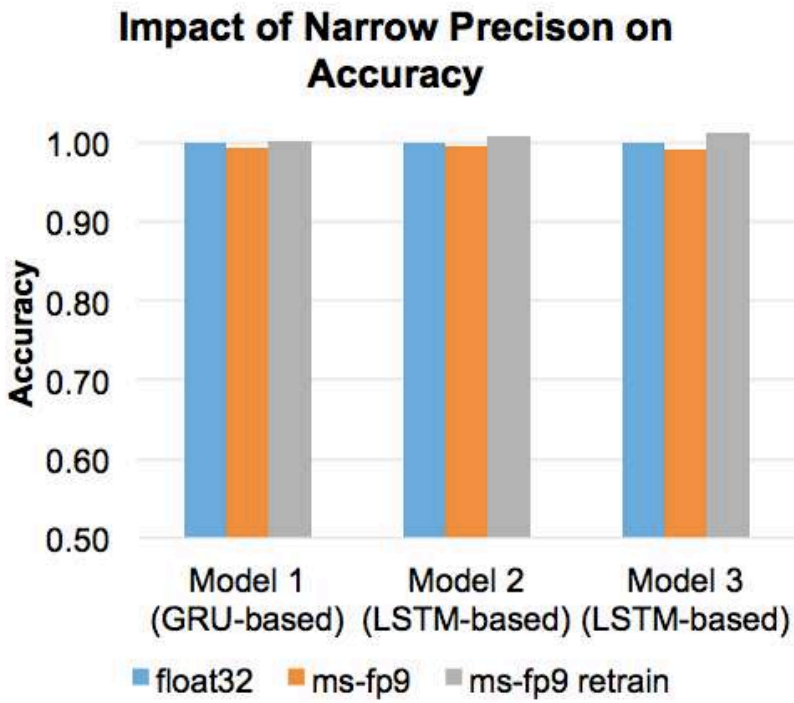
8-bit Inference & bfloat16 for Training



Microsoft BrainWave

Narrow Precision for Inference





Custom 8-bit floating point format ("ms-fp8")

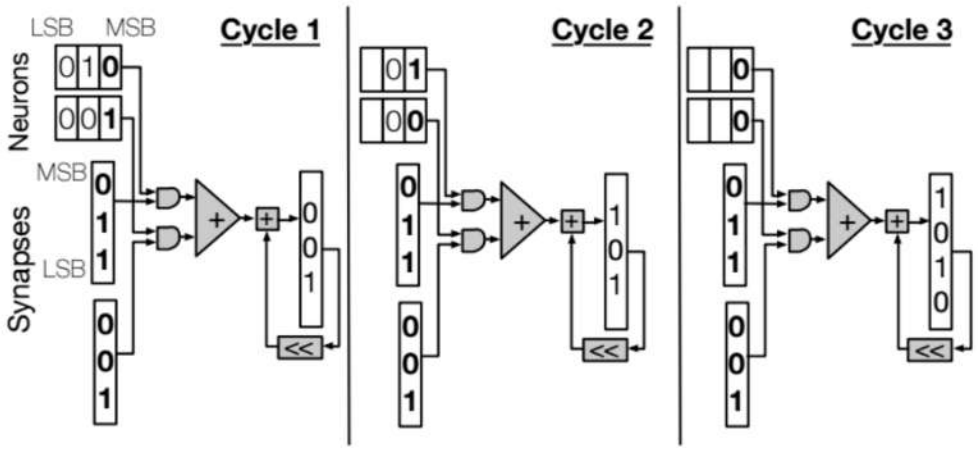
[Chung et al., Hot Chips 2017]

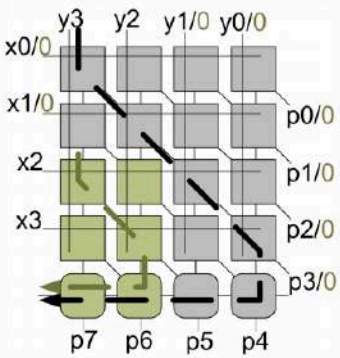
Reduced Precision Hardware

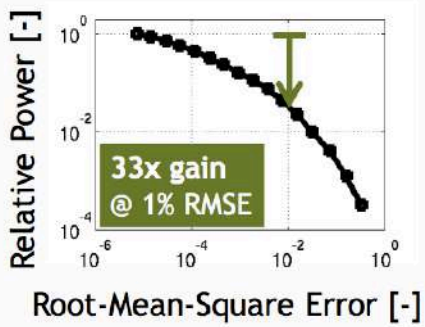
Stripes

[Judd et al., MICRO 2016]

Bit-serial processing for speed







$$P_{precise} = \alpha C f V^2 \quad \Rightarrow \quad P_{imprecise} = \frac{\alpha}{k_1} C f \left(\frac{V}{k_2}\right)^2$$

KU Leuven

[Moons et al., VLSI 2016]

Voltage scaling for energy savings

Binary Nets

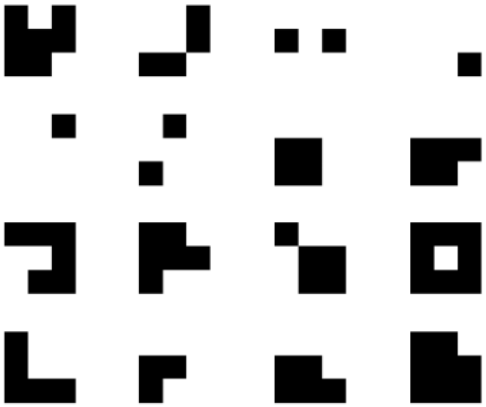
- **Binary Connect (BC)**
  - Weights {-1,1}, Activations 32-bit float
  - MAC → addition/subtraction
  - Accuracy loss: **19%** on AlexNet

[Courbariaux, NeurIPS 2015]

- **Binarized Neural Networks (BNN)**
  - Weights {-1,1}, Activations {-1,1}
  - MAC → XNOR
  - Accuracy loss: **29.8%** on AlexNet

[Courbariaux, arXiv 2016]

Binary Filters



Scale the Weights and Activations

- **Binary Weight Nets (BWN)**
  - Weights $\{-\alpha, \alpha\}$ → except first and last layers are 32-bit float
  - Activations: 32-bit float
  - α determined by the l_1-norm of all weights in a filter
  - Accuracy loss: **0.8%** on AlexNet
- **XNOR-Net**
  - Weights $\{-\alpha, \alpha\}$
  - Activations $\{-\beta_i, \beta_i\}$ → except first and last layers are 32-bit float
  - β_i determined by the l_1-norm of all activations across channels ***for given position i*** of the input feature map
  - Accuracy loss: **11%** on AlexNet

Hardware needs to support both activation precisions

Scale factors (α, β_i) can change per filter or position in filter

[Rastegari et al., BWN & XNOR-Net, ECCV 2016]

Ternary Nets

- **Allow for weights to be zero**
  - Increase sparsity, but also increase number of bits (2-bits)

- **Ternary Weight Nets (TWN)** [Li et al., arXiv 2016]
  - Weights {-w, 0, w} → except first and last layers are 32-bit float
  - Activations: 32-bit float
  - Accuracy loss: 3.7% on AlexNet

- **Trained Ternary Quantization (TTQ)** [Zhu et al., ICLR 2017]
  - Weights $\{-w_1, 0, w_2\}$ → except first and last layers are 32-bit float
  - Activations: 32-bit float
  - Accuracy loss: 0.6% on AlexNet

Non-Linear Quantization

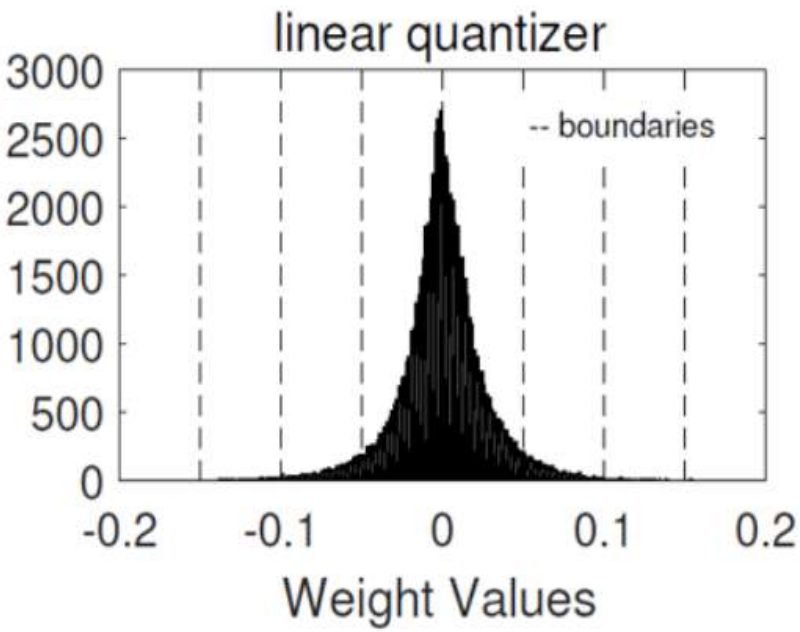
- **Precision** refers to the **number of levels**
  - Number of bits = $\log_2$ (number of levels)

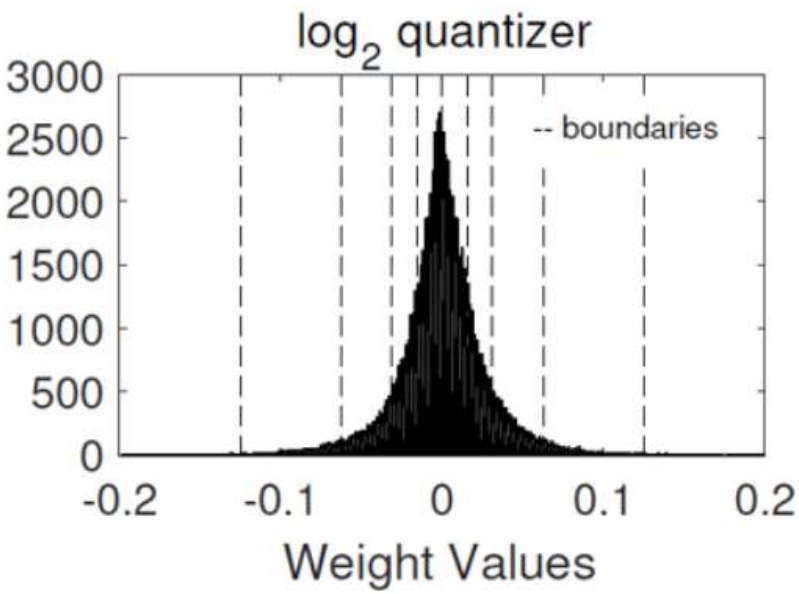
- **Quantization:** mapping data to a smaller set of **levels**
  - Linear, e.g., fixed-point
  - Non-linear
    - Computed
    - Table lookup

Objective: Reduce size to improve speed and/or reduce energy while preserving accuracy

Computed Non-linear Quantization

Log Domain Quantization





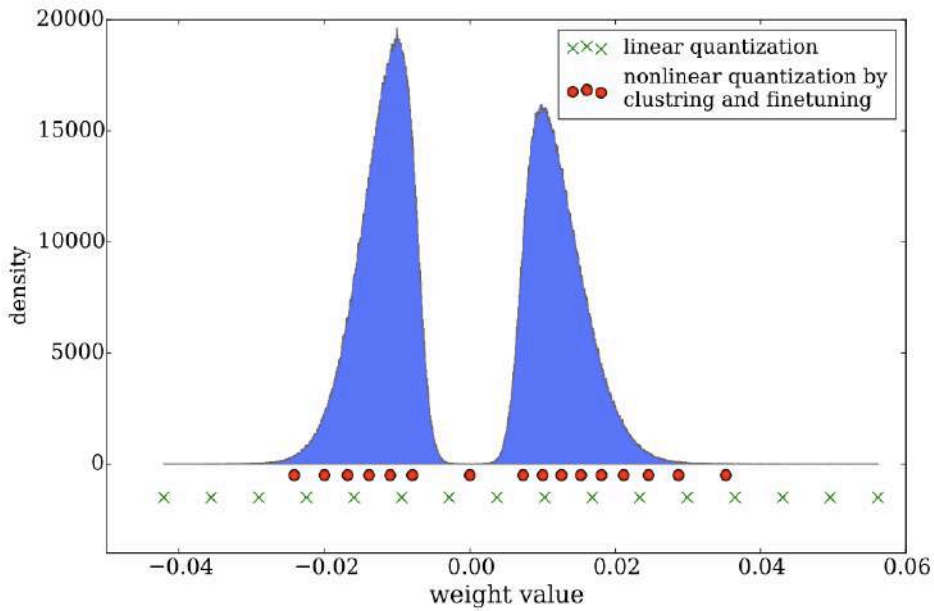
Product = X * W

Product = X << W

[Lee et al., LogNet, ICASSP 2017]

Reduce Precision Overview

- Learned mapping of data to quantization levels (e.g., k-means)



Implement with look up table

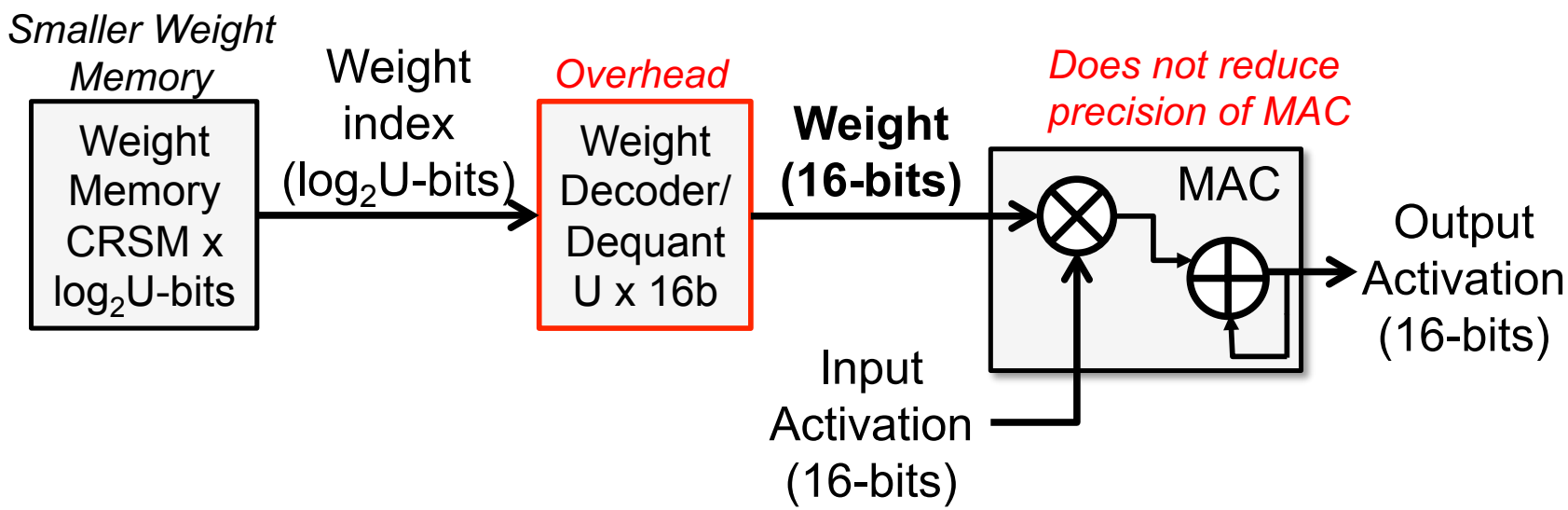
[Han et al., ICLR 2016]

- Additional Properties
  - Fixed or Variable (across data types, layers, channels, etc.)

Non-Linear Quantization Table Lookup

Trained Quantization: Find K weights via K-means clustering to reduce number of unique weights ***per layer*** (weight sharing)

Example: AlexNet (no accuracy loss)
256 unique weights for CONV layer
16 unique weights for FC layer



Consequences: Narrow weight memory and second access from (small) table

Summary of Reduce Precision

Category	Method	Weights (# of bits)	Activations (# of bits)	Accuracy Loss vs. 32-bit float (%)
Dynamic Fixed Point	w/o fine-tuning	8	10	0.4
	w/ fine-tuning	8	8	0.6
Reduce weight	Ternary weights Networks (TWN)	2*	32	3.7
	Trained Ternary Quantization (TTQ)	2*	32	0.6
	Binary Connect (BC)	1	32	19.2
	Binary Weight Net (BWN)	1*	32	0.8
Reduce weight and activation	Binarized Neural Net (BNN)	1	1	29.8
	XNOR-Net	1*	1	11
Non-Linear	LogNet	5(conv), 4(fc)	4	3.2
	Weight Sharing	8(conv), 4(fc)	16	0

* first and last layers are 32-bit float

Approaches

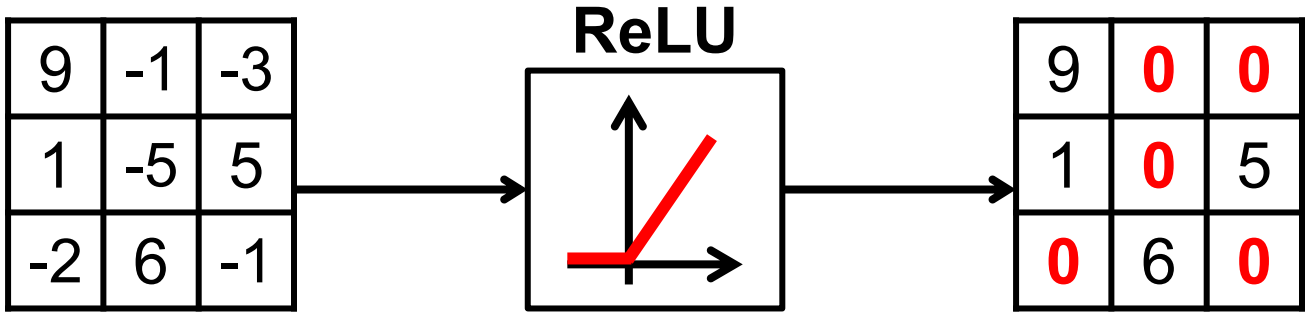
- **<u>Reduce size</u> of operands for storage/compute**
  - Floating point → Fixed point
  - Bit-width reduction
  - Non-linear quantization

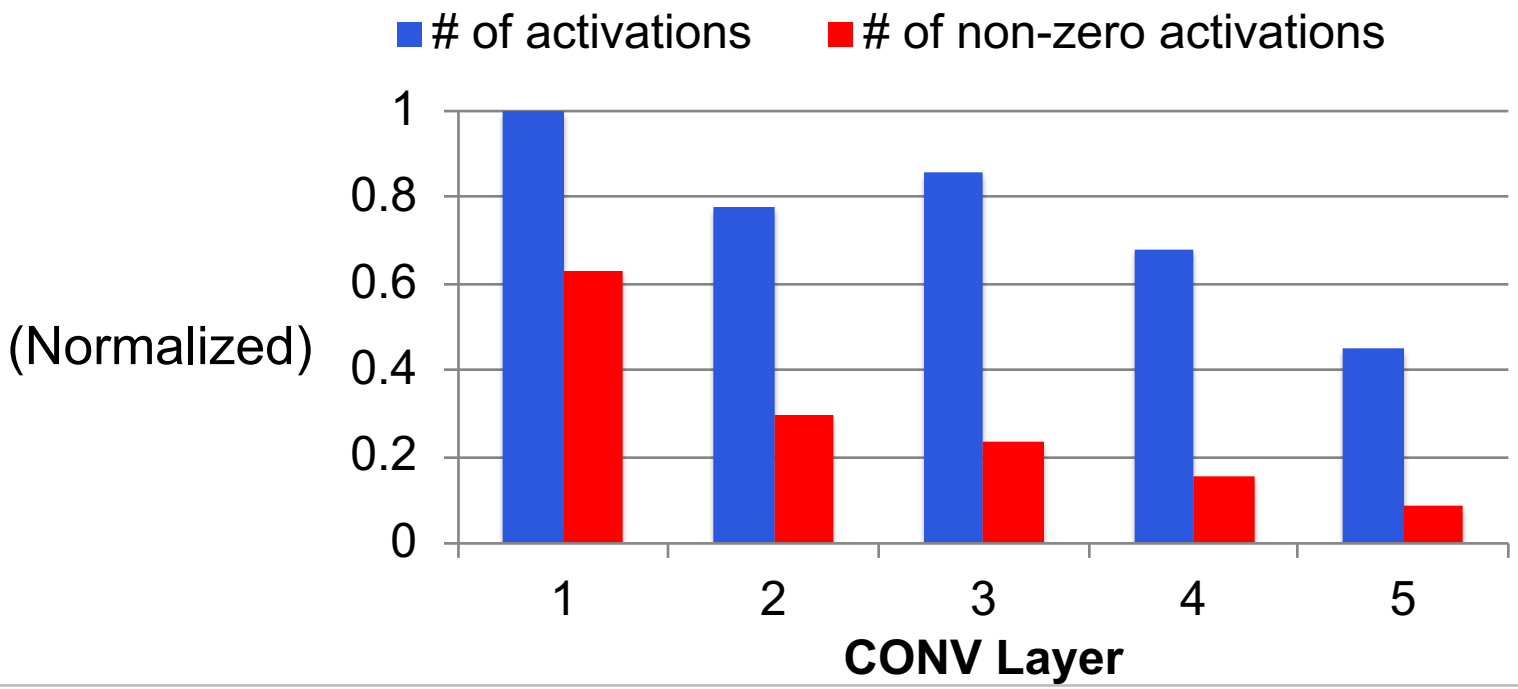
- **<u>Reduce number</u> of operations for storage/compute**
  - **Exploit Activation Statistics (Compression)**
  - **Network Pruning**
  - **Compact Network Architectures**

Exploit Sparsity

Sparsity in Feature Maps

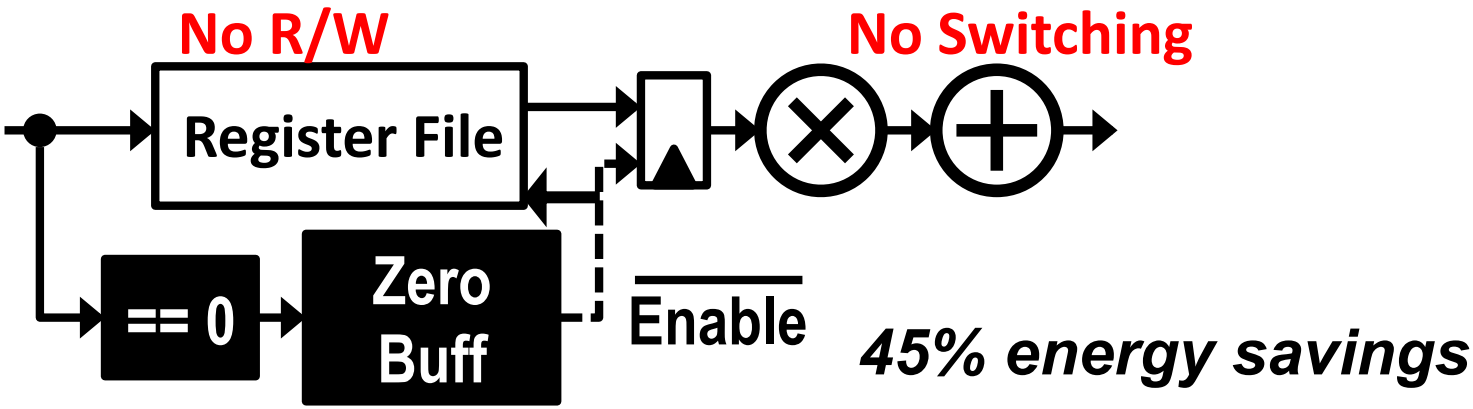
Many **zeros** in **output fmaps** after **ReLU**



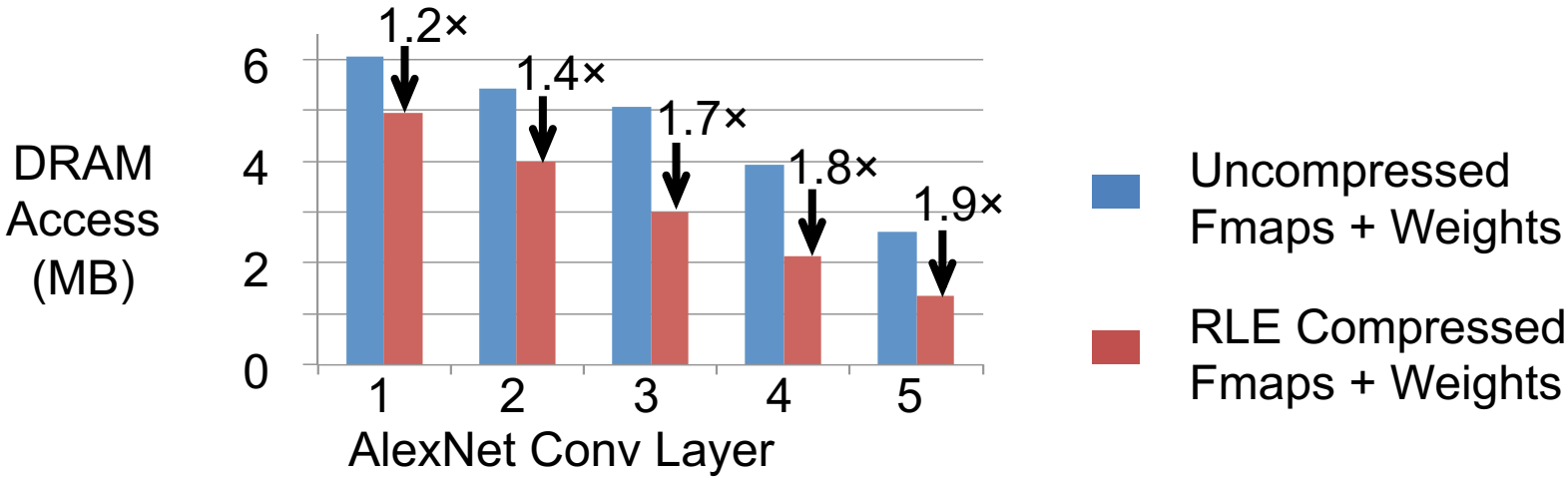


Exploit Sparsity

Method 1: Skip memory access and computation



Method 2: Compress data to reduce storage and data movement

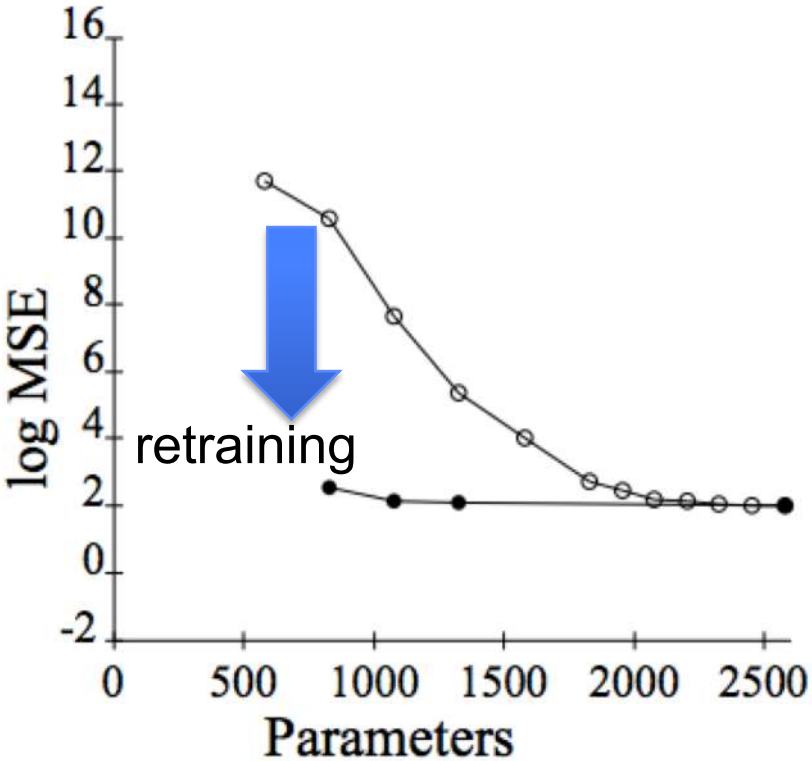


[Chen et al., ISSCC 2016]

Pruning – Make Weights Sparse

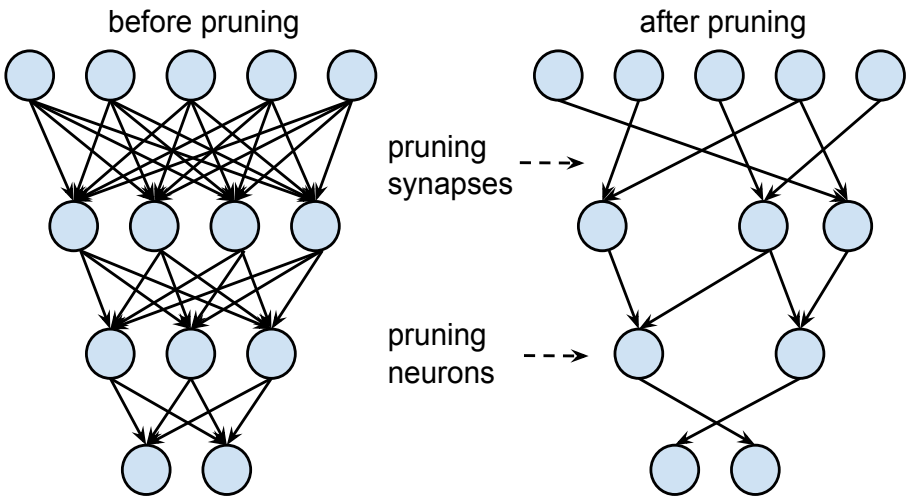
Optimal Brain Damage

[Lecun et al., NeurIPS 1989]



Prune DNN based on ***magnitude*** of weights

[Han et al., NeurIPS 2015]



Example: *AlexNet*

Weight Reduction:

CONV layers 2.7x, ***FC layers 9.9x***

Overall Reduction:

Weights 9x, MACs 3x

Pruning – Make Weights Sparse

Remove the weights with the **smallest joint impact** on the output feature map instead of that with the **smallest magnitude**

Magnitude-Based Method

fmap

1	1	1
1	1	1
1	1	1

*

filter

-8	1	1
1	1	1
1	1	1

Prune →

-8	0	0
0	0	0
0	1	1

= 0 → -6

Error: 6

Feature-Map-Based Method

fmap

1	1	1
1	1	1
1	1	1

*

filter

-8	1	1
1	1	1
1	1	1

Prune →

0	0	0
0	0	0
1	1	1

= 0 → 3

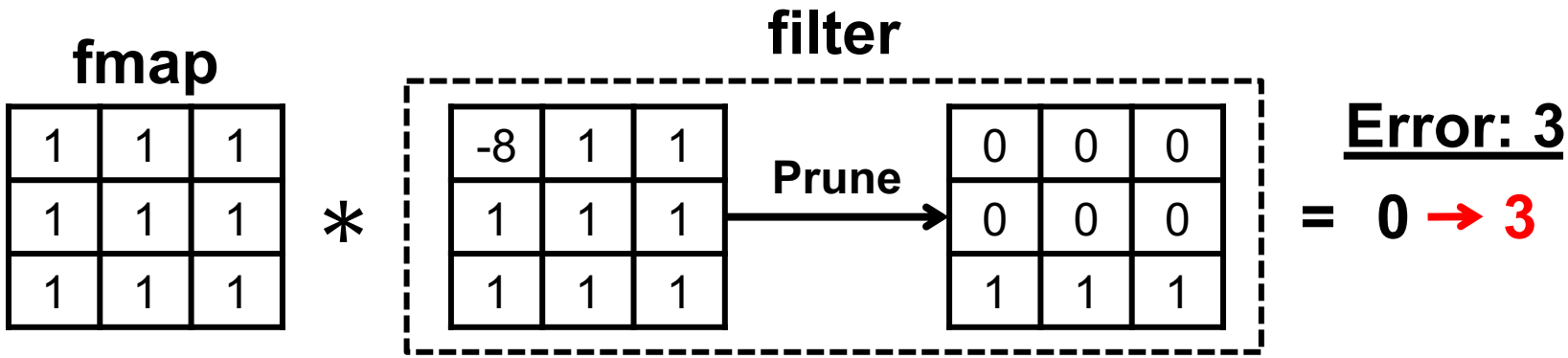
Error: 3

[Yang et al., Energy-Aware Pruning, CVPR 2017]

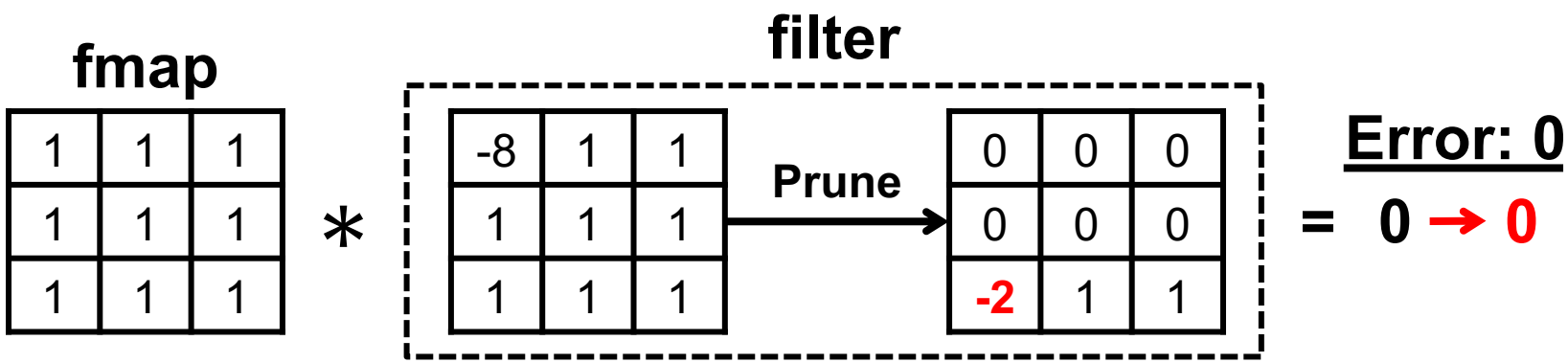
Fast Local Fine-Tuning

We then **locally fine-tune** the remaining weights, which is much faster than performing end-to-end training

After Pruning



After Local Fine-Tuning



[Yang et al., Energy-Aware Pruning, CVPR 2017]

Compression of Weights & Activations

- Compress weights and activations between DRAM and accelerator
- Variable Length / Huffman Coding

Example:

Value: **16'b0** → Compressed Code: {**1'b0**}

Value: **16'bx** → Compressed Code: {**1'b1**, **16'bx**}

- Tested on AlexNet → 2× overall BW Reduction

Layer	Filter / Image bits (0%)	Filter / Image BW Reduc.	IO / HuffIO (MB/frame)	Voltage (V)	MMACs/ Frame	Power (mW)	Real (TOPS/W)
General CNN	16 (0%) / 16 (0%)	1.0x		1.1	—	**288**	**0.3**
AlexNet l1	7 (21%) / 4 (29%)	1.17x / 1.3x	1 / 0.77	0.85	105	85	0.96
AlexNet l2	7 (19%) / 7 (89%)	1.15x / **5.8x**	3.2 / 1.1	0.9	224	55	1.4
AlexNet l3	8 (11%) / 9 (82%)	1.05x / 4.1x	6.5 / 2.8	0.92	150	77	0.7
AlexNet l4	9 (04%) / 8 (72%)	1.00x / 2.9x	5.4 / 3.2	0.92	112	95	0.56
AlexNet l5	9 (04%) / 8 (72%)	1.00x / 2.9x	3.7 / 2.1	0.92	75	95	0.56
Total / avg.	—	—	19.8 / **10**	—	—	**76**	**0.94**
LeNet-5 l1	3 (35%) / 1 (87%)	1.40x / 5.2x	0.003 / 0.001	0.7	0.3	25	1.07
LeNet-5 l2	4 (26%) / 6 (55%)	1.25x / 1.9x	0.050 / 0.042	0.8	1.6	35	1.75
Total / avg.	—	—	0.053 / **0.043**	—	—	**33**	**1.6**

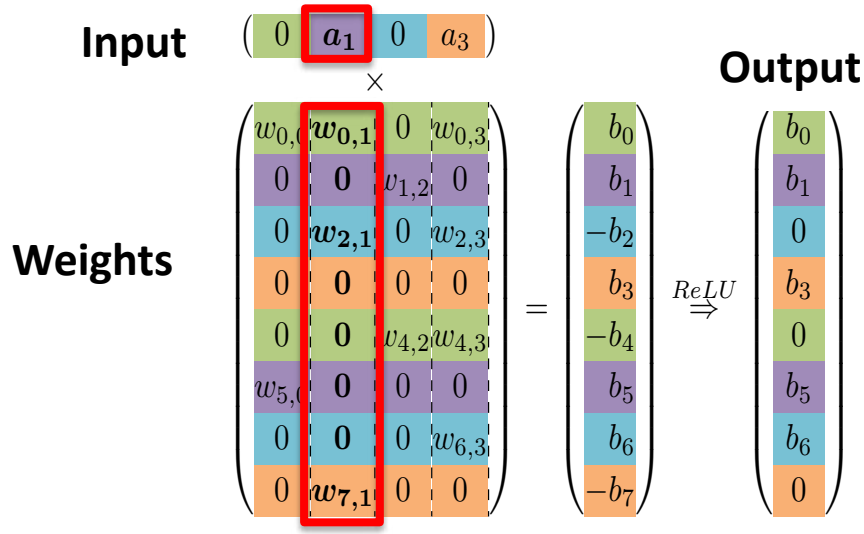
[Moons et al., VLSI 2016; Han et al., ICLR 2016]

Sparse Hardware

EIE

[Han et al., ISCA 2016]

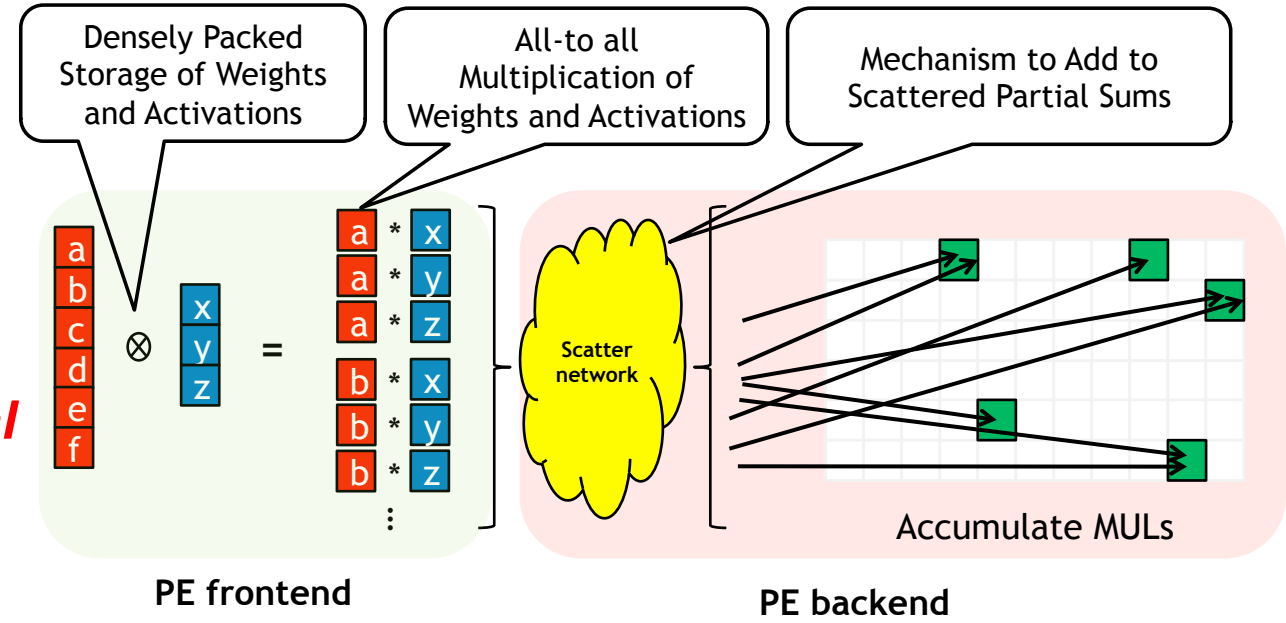
Supports Fully Connected Layers Only



SCNN

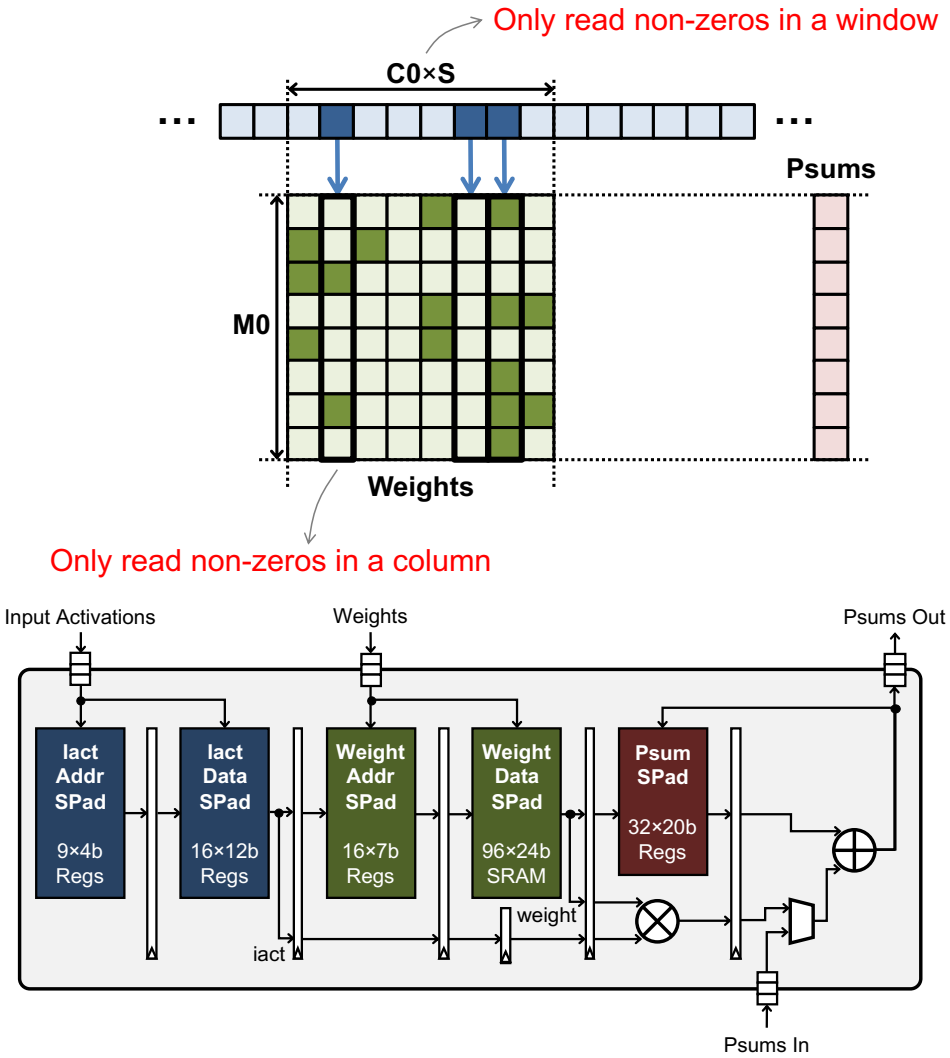
[Parashar et al., ISCA 2017]

Supports Convolutional Layers Only



Sparse Hardware – Eyeriss v2

Supports both Convolutional and Fully Connected Layers

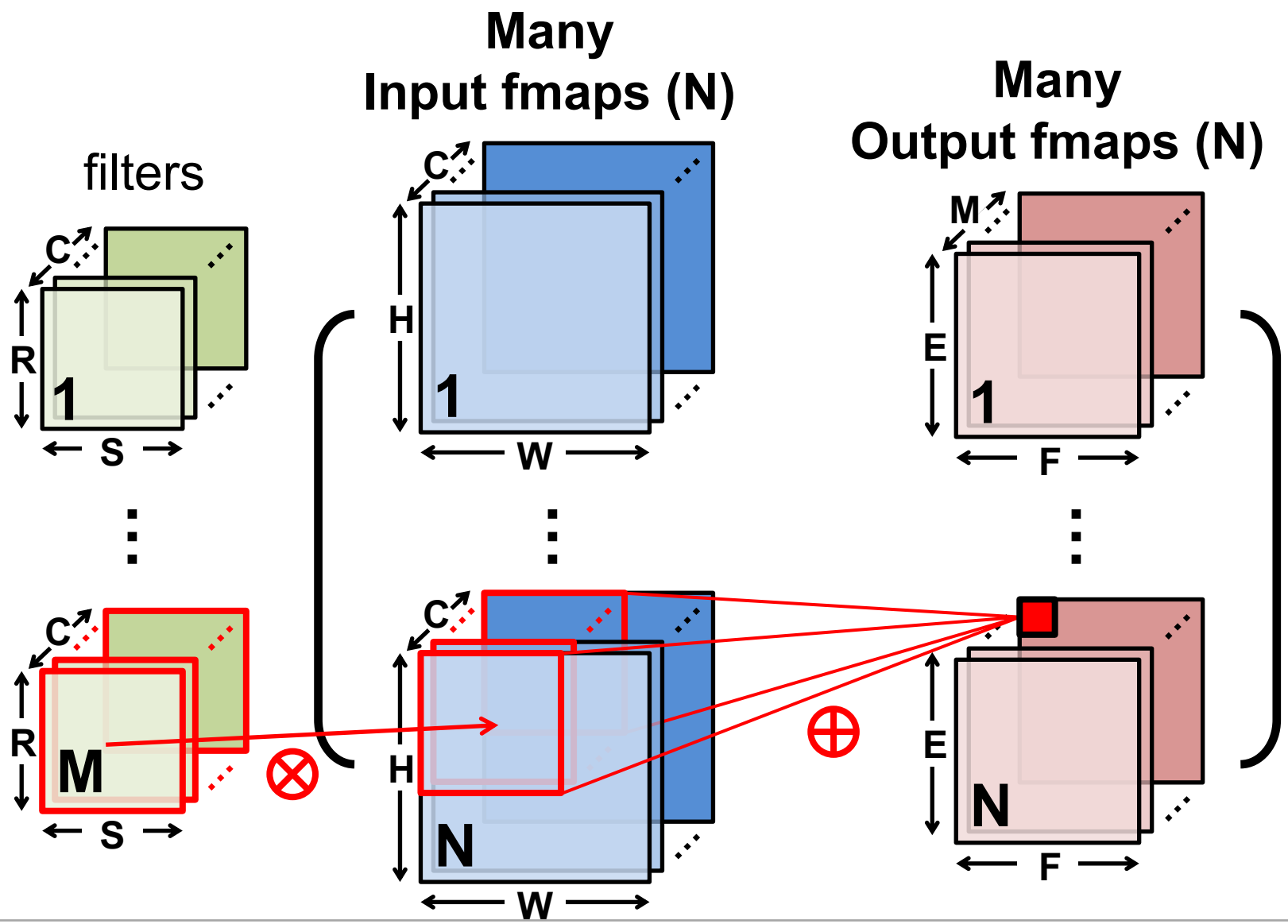


	AlexNet	sparse-AlexNet
GOPS	148.3	405.8
fps	102.4	280.1
Over v1	15.5×	42.5×
GOPS/W	277.9	1028.1
Inferences/J	191.8	709.7
Over v1	3.0×	11.3×

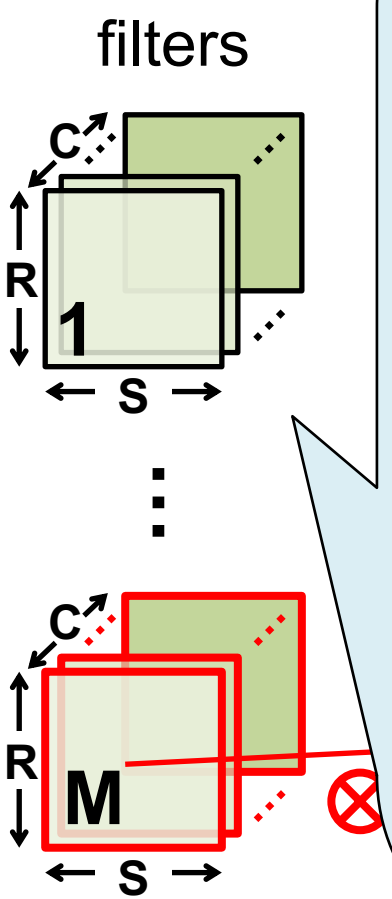
[Chen et al., JETCAS 2019]

Manual Network Architecture Design

Simplify CONV Layers



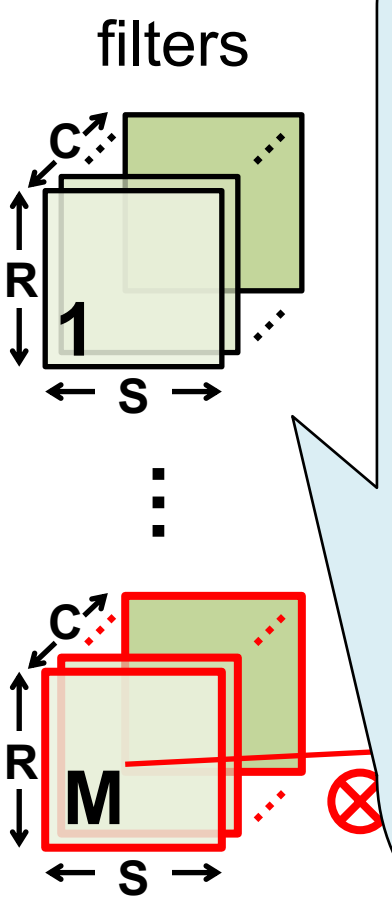
Simplify CONV Layers



Methods can be roughly categorized by how the filters are simplified:

- Reduce spatial size (R, S): stacked filters
- Reduce channels (C): 1x1 convolution, group of filters
- Reduce filters (M): feature map reuse

Simplify CONV Layers

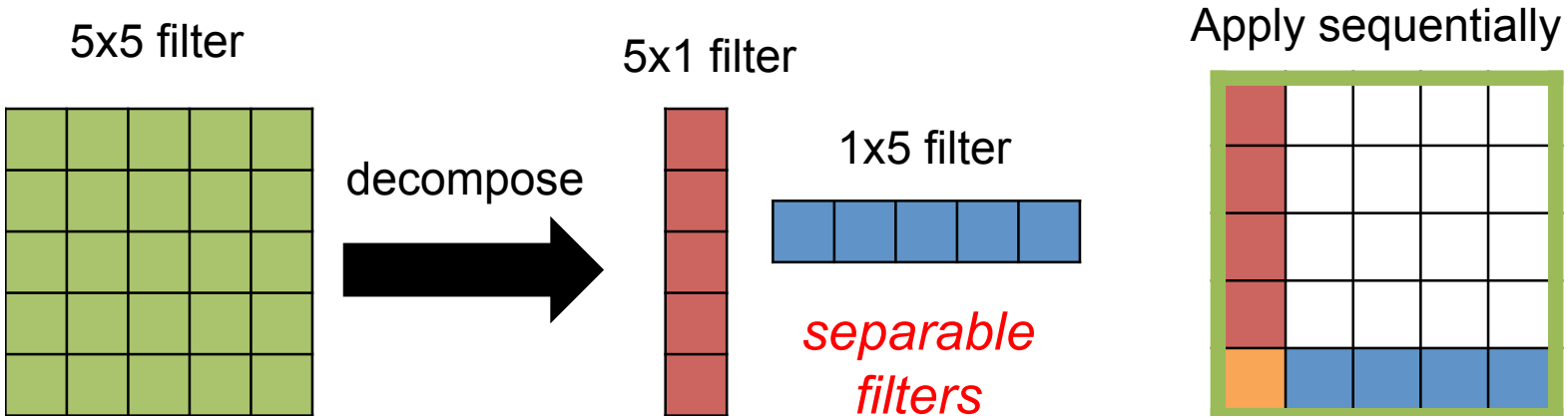


Methods can be roughly categorized by how the filters are simplified:

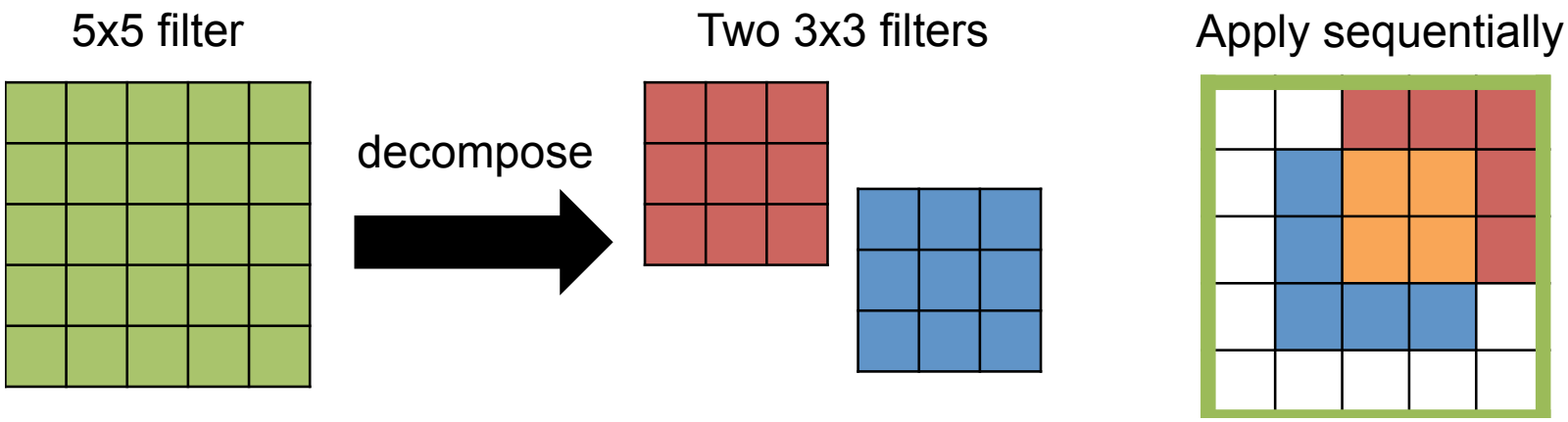
- Reduce spatial size (R, S): stacked filters
- Reduce channels (C): 1x1 convolution, group of filters
- Reduce filters (M): feature map reuse

Stacked Filters

GoogleNet/Inception v3



VGG-16

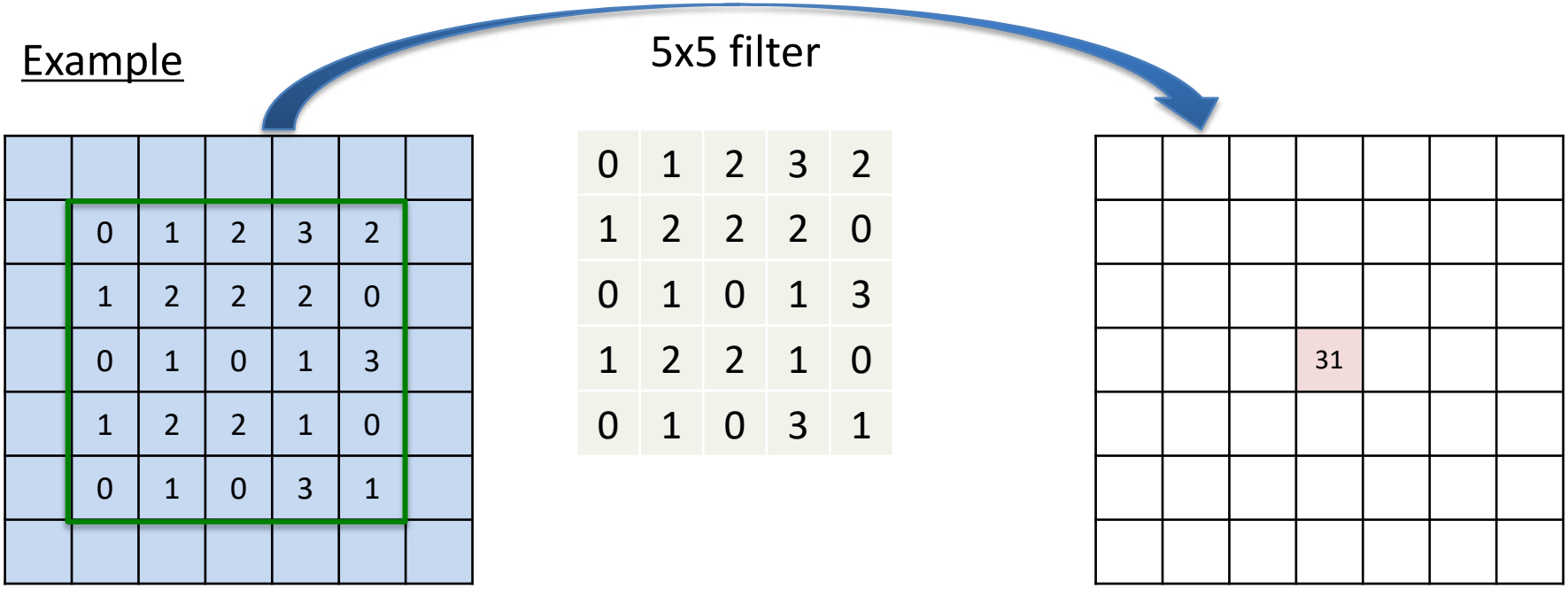


Replace a large filter with a series of smaller filters

Stacked Filters

- Use stack of smaller filters (3x3) to cover the same receptive field with fewer filter weights

Example



Stacked Filters

- Use stack of smaller filters (3x3) to cover the same receptive field with fewer filter weights

filter (3x3)

0	1	0
1	1	1
0	1	0

Example

0	0	0	0	0	0	0
0	0	1	2	3	2	0
0	1	2	2	2	0	0
0	0	1	0	1	3	0
0	1	2	2	1	0	0
0	0	1	0	3	1	0
0	0	0	0	0	0	0

		7	8	8		
		5	6	7		
		6	5	7		

3x3 filter$_1$

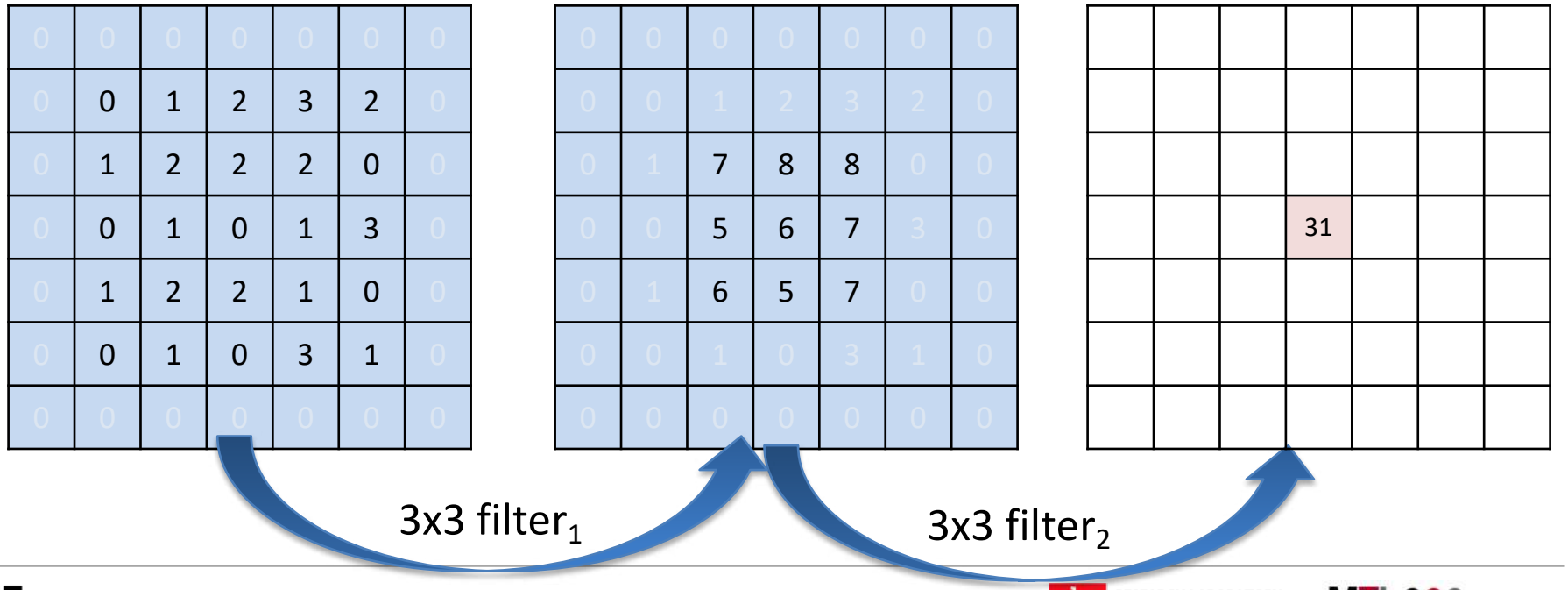
Stacked Filters

- Use stack of smaller filters (3x3) to cover the same receptive field with fewer filter weights

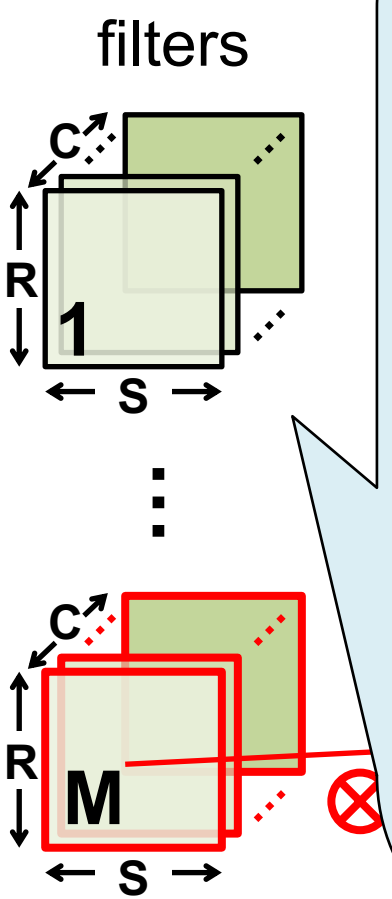
filter (3x3)

0	1	0
1	1	1
0	1	0

<u>Example</u>: 5x5 filter (25 weights) → two 3x3 filters (18 weights)



Simplify CONV Layers

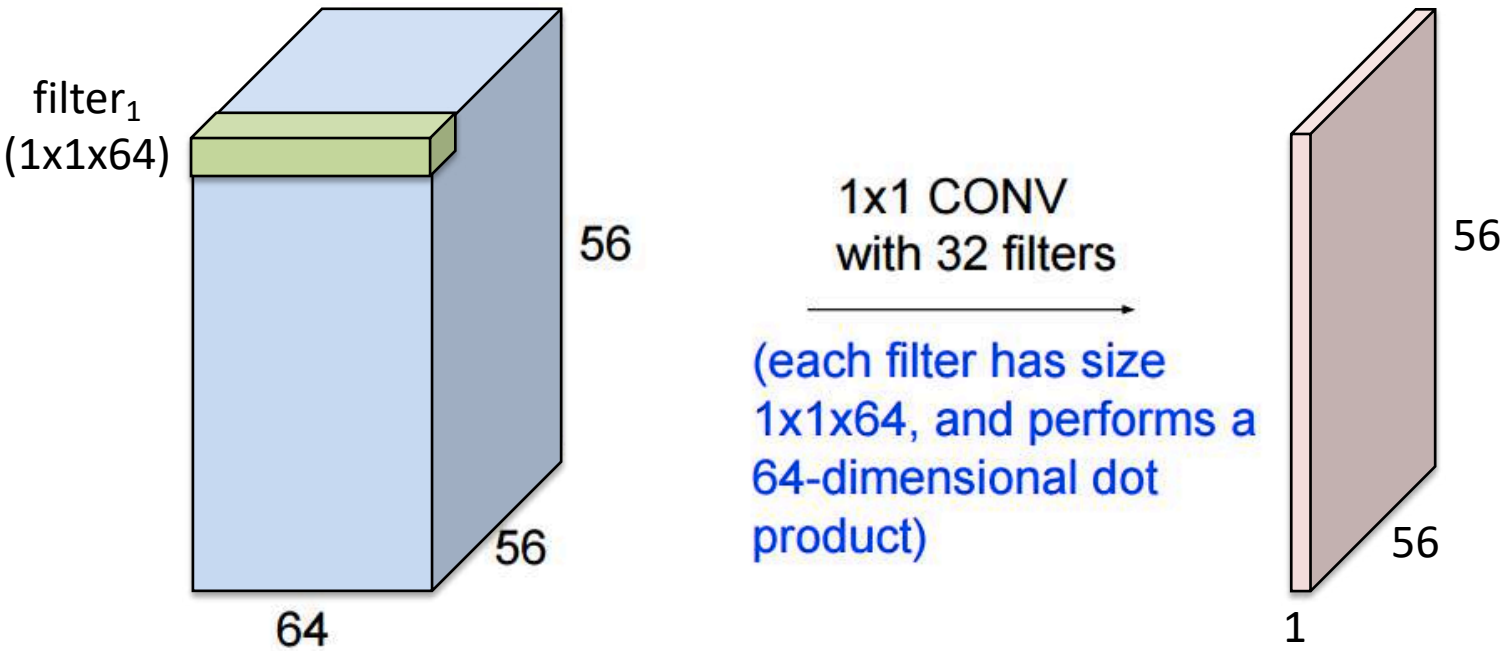


Methods can be roughly categorized by how the filters are simplified:

- Reduce spatial size (R, S): stacked filters
- Reduce channels (C): 1x1 convolution, group of filters
- Reduce filters (M): feature map reuse

1x1 Convolution

Use **1x1 filter** to condense the cross-channel information.

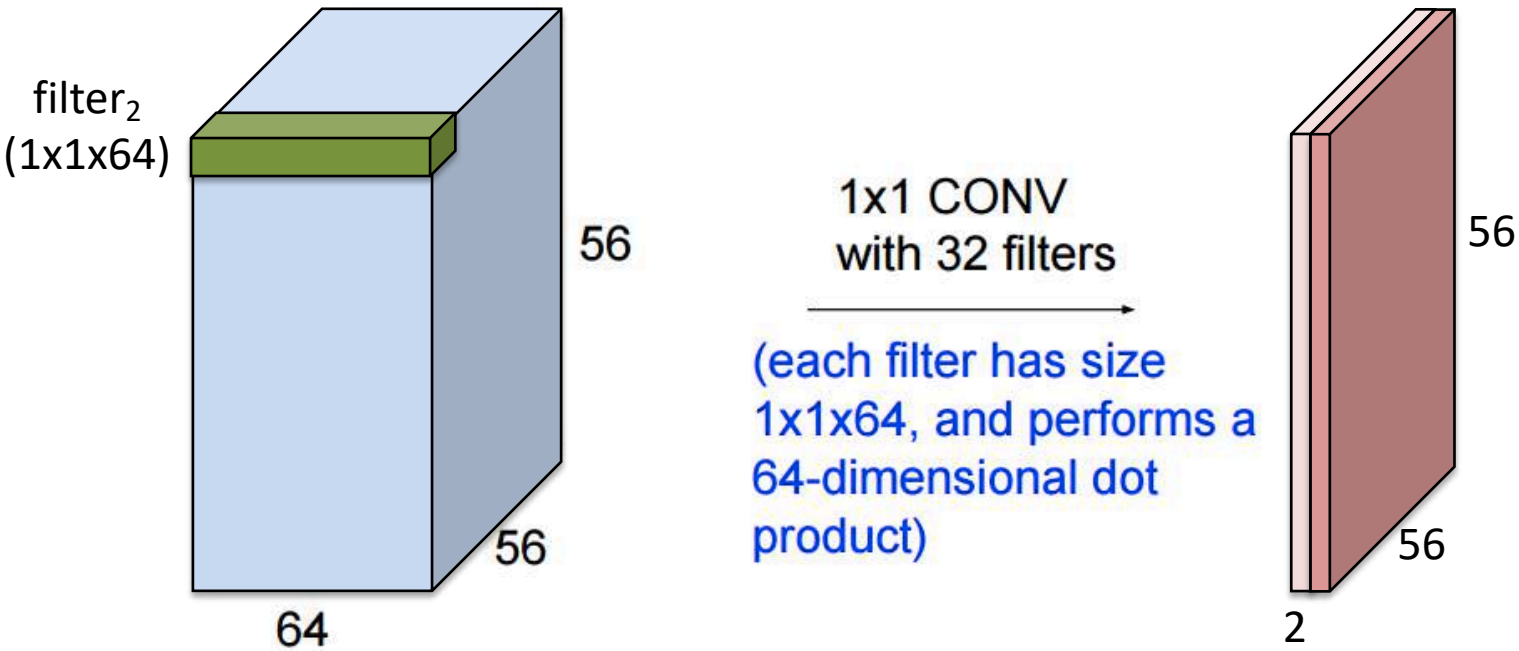


Modified image from source: Stanford cs231n

[Lin et al., Network in Network, arXiv 2013, ICLR 2014]

1x1 Convolution

Use **1x1 filter** to condense the cross-channel information.

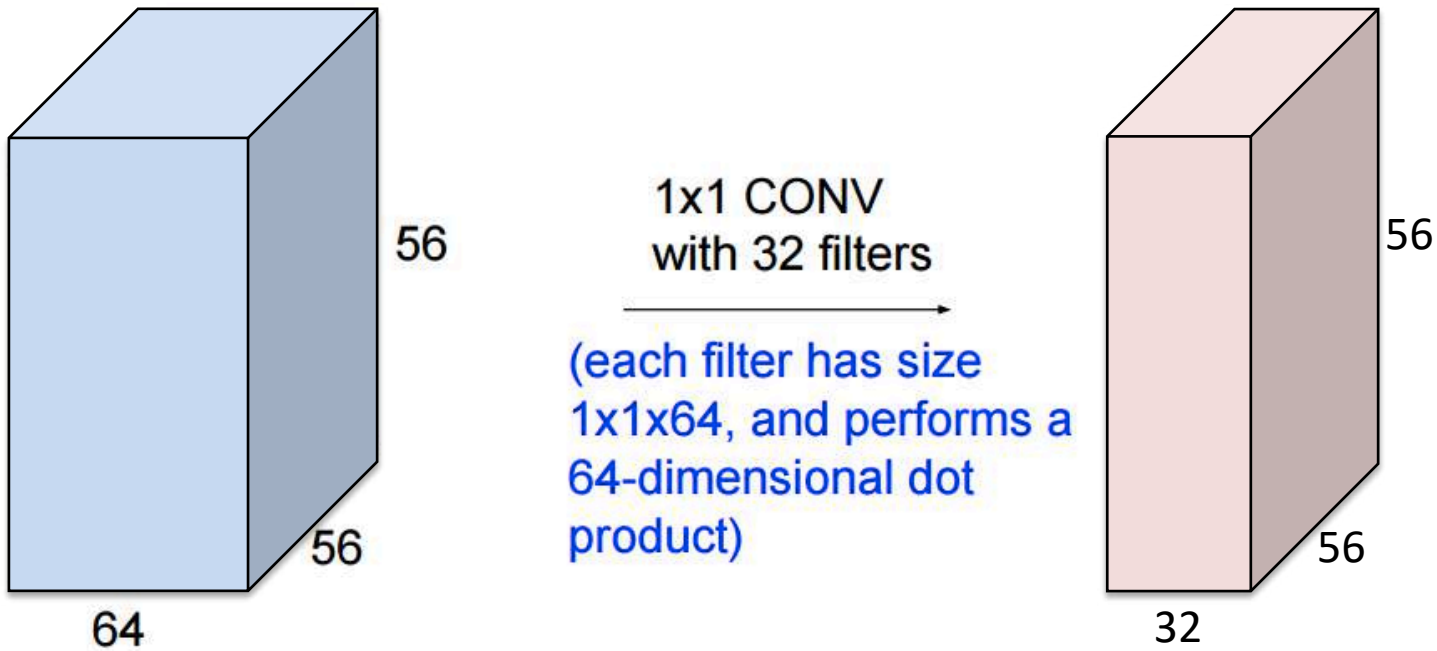


Modified image from source: Stanford cs231n

[Lin et al., Network in Network, arXiv 2013, ICLR 2014]

1x1 Convolution

Use **1x1 filter** to condense the cross-channel information.



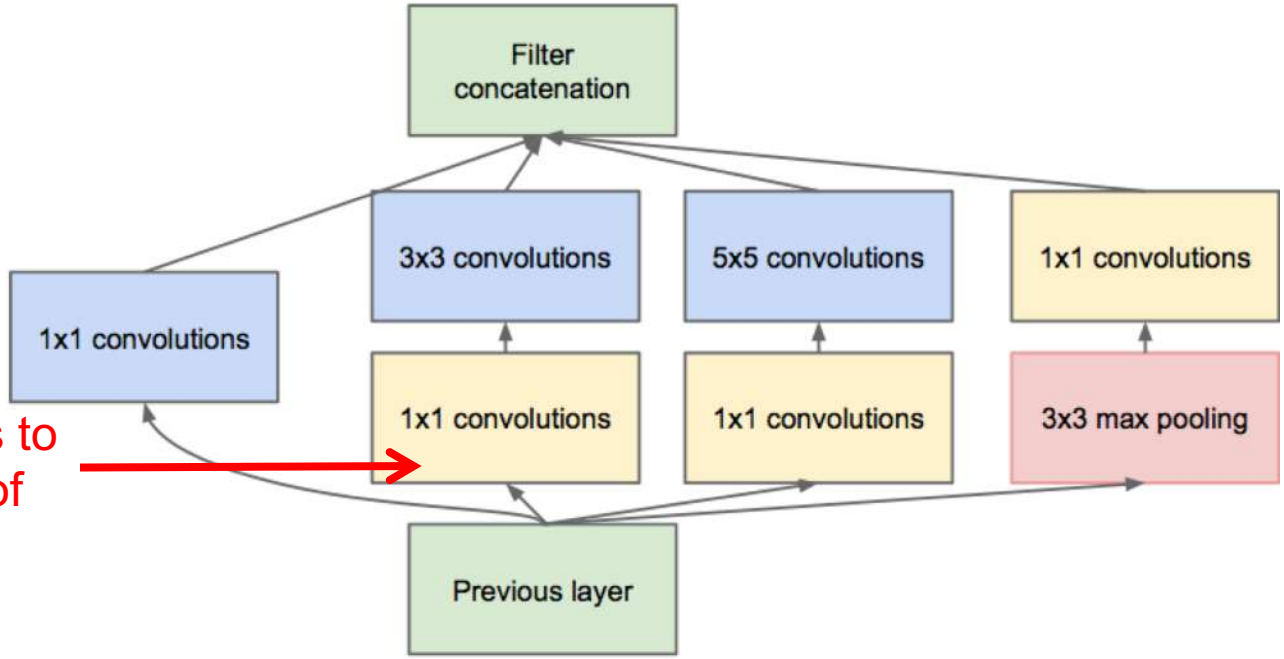
Modified image from source:
Stanford cs231n

[Lin et al., Network in Network, arXiv 2013, ICLR 2014]

GoogLeNet:1x1 Convolution

Apply 1x1 convolution before 'large' convolution filters.
Reduce weights such that **entire CNN can be trained on one GPU.**
Number of multiplications reduced from 854M → 358M

Inception Module



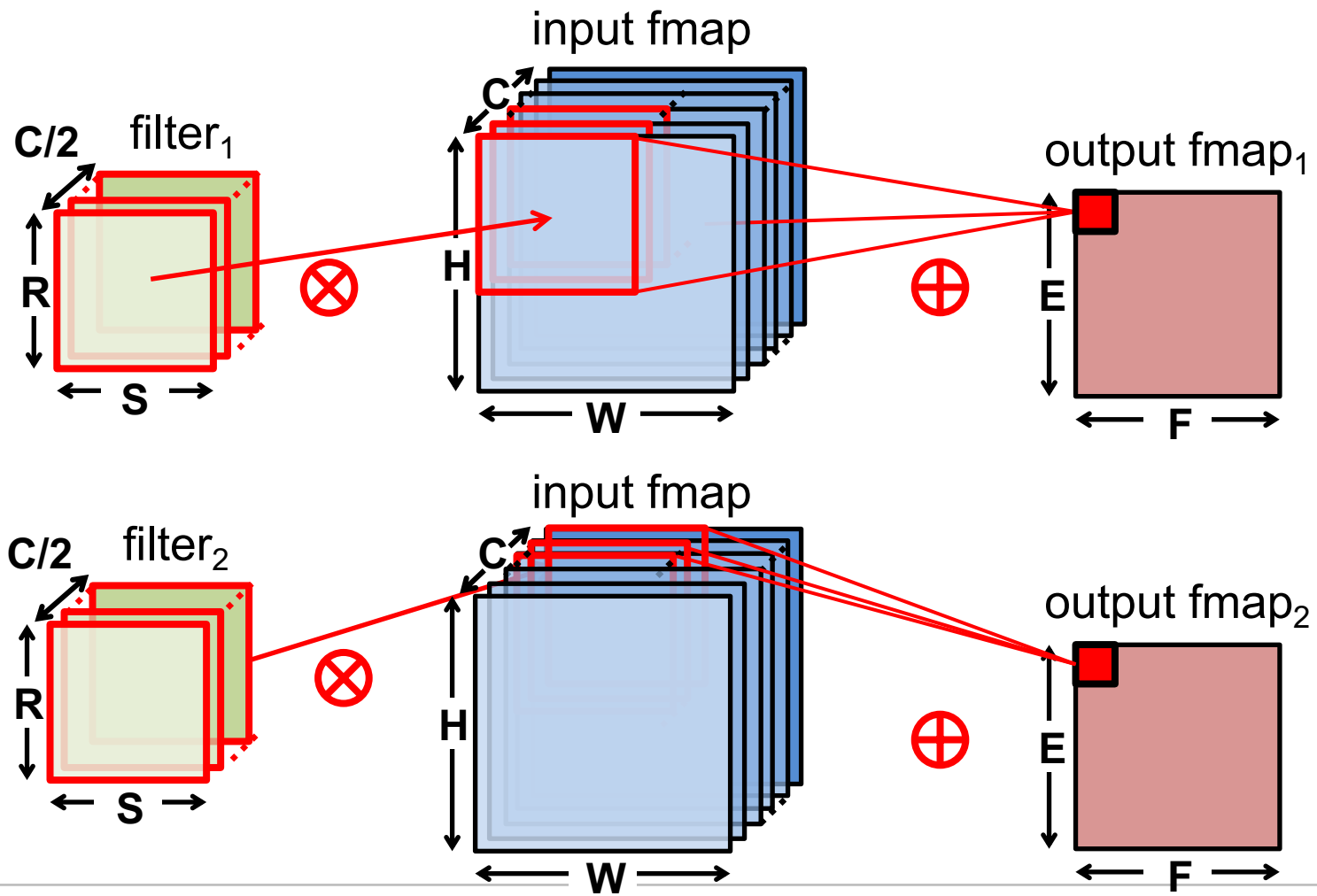
1x1 convolutions to reduce number of weights and multiplications

[Szegedy et al., arXiv 2014, CVPR 2015]

Group of Filters

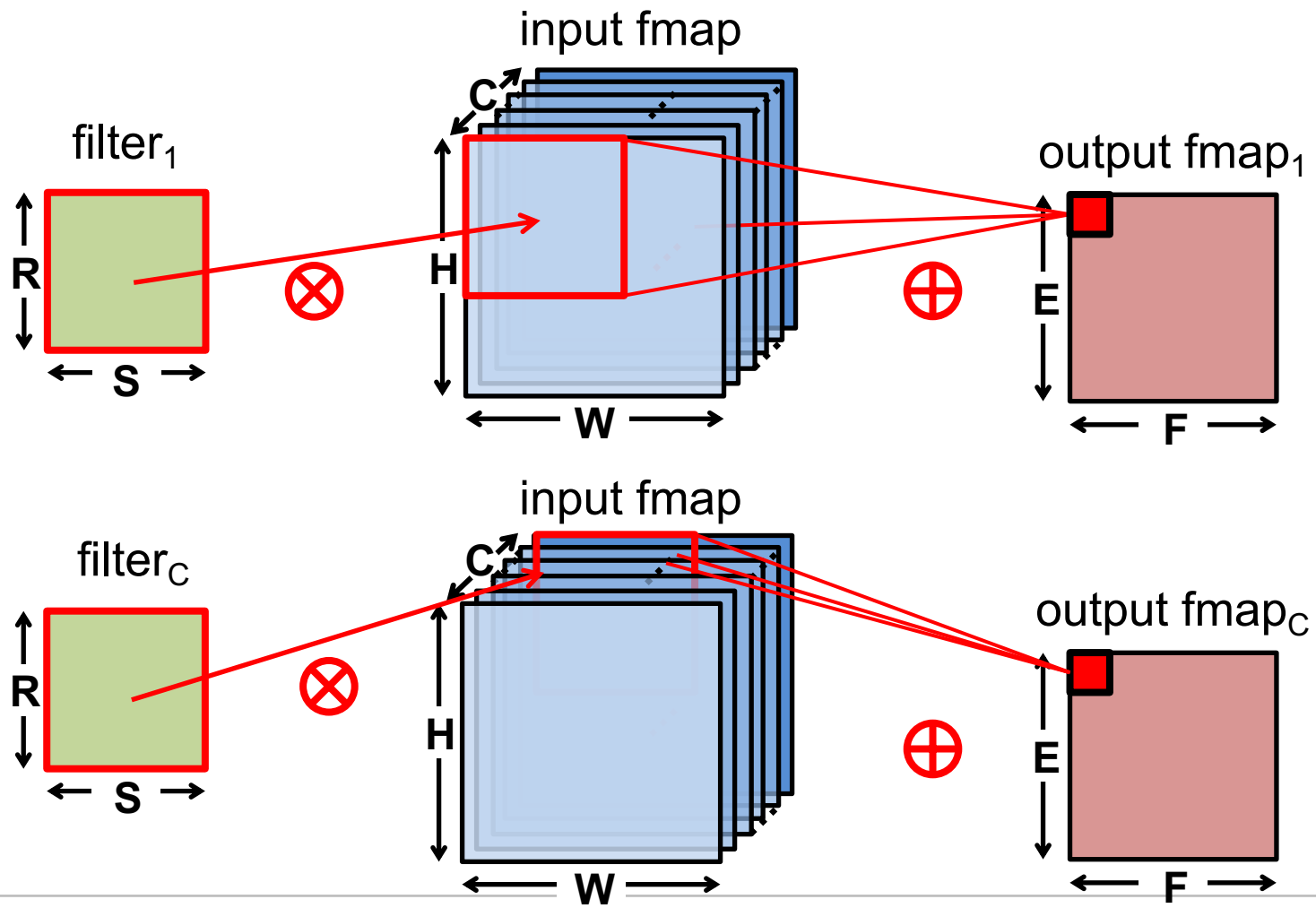
Idea: split filters and channels of feature map into different groups

Example: 2 groups, each filter requires **2x fewer weights and multiplications**.



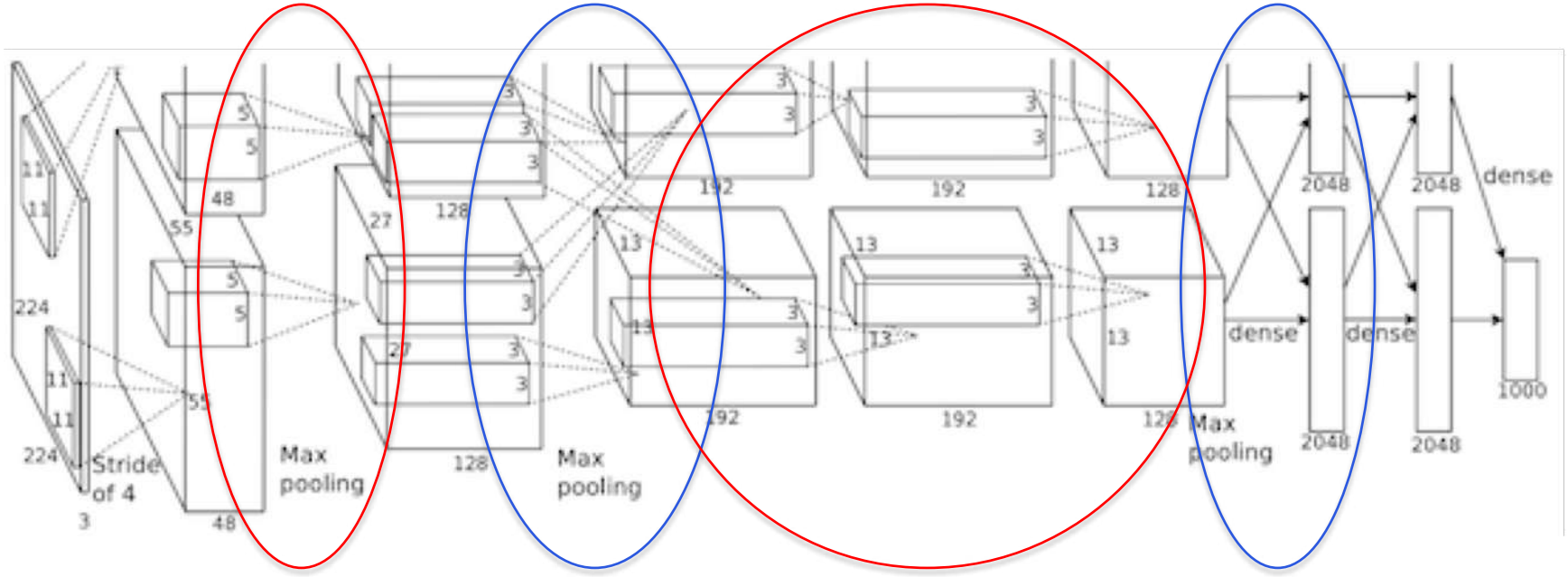
Group of Filters

The extreme case is **depthwise convolution** – each group contains only one channel.



Group of Filters

AlexNet uses group of filters to train on two separate GPUs
(Drawback: correlation between channels of different groups is not used)



Split into 2 Groups

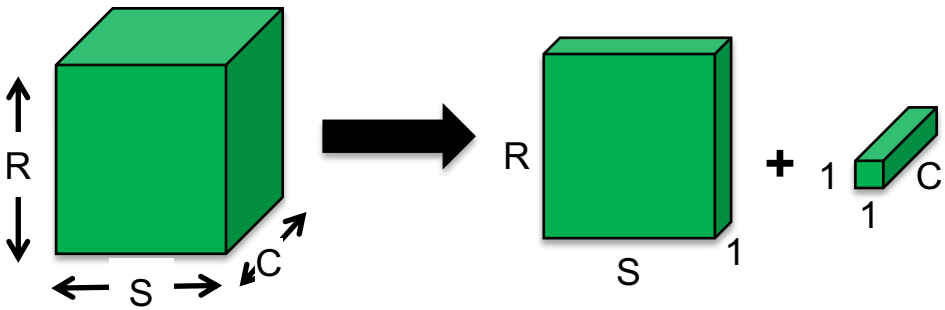
Mix Information (3x3 CONV)

Split into 2 Groups

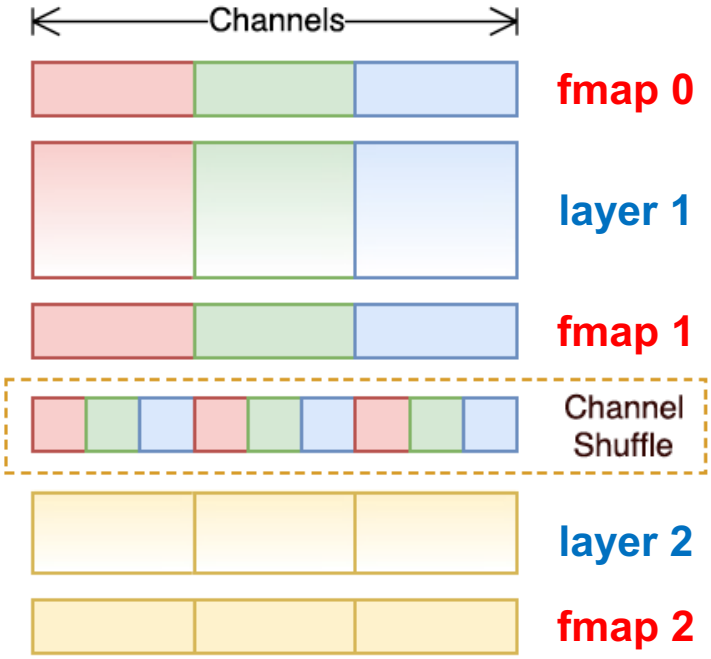
Mix Information (FC)

Group of Filters

Two ways of mixing information from groups



Pointwise (1x1) Convolution
(Mix in one step)
MobileNet



Shuffle Operation
(Mix in multiple steps)
ShuffleNet

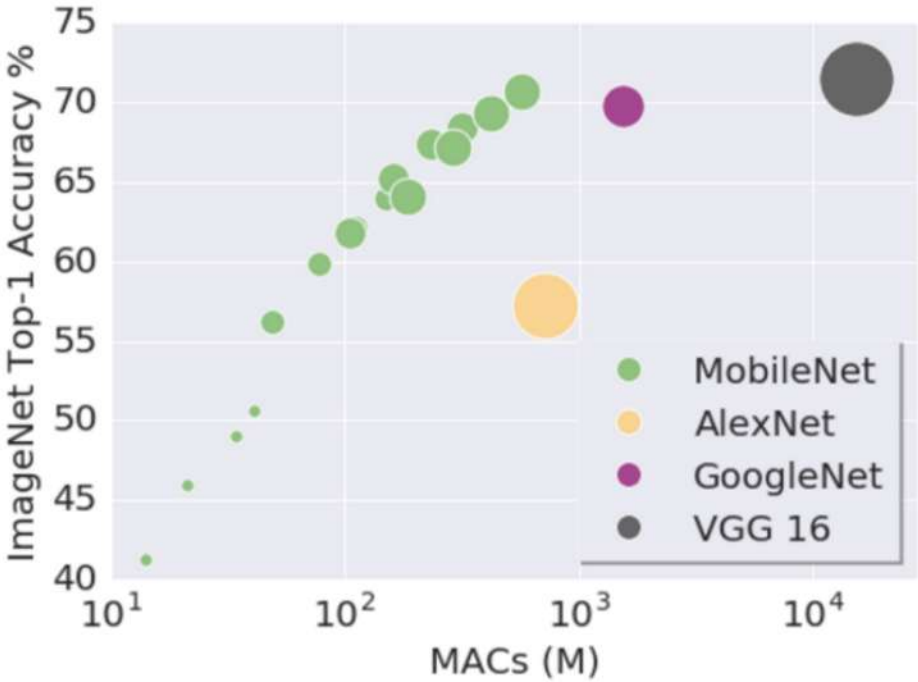
MobileNets: Comparison

Table 8. MobileNet Comparison to Popular Models

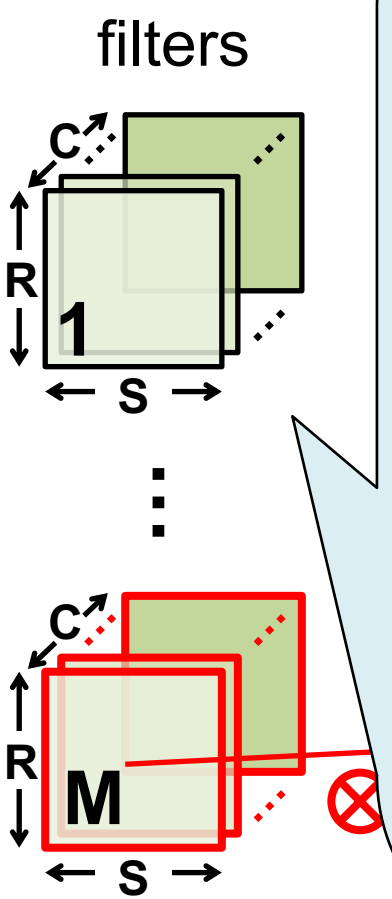
Model	ImageNet Accuracy	Million Mult-Adds	Million Parameter
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

Table 9. Smaller MobileNet Comparison to Popular Models

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameter
0.50 MobileNet-160	60.2%	76	1.32
Squeezenet	57.5%	1700	1.25
AlexNet	57.2%	720	60



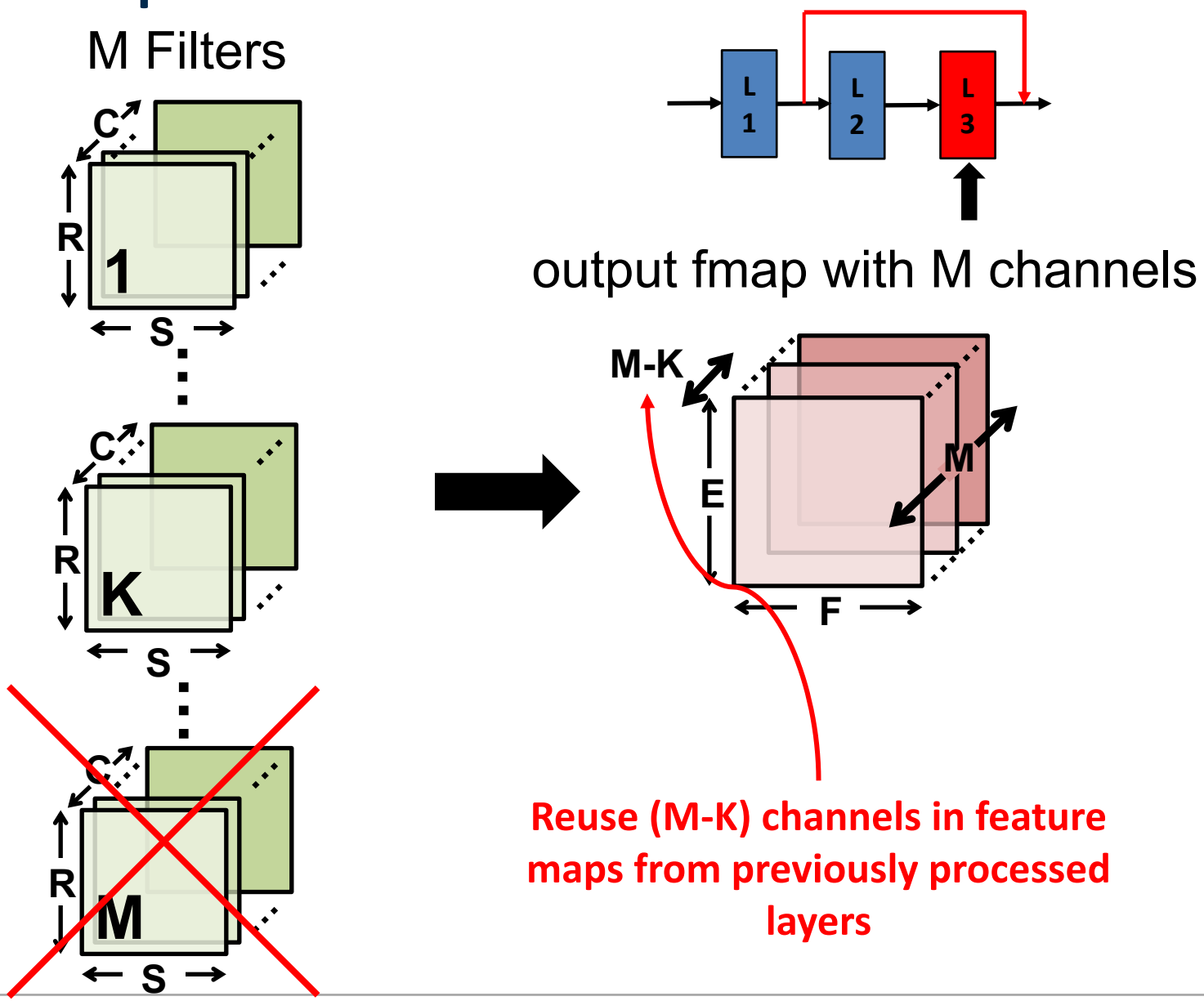
Simplify CONV Layers



Methods can be roughly categorized by how the filters are simplified:

- Reduce spatial size (R, S): stacked filters
- Reduce channels (C): 1x1 convolution, group of filters
- Reduce filters (M): feature map reuse

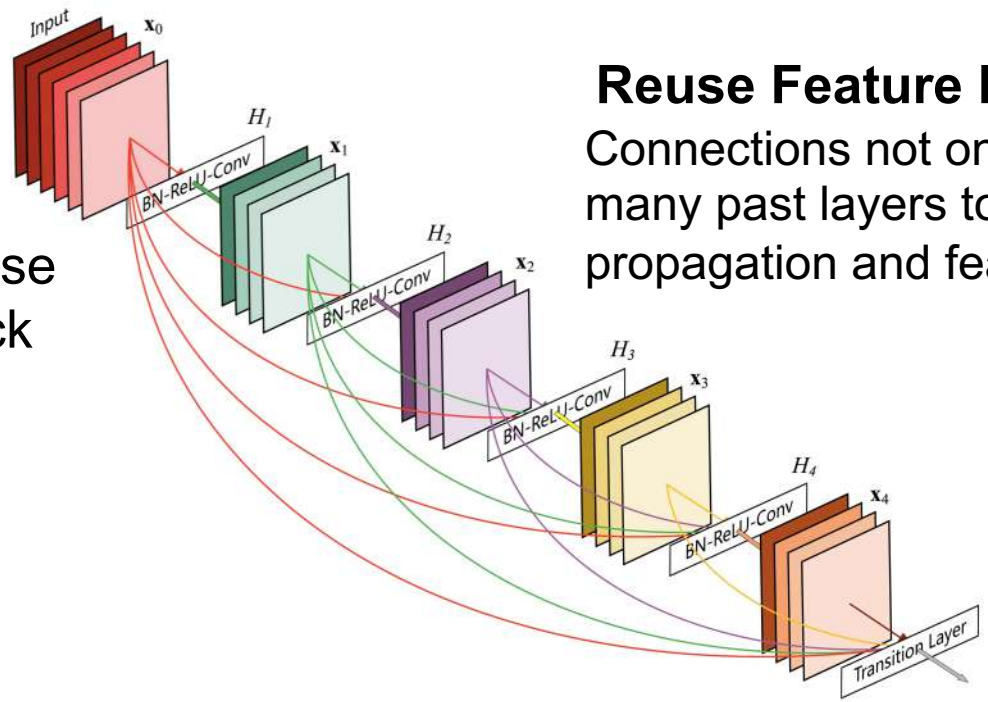
Feature Map Reuse



Reuse (M-K) channels in feature maps from previously processed layers

Feature Map Reuse

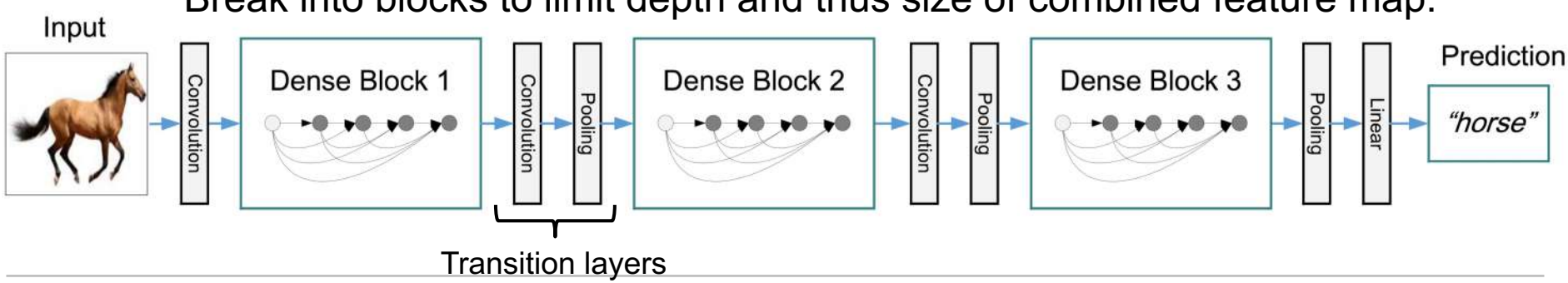
Dense Block



Reuse Feature Maps from Multi. Layers!

Connections not only from previous layer, but many past layers to strengthen feature map propagation and feature reuse.

Feature maps are concatenated rather than added.
Break into blocks to limit depth and thus size of combined feature map.

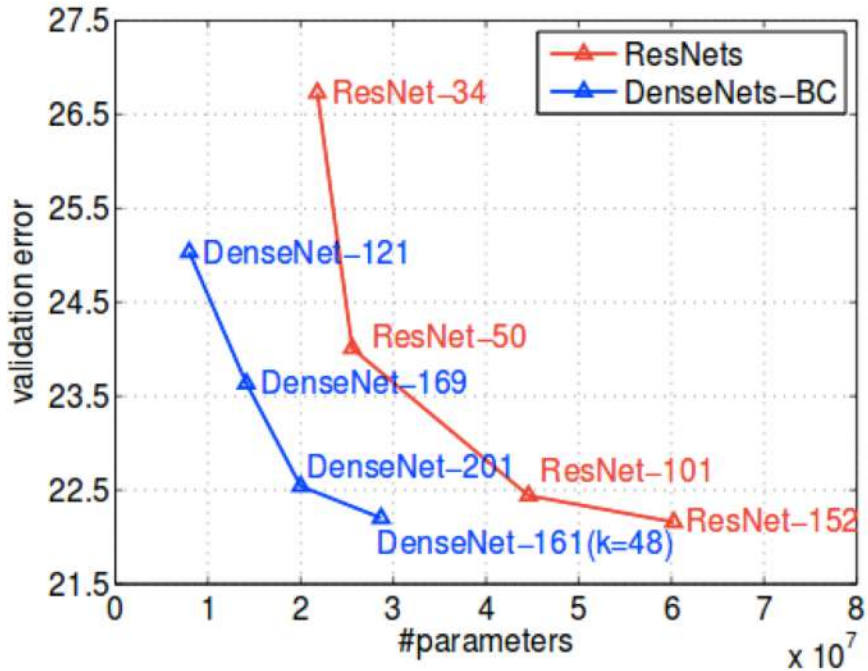


[Huang et al., CVPR 2017]

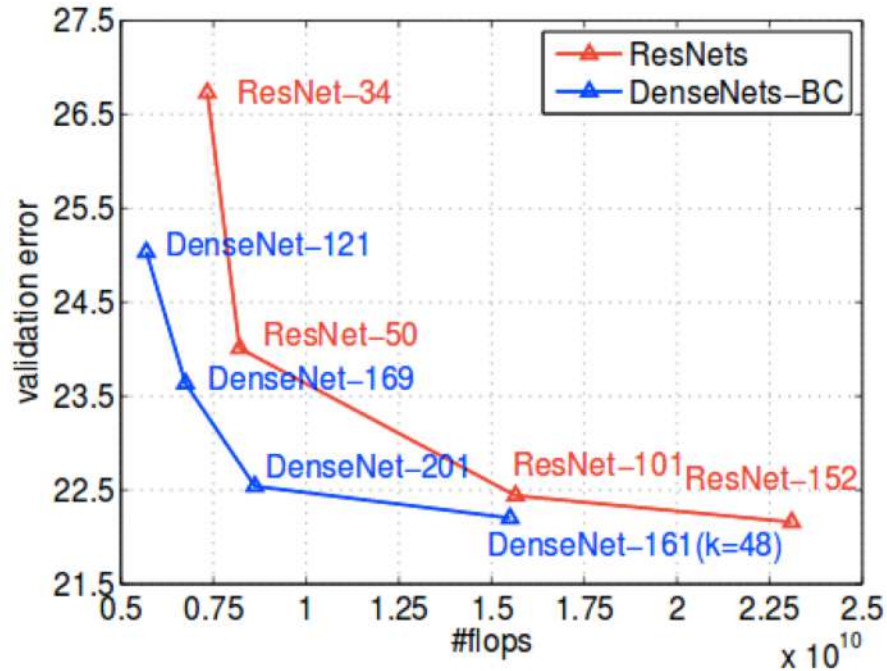
DenseNet

Higher accuracy than ResNet with fewer weights and multiplications

Top-1 error



Top-1 error

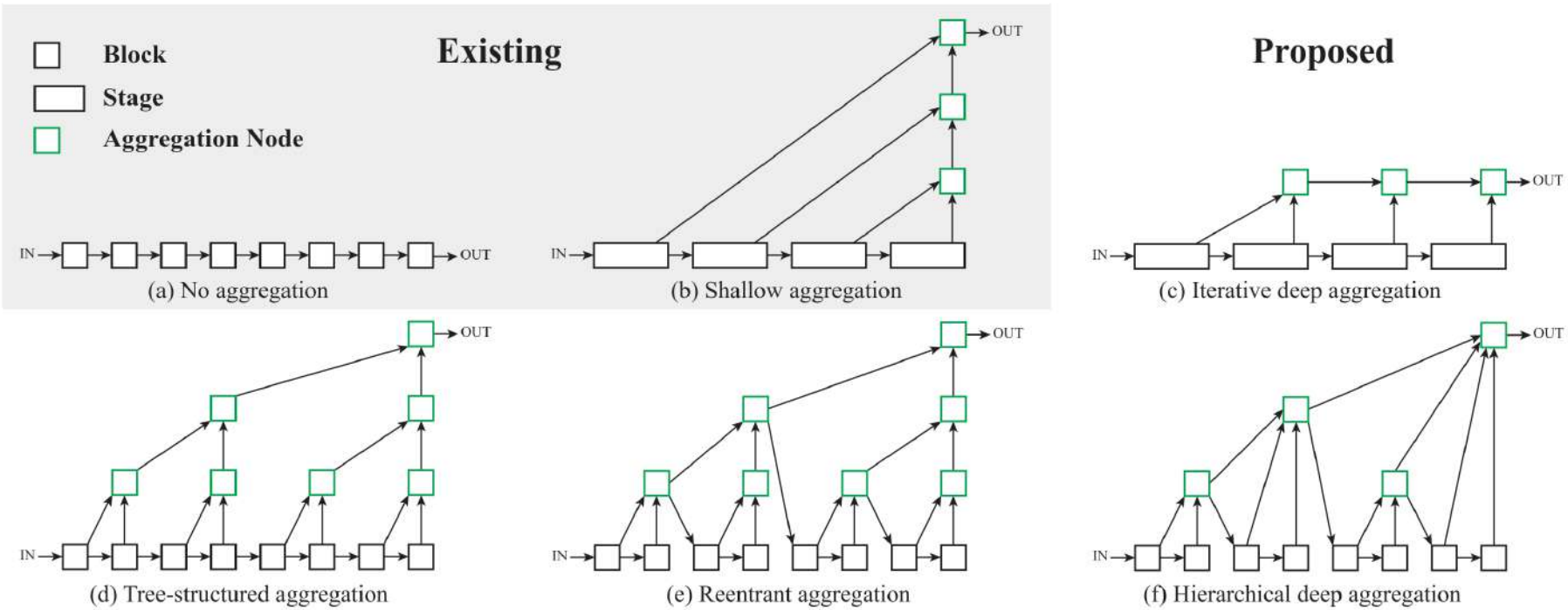


Note: 1 MAC = 2 FLOPS

[Huang et al., CVPR 2017]

Feature Map Reuse

- **More complicated layer aggregation**



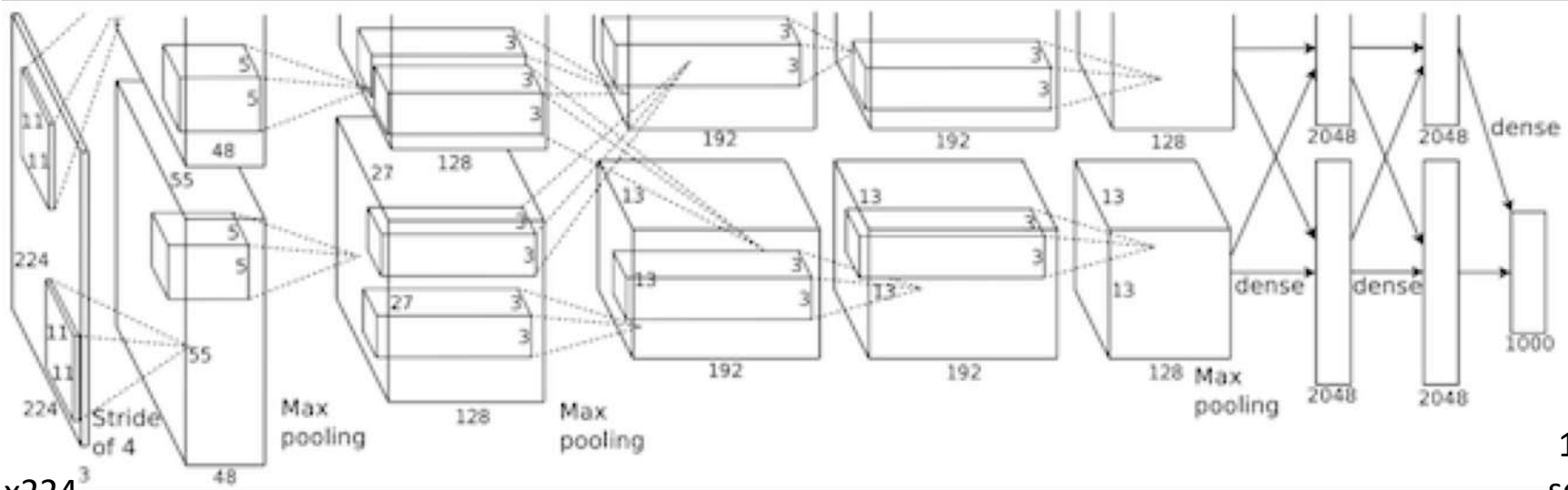
[Yu et al., CVPR 2018]

Simplify FC Layers

CONV Layers: 5
Fully Connected Layers: 3
Weights: 61M
MACs: 724M

ILSCVR12 Winner

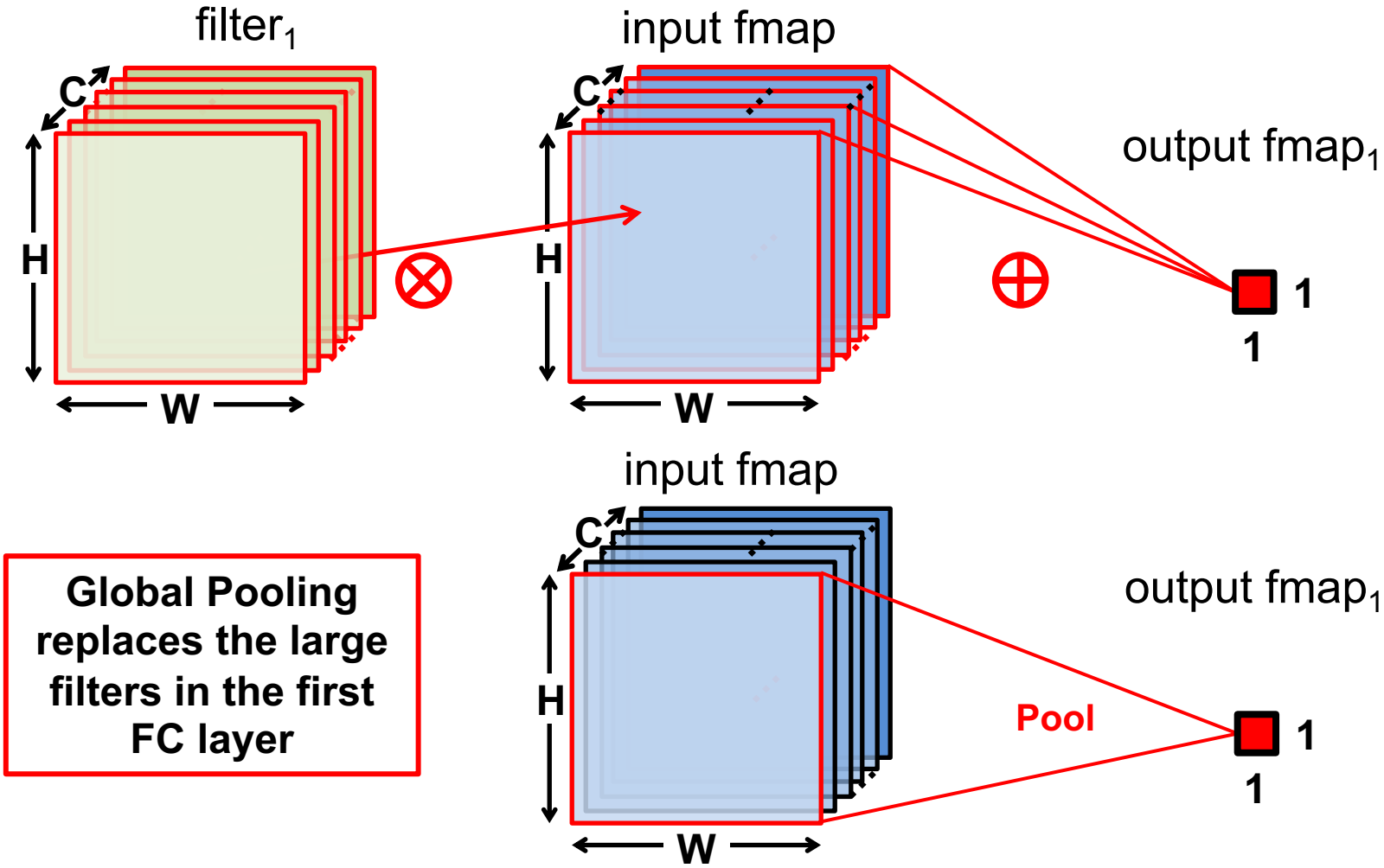
[Krizhevsky et al., NIPS 2012]



224x224 Input Image → Conv (11x11) / Non-Linearity / Norm (LRN) / Max Pool → L1 → Conv (5x5) / Non-Linearity / Norm (LRN) / Max Pooling → L2 → Conv (3x3) / Non-Linearity → L3 → Conv (3x3) / Non-Linearity → L4 → Conv (3x3) / Non-Linearity / Max Pooling → L5 → Fully Connect / Non-Linearity → L6 → Fully Connect / Non-Linearity → L7 → Fully Connect → 1000 scores

	L1	L2	L3	L4	L5	L6	L7
# of weights	34k	307k	885k	664k	442k	38M	16.8M	4.1M
# of MACs	105M	224M	150M	112M	75M	38M	17M	4M

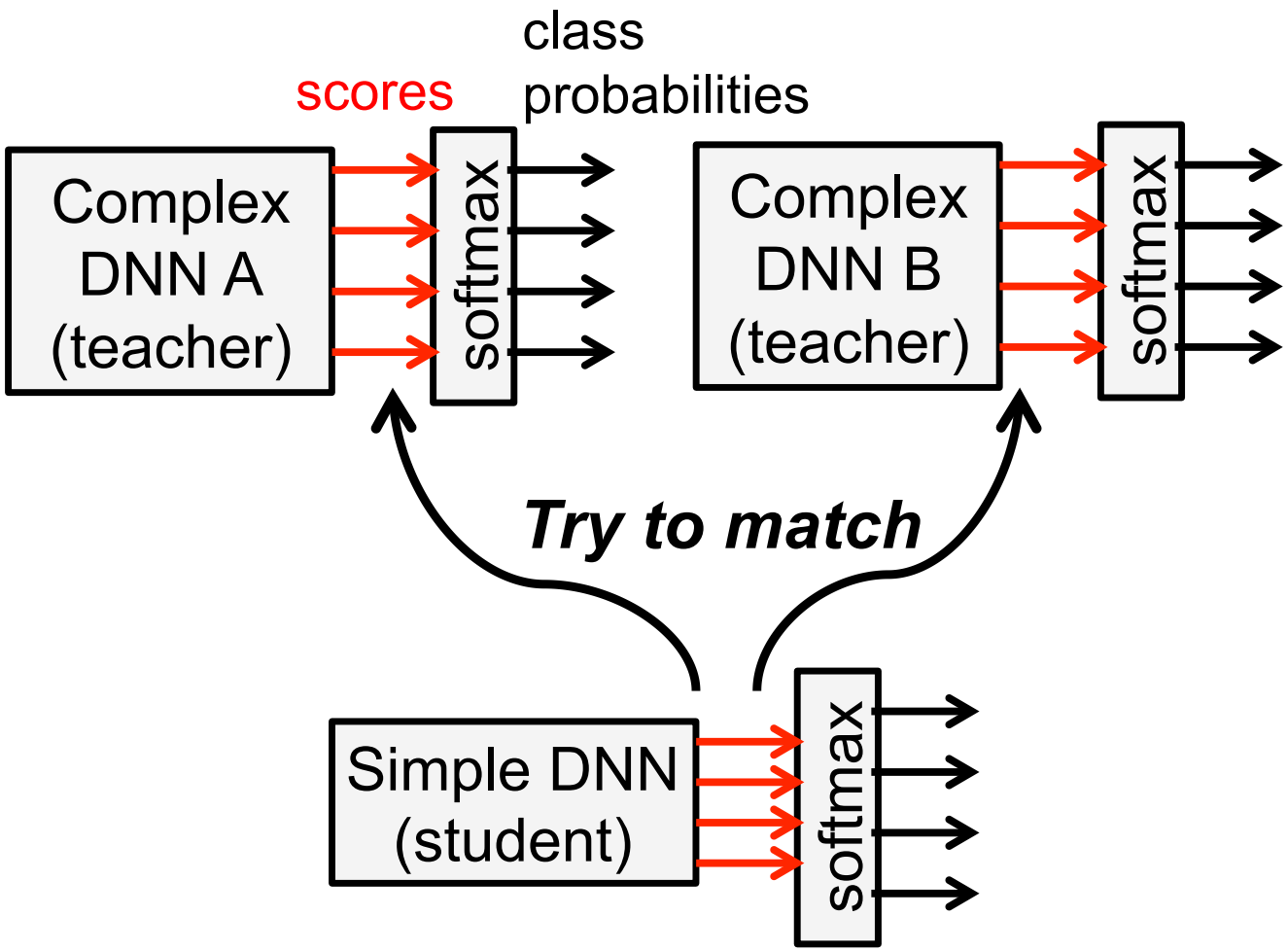
Simplify FC Layers



Global Pooling replaces the large filters in the first FC layer

[Lin et al., ICLR 2014]

Knowledge Distillation

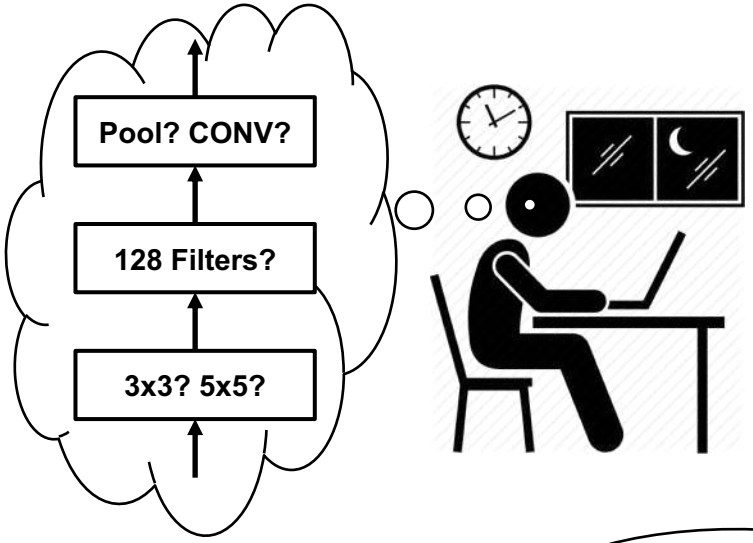


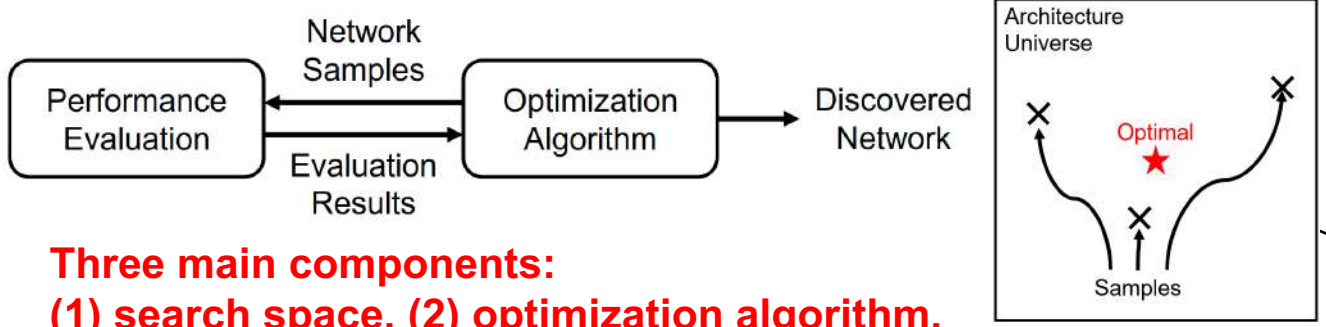
[Bucilu et al., KDD 2006],[Hinton et al., arXiv 2015]

Network Architecture Search (NAS)

Learn Network Architecture

Rather than handcrafting the architecture, automatically search for it





Three main components:
(1) search space, (2) optimization algorithm, and (3) performance evaluation.

Evaluate NAS Performance

- **Key Metrics**
  - **Achievable DNN accuracy**
  - **Required search time**

$$time_{nas} = num_{samples} \times time_{per_sample}$$

$$time_{nas} \propto \left(\frac{num_{nas_tuning} \times size_{search_space}}{efficiency_{alg}}\right) \times (time_{train} + time_{eval})$$

(2) Improve the optimization algorithm

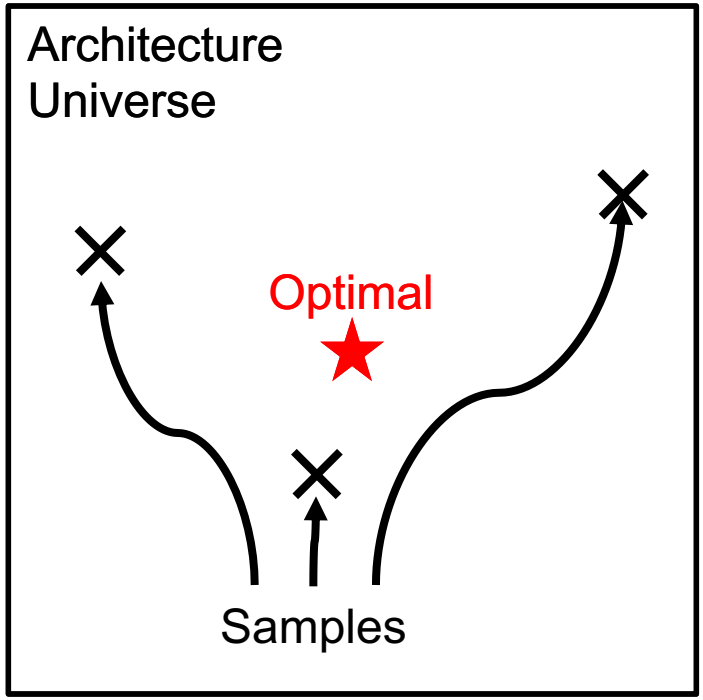
(1) Shrink the search space

(3) Simplify the performance evaluation

Researchers improve the efficiency of NAS in 3 main components

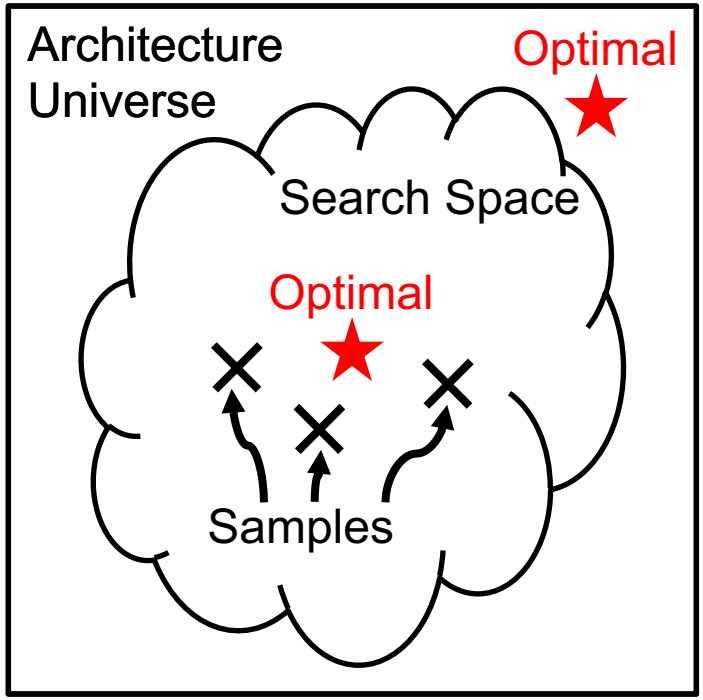
(1) Shrink the Search Space

- **Trade the discoverable architectures for search speed**



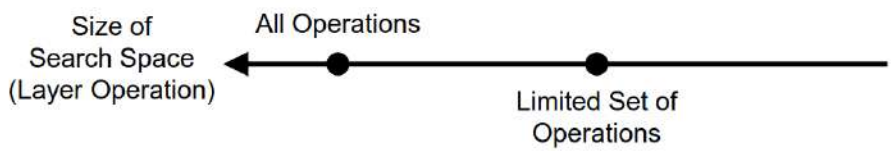
(1) Shrink the Search Space

- **Trade the discoverable architectures for search speed**
- **May irrecoverably limit the achievable network performance**
  - **Domain knowledge learned in manual network design provides guidance**



(1) Shrink the Search Space

- **Search space = layer operations + connections between layers**

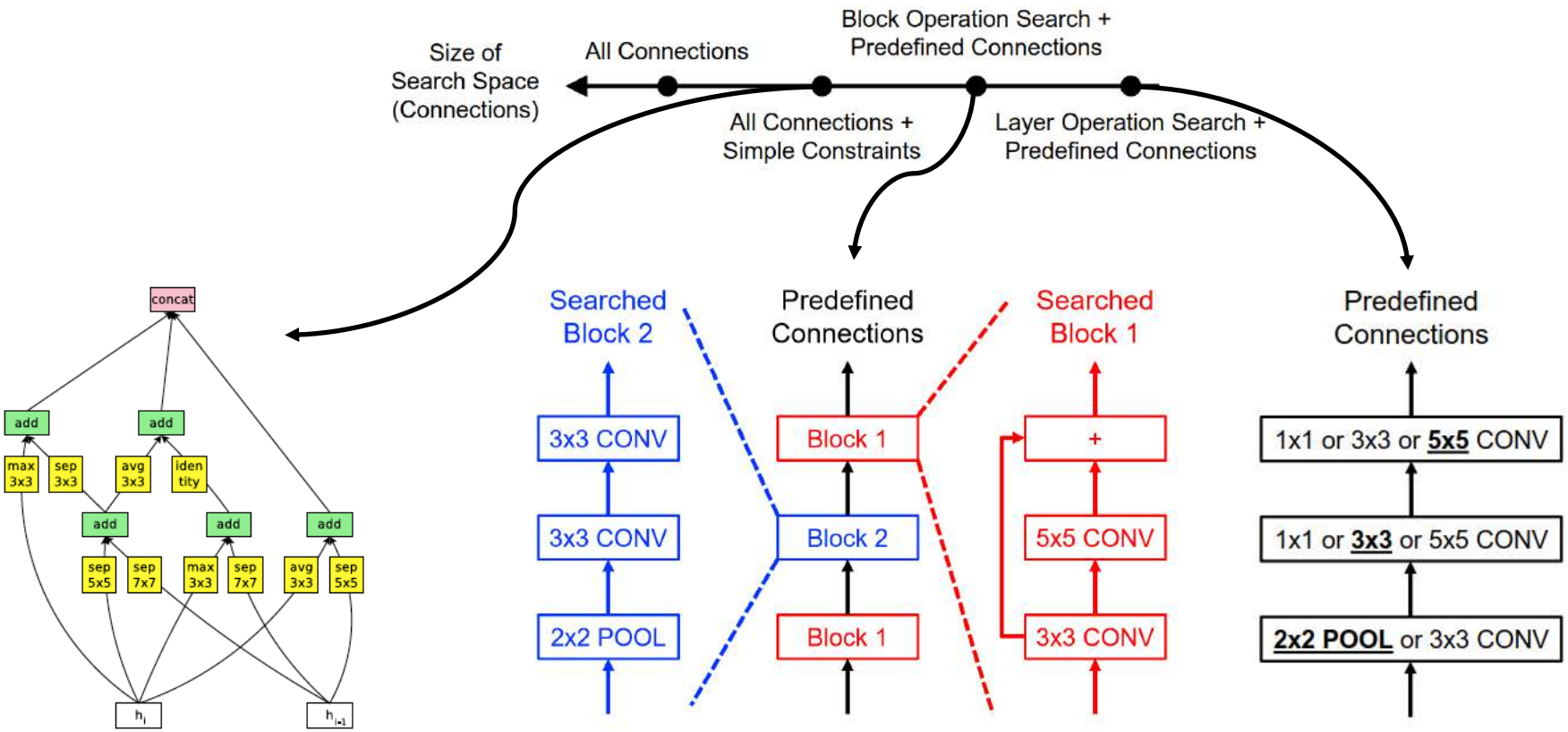


Common layer operations:

- Identity
- 1x3 then 3x1 convolution
- 1x7 then 7x1 convolution
- 3x3 dilated convolution
- 1x1 convolution
- 3x3 convolution
- 3x3 separable convolution
- 5x5 separable convolution
- 3x3 average pooling
- 3x3 max pooling
- 5x5 max pooling
- 7x7 max pooling

(1) Shrink the Search Space

- **Search space = layer operations + <u>connections between layers</u>**



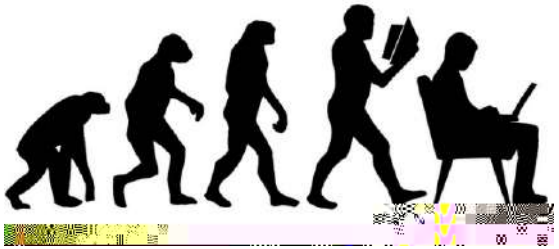
[Zoph et al., arXiv, July 2017]

(2) Improve Optimization Algorithm

Starts from the previous best sample and goes in the best randomly-sampled direction

- The metric does not need to be differentiable
- More complicated

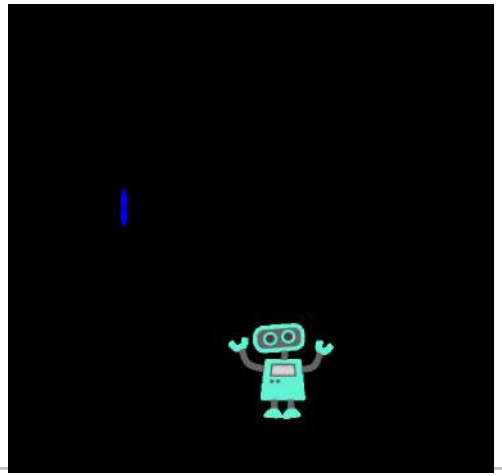
Evolutionary



Learns from the previous samples and infers the best sample

- Better uses the previous samples
- Needs to design and train the agent

Reinforcement Learning



Models the entire surface of the search space and picks the best sample

- Gets rid of the iterative process
- Hard to model a large search space

Bayesian

(3) Simplify the Performance Evaluation

- NAS needs only the rank of the performance values
- Method 1: approximate accuracy
- Method 2: approximate weights
- Method 3: approximate metrics (e.g., latency, energy)

(3) Simplify the Performance Evaluation

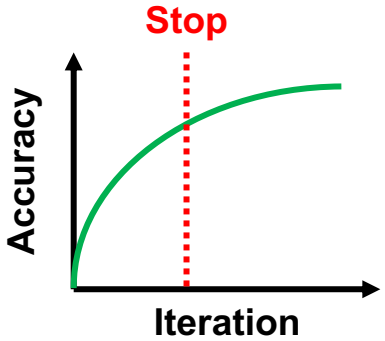
- **NAS needs only the <u>rank</u> of the performance values**
- **Method 1: approximate accuracy**

Proxy Task

E.g., Smaller resolution, simpler tasks

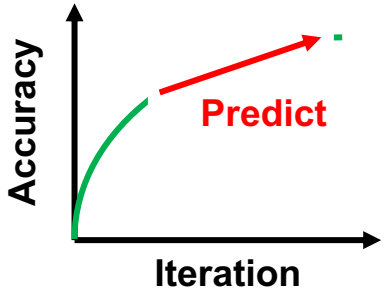
Early Termination

Stop training earlier



Accuracy Prediction

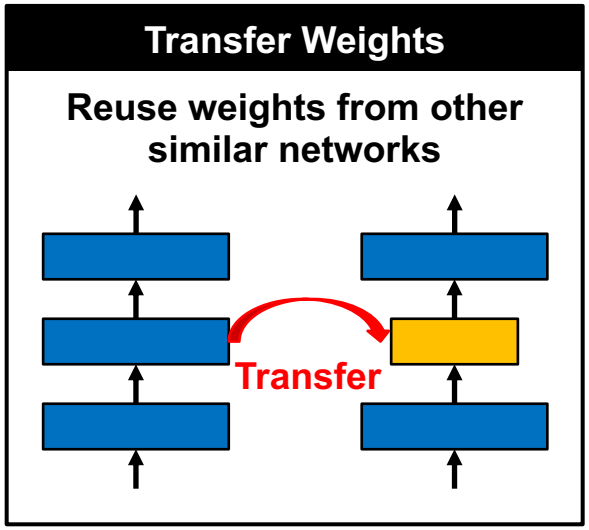
Extrapolate accuracy

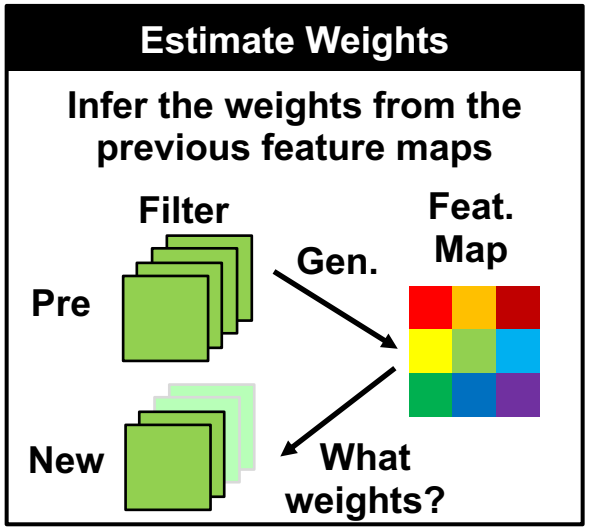


- Method 2: approximate weights
- Method 3: approximate metrics

(3) Simplify the Performance Evaluation

- **NAS needs only the <u>rank</u> of the performance values**
- Method 1: approximate accuracy
- **Method 2: approximate weights**





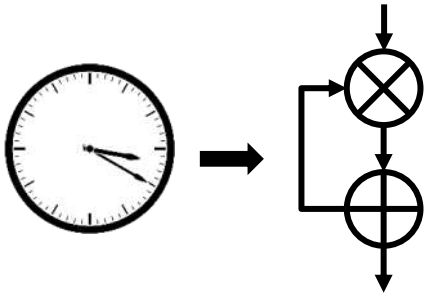
- Method 3: approximate metrics

(3) Simplify the Performance Evaluation

- **NAS needs only the <u>rank</u> of the performance values**
- **Method 1: approximate accuracy**
- **Method 2: approximate weights**
- **Method 3: approximate metrics (e.g., latency, energy)**

Proxy Metric

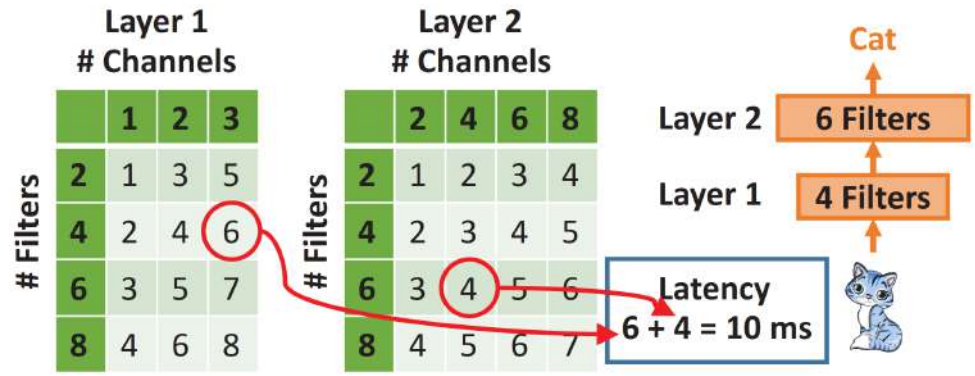
Use an easy-to-compute metric to approximate target



Latency **# MACs**

Look-Up Table

Use table lookup



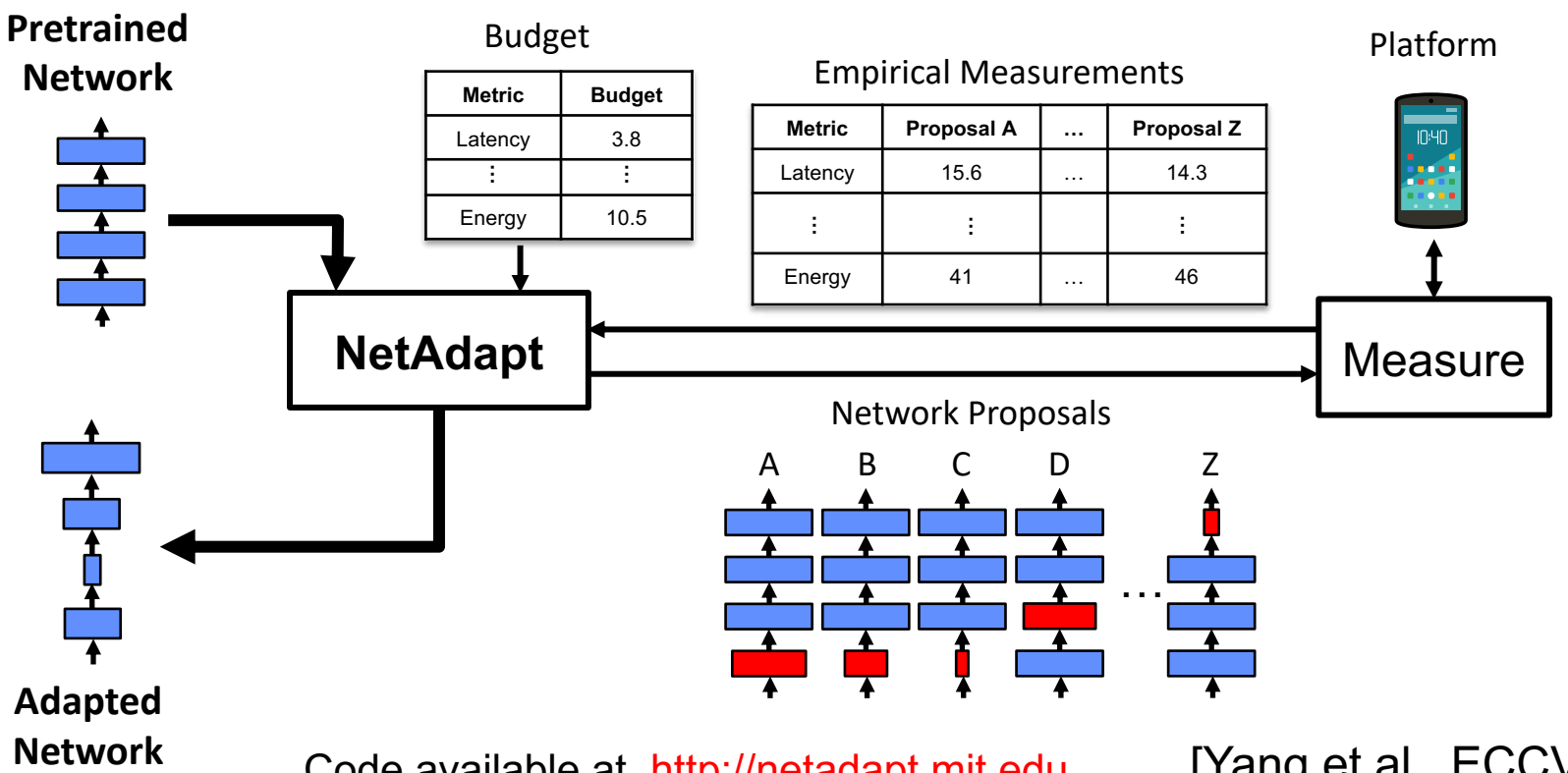
Other Things to Know

- **The components may not be chosen individually**
  - **Some optimization algorithms limit the search space**
  - **Using direct hardware metrics may limit the selection of the optimization algorithms**

- **Commonly overlooked properties**
  - **The complexity of implementation and usage**
  - **The ease of tuning**
  - **The probability of convergence to a good architecture**

NetAdapt: Platform-Aware DNN Adaptation

- **Automatically adapt DNN** to a mobile platform to reach a target latency or energy budget
- An example of **coordinate descent NAS**



Code available at http://netadapt.mit.edu

[Yang et al., ECCV 2018]

Problem Formulation

$$\max_{Net} Acc(Net) \;\; subject\ to\ Res_j(Net) \leq Bud_j, j = 1, \cdots, m$$

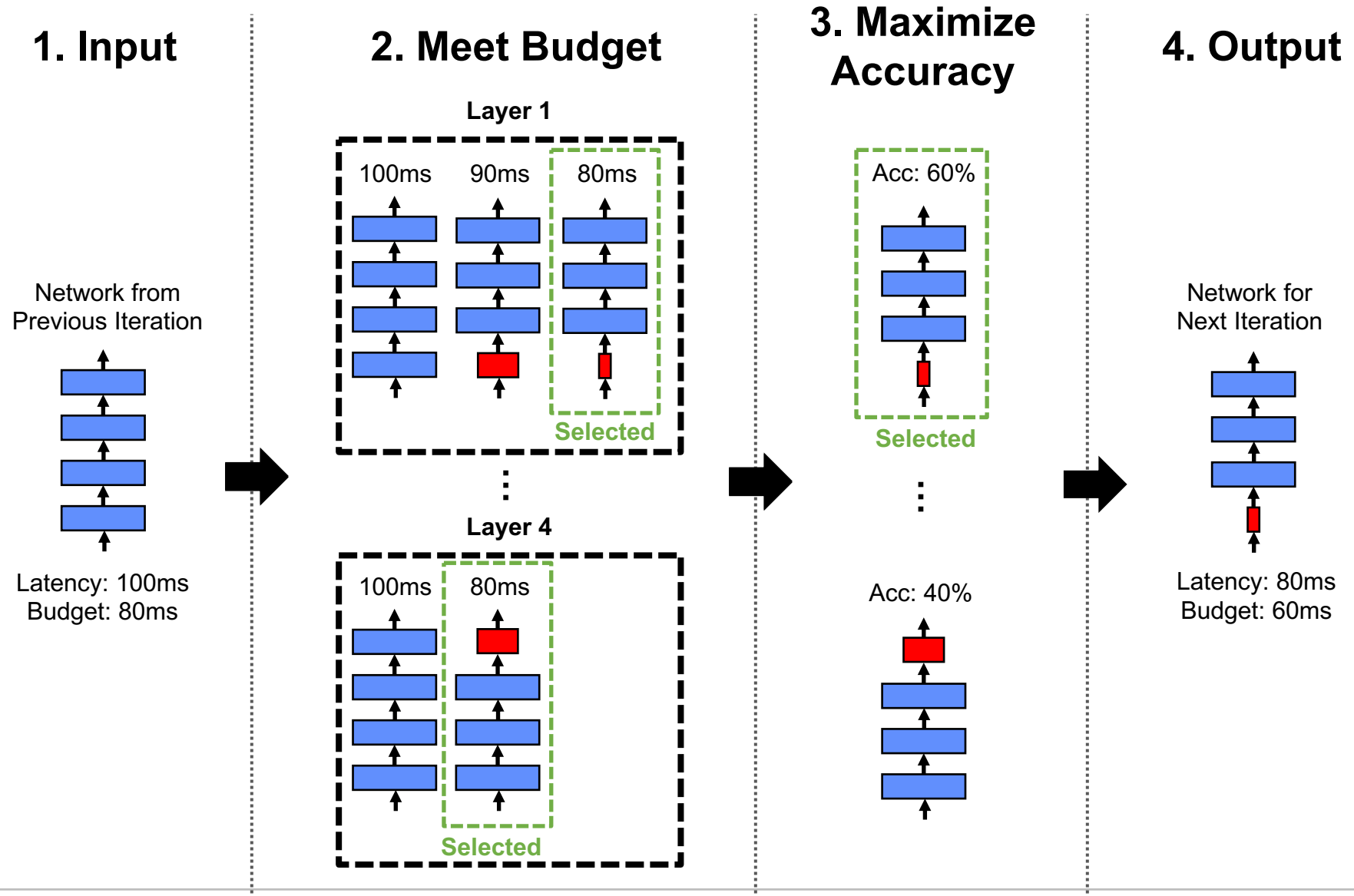
Break into a set of simpler problems and solve iteratively

$$\max_{Net_i} Acc(Net_i) \;\; subject\ to\ Res_j(Net_i) \leq Res_j(Net_{i-1}) - \Delta R_{i,j}, j = 1, \cdots, m$$

**Acc*: accuracy function, *Res*: resource evaluation function,
***ΔR*: resource reduction**, *Bud*: given budget

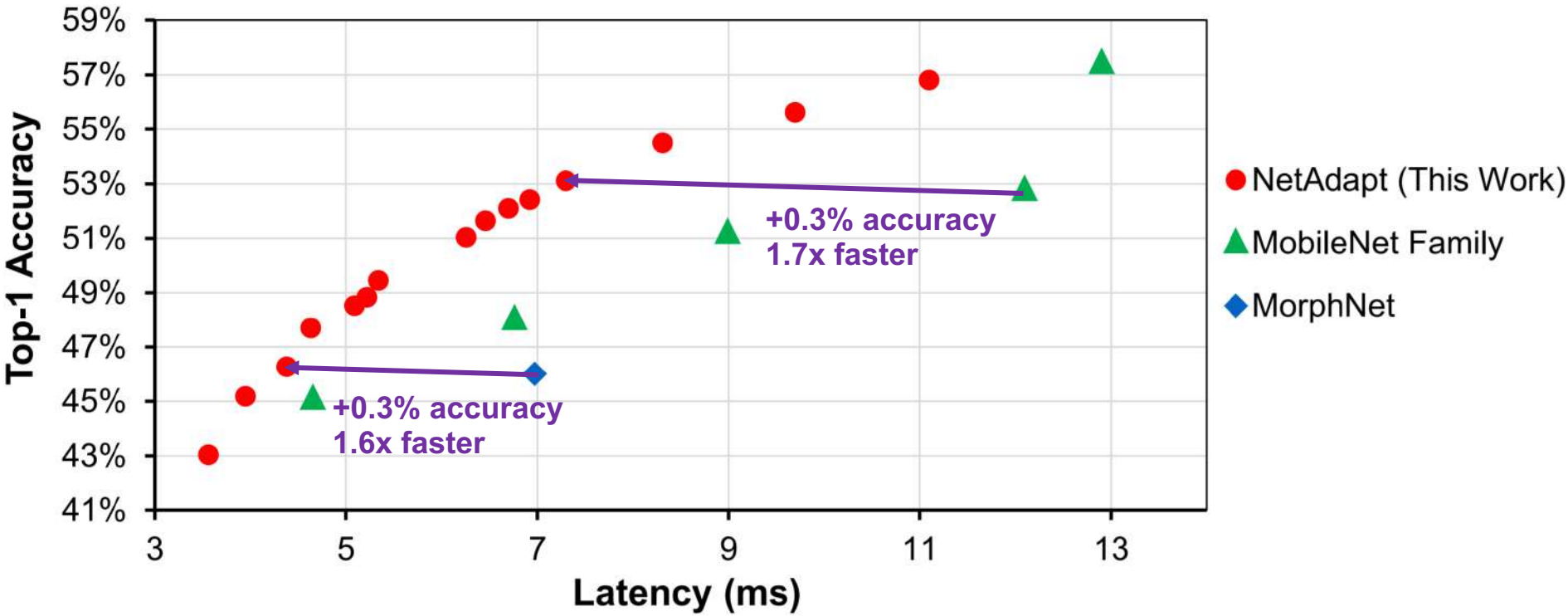
- **Advantages**
  - Supports multiple resource budgets at the same time
  - Guarantees that the budgets will be satisfied because the resource consumption decreases monotonically
  - Generates a family of networks (from each iteration) with different resource versus accuracy trade-offs

Simplified Example of One Iteration



Improved Latency vs. Accuracy Tradeoff

- NetAdapt boosts **the real inference speed** of MobileNet by up to 1.7x with higher accuracy

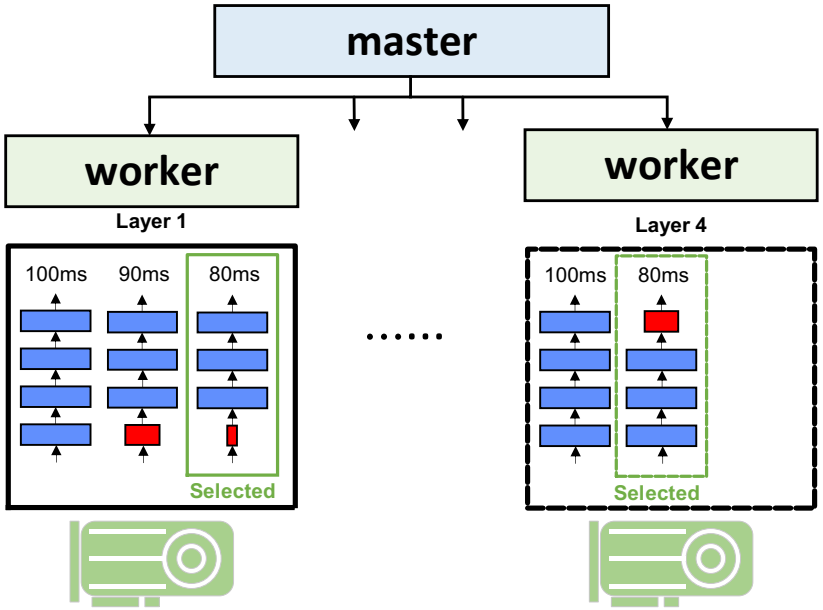


*Tested on the ImageNet dataset and a Google Pixel 1 CPU

Reference:
MobileNet: Howard et al, "Mobilenets: Efficient convolutional neural networks for mobile vision applications", arXiv 2017
MorphNet: Gordon et al., "Morphnet: Fast & simple resource-constrained structure learning of deep networks", CVPR 2018

Code of NetAdapt

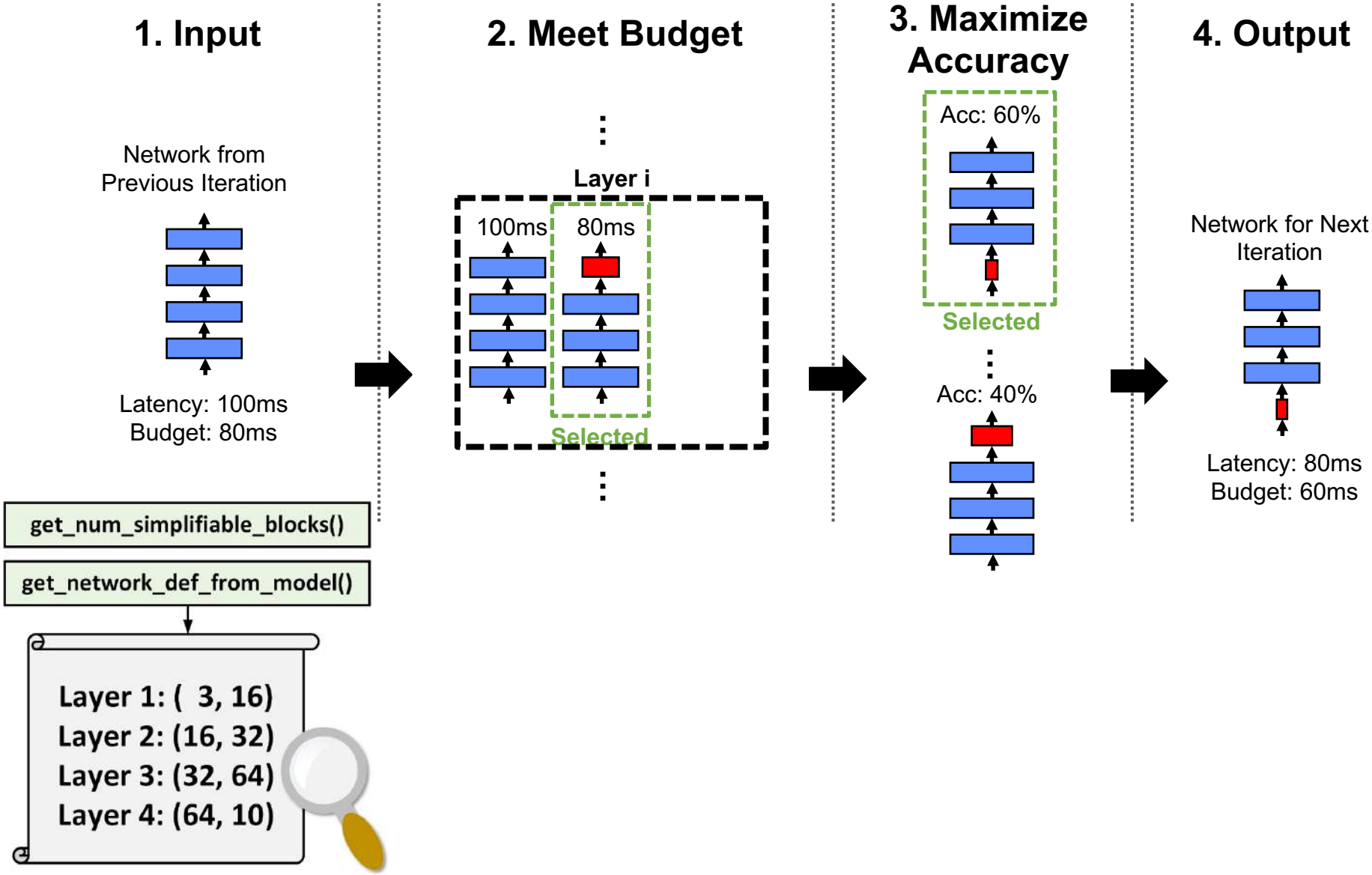
- Reimplemented framework on PyTorch
- **Flexible**: can support **different networks and tasks**
- **Scalable**: spawn multiple workers to simplify networks in parallel



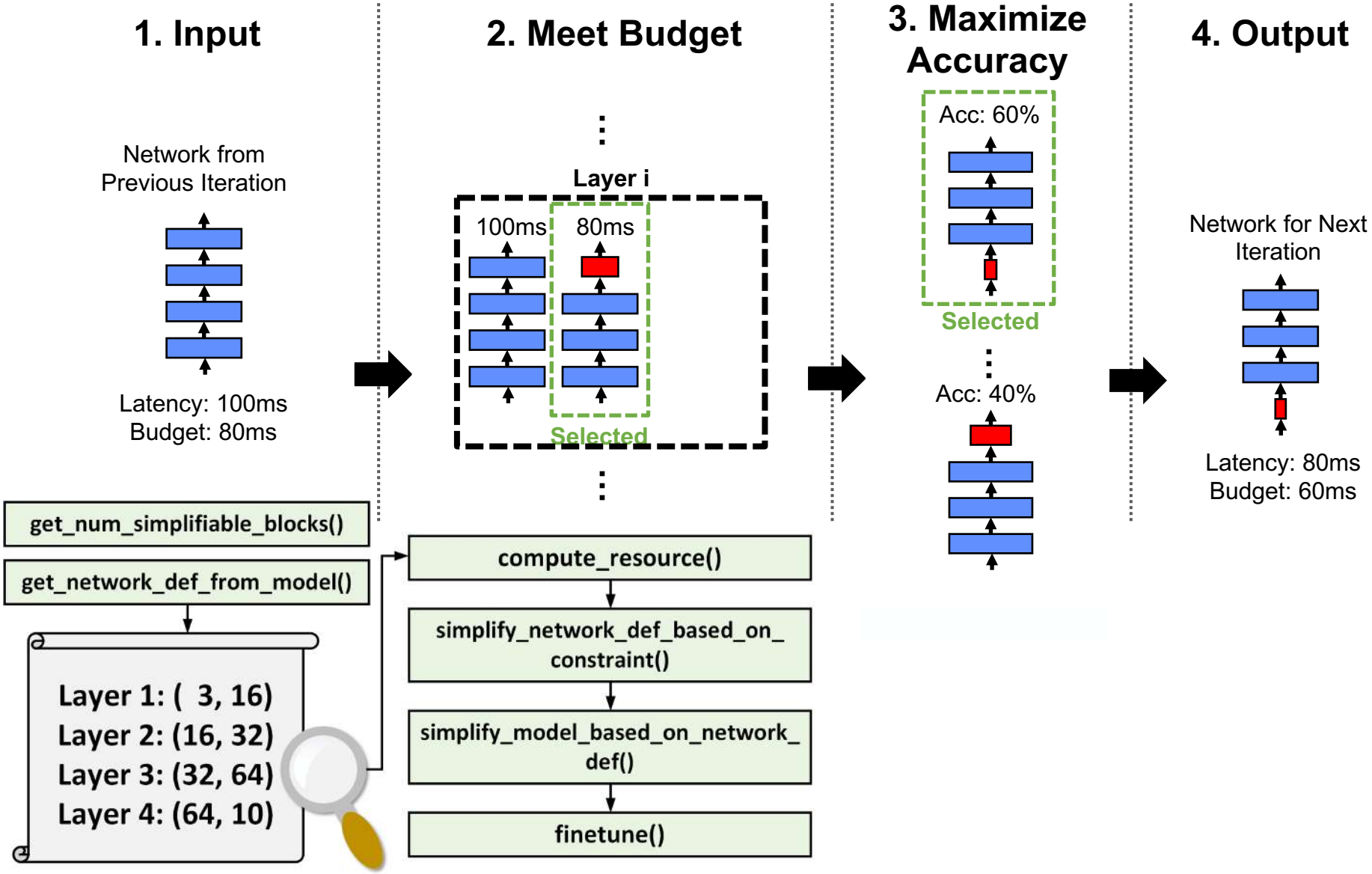
- **Easy-to-use**: require implementing only **one file (8 functions)**

Code available at https://github.com/denru01/netadapt

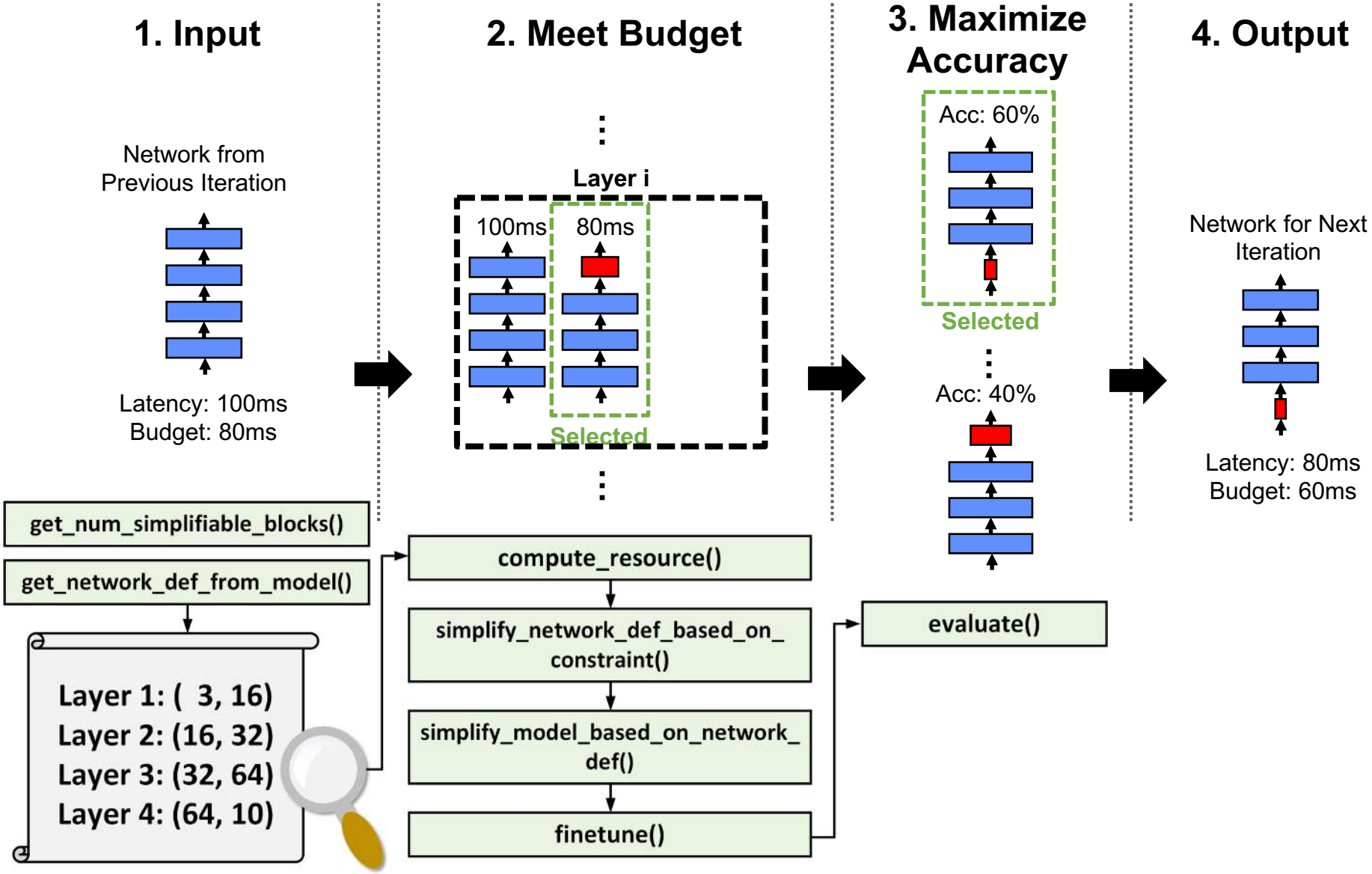
Code of NetAdapt



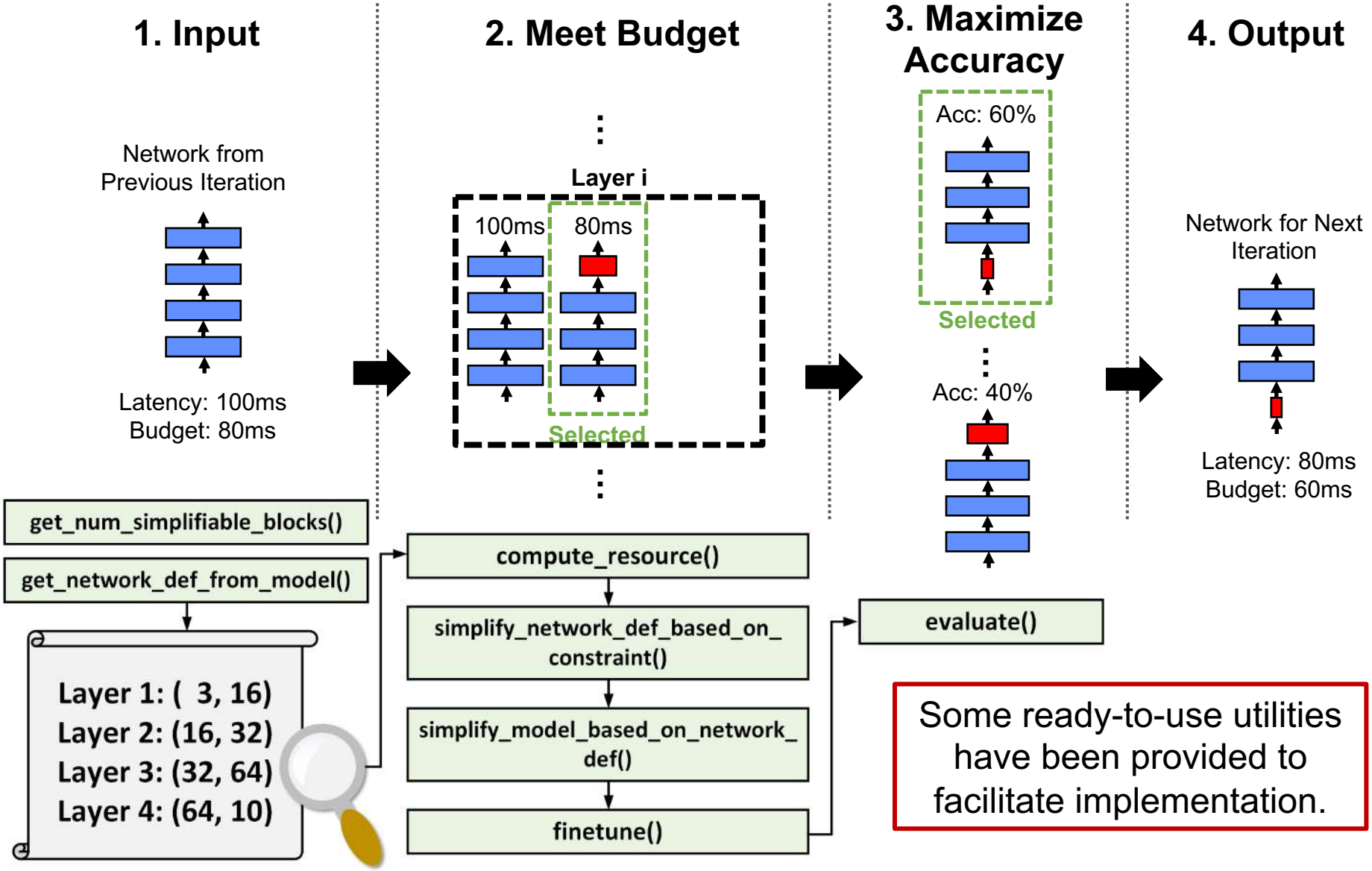
Code of NetAdapt



Code of NetAdapt



Code of NetAdapt

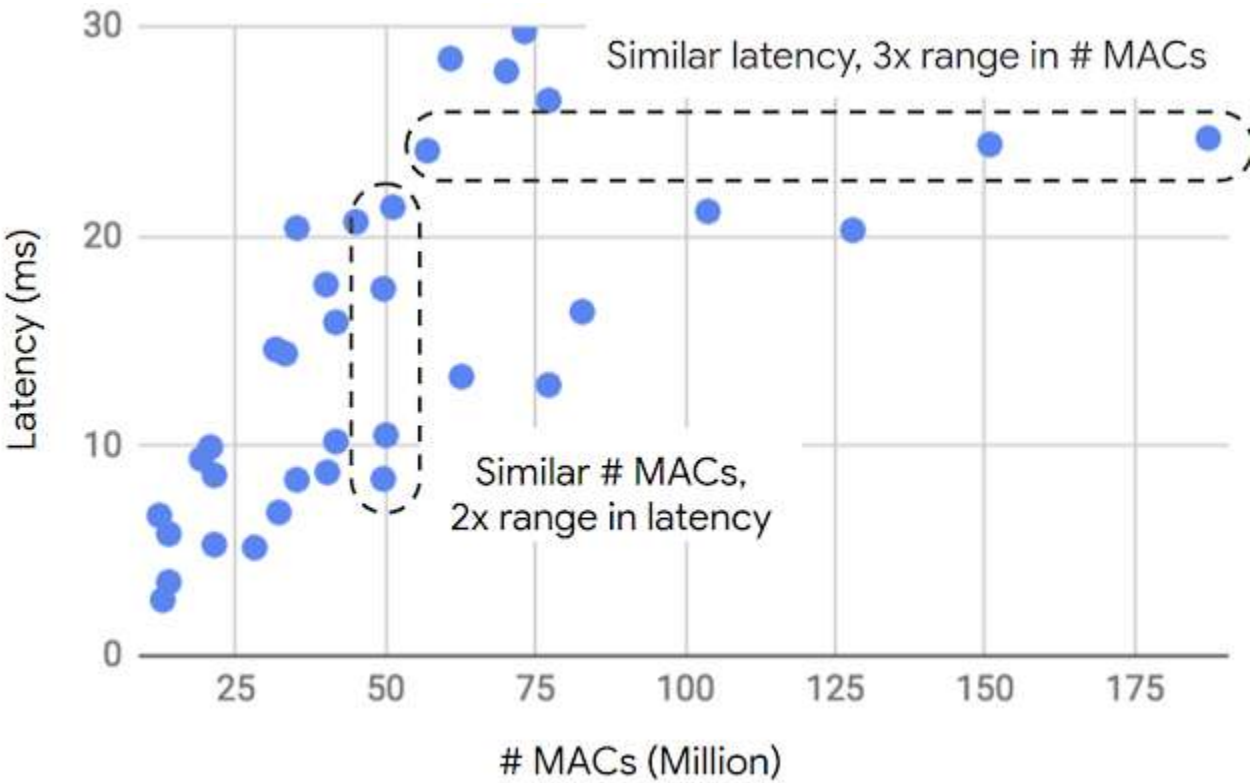


Some ready-to-use utilities have been provided to facilitate implementation.

Hardware In the Loop

of Operations vs. Latency

- # of operations (MACs) does not approximate latency well

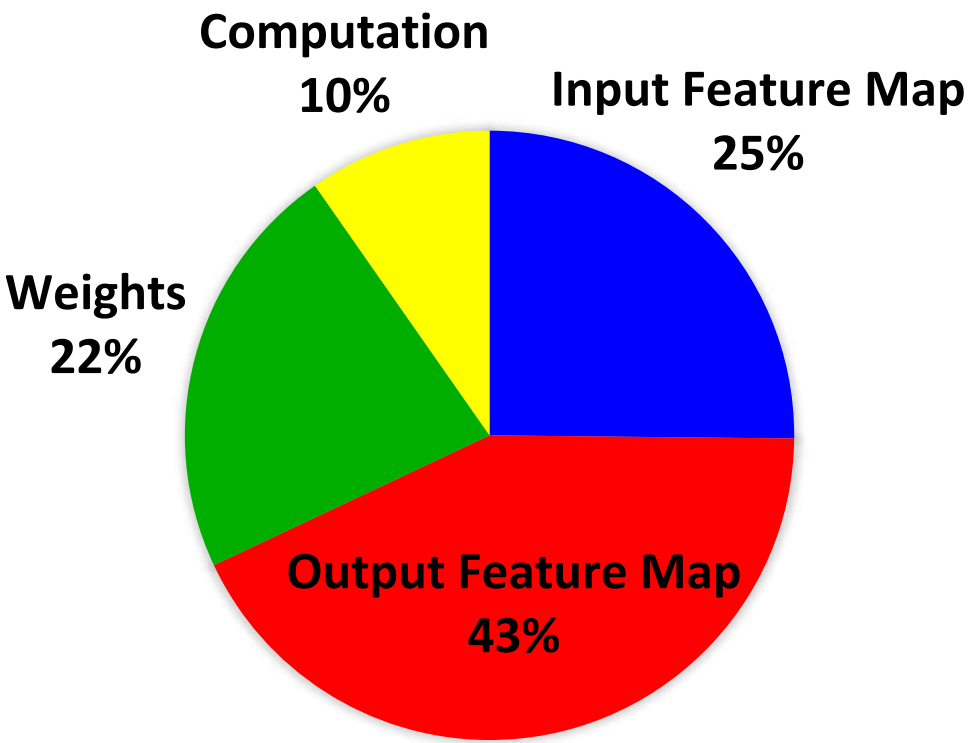


Source: Google (https://ai.googleblog.com/2018/04/introducing-cvpr-2018-on-device-visual.html)

of Weights vs. Energy

- Number of weights ***alone*** is not a good metric for energy
- **All data types** should be considered

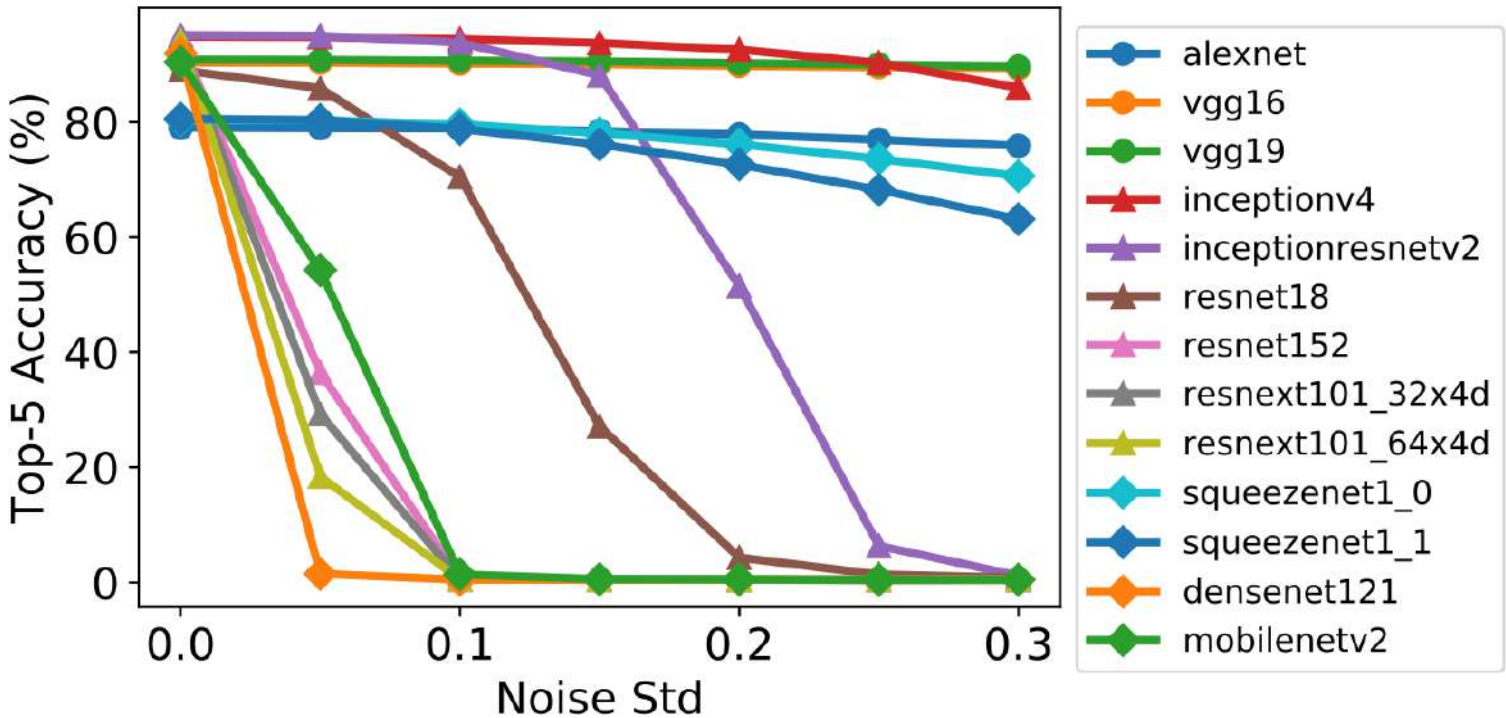
Energy Consumption of GoogLeNet



[Yang et al., CVPR 2017]

Other Hardware Metrics

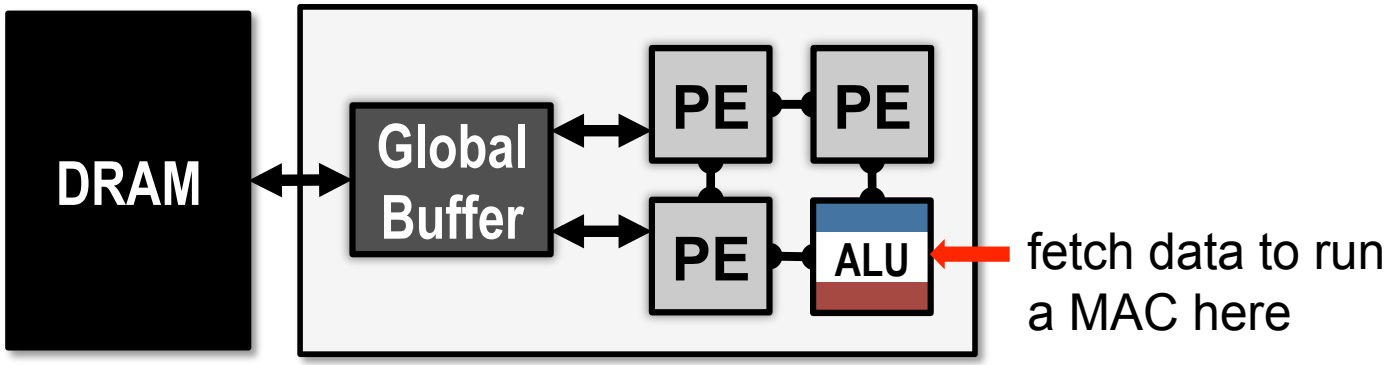
- **E.g., noise resilience in analog accelerators**

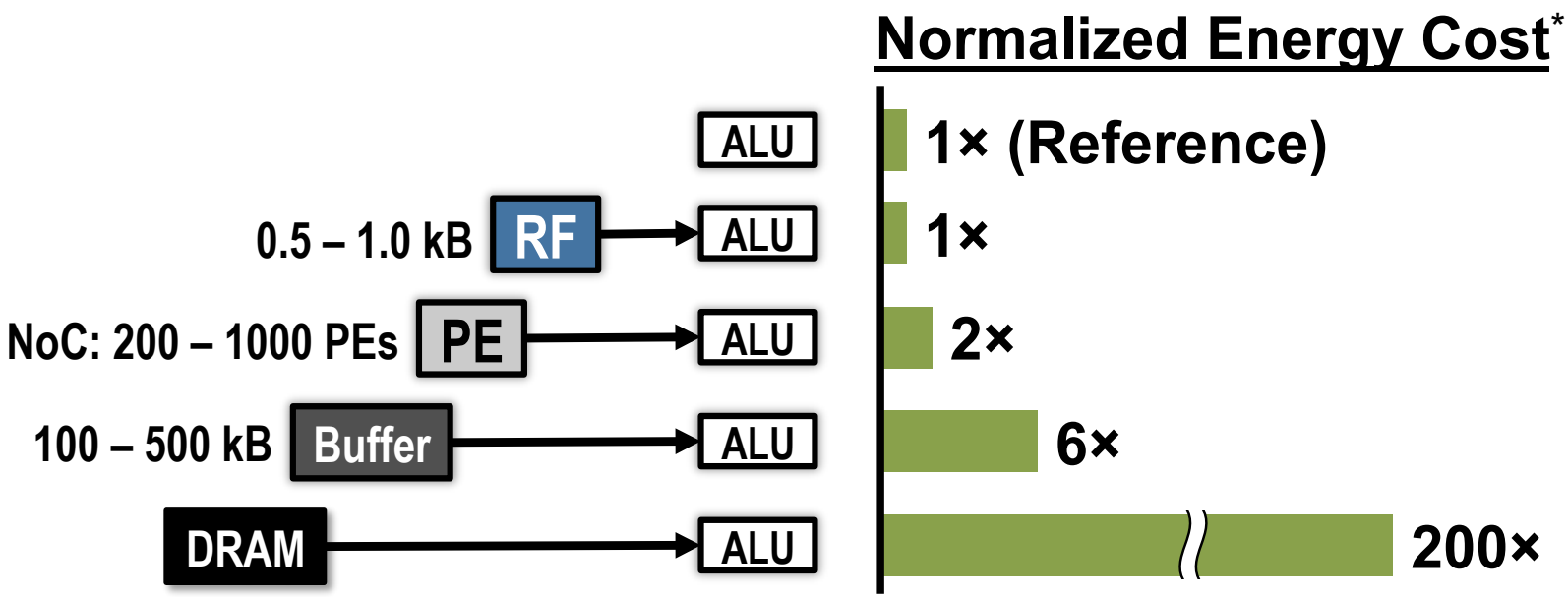


DNN model that gives highest accuracy on a digital processor may not be the best for an analog processor

[Yang et al., IEDM 2019]

Data Movement is Expensive

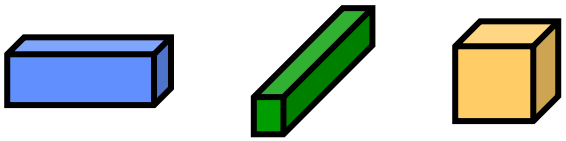




* measured from a commercial 65nm process

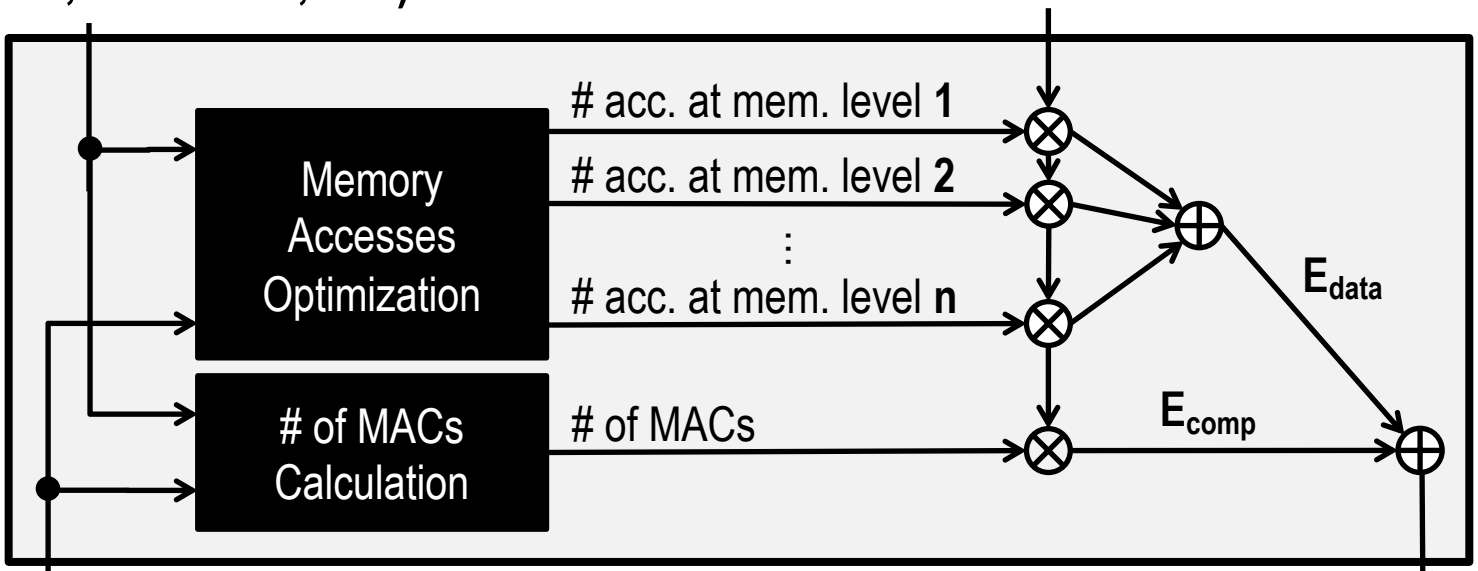
Energy of weight depends on **memory hierarchy** and **dataflow**

Energy Estimation Methodology



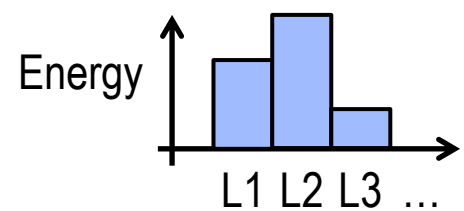
DNN Shape Configuration
(# of channels, # of filters, etc.)

Hardware Energy Costs of each MAC and Memory Access



DNN Weights and Input Data

[0.3, 0, -0.4, 0.7, 0, 0, 0.1, …]



DNN Energy Consumption

Energy Estimation Tool V1

Website: https://energyestimation.mit.edu/

Deep Neural Network Energy Estimation Tool

Overview

This Deep Neural Network Energy Estimation Tool is used for evaluating and designing energy-efficient deep neural networks that are critical for embedded deep learning processing. Energy estimation was used in the development of the energy-aware pruning method (Yang et al., CVPR 2017), which reduced the energy consumption of AlexNet and GoogLeNet by 3.7x and 1.6x, respectively, with less than 1% top-5 accuracy loss. This website provides a simplified version of the energy estimation tool for shorter runtime (around 10 seconds).

Input

To support the variety of toolboxes, this tool takes a single network configuration file. The network configuration file is a txt file, where each line denotes the configuration of a CONV/FC layer. The format of each line is:

height nChannels nZeroEntries y bottom right
width nMapsOrFilts bitwidth x top left

2 , 27 , 27 , 96 , 44 , 3.5731e+05 , 16 , 5 , 5 , 48 , 256 , 0 , 16 , 27 , 27 , 256 , 44 , 6.623e+06 , 16 , 1 , 1 , 2 , 2 , 2 , 2

Layer_Index | Conf_IfMap | Conf_Filt | Conf_OfMap | Stride | Padding

- Layer Index: the index of the layer, from 1 to the number of layers. It should be the same as the line number.
- Conf_IfMap, Conf_Filt, Conf_OfMap: the configuration of the input feature maps, the filters and the output feature maps. The configuration of each of the three data types is in the format of "height width number_of_channels number_of_maps_or_filts number_of_zero_entries bitwidth_in_bits".
- Stride: the stride of this layer. It is in the format of "stride_y stride_x".
- Pad: the amount of input padding. It is in the format of "pad_top pad_bottom pad_left pad_right".

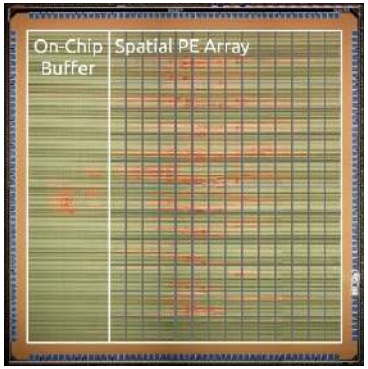
Therefore, there will be 25 entries separated by commas in each line.

Running the Estimation Model

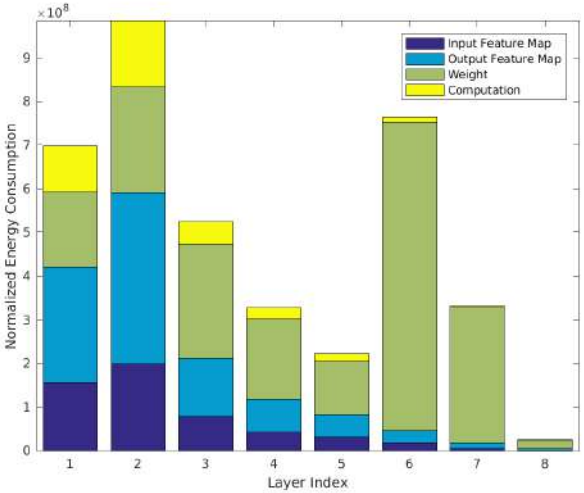
After creating your text file, follow these steps to upload your text file and run the estimation model:

1. Check the "I am not a robot" checkbox and complete the Google reCAPTCHA challenge. Help us prevent spam.
2. Click the "Choose File" button below to choose your text file from your computer.
3. Click the "Run Estimation Model" button below to upload your text file and run the estimation model.

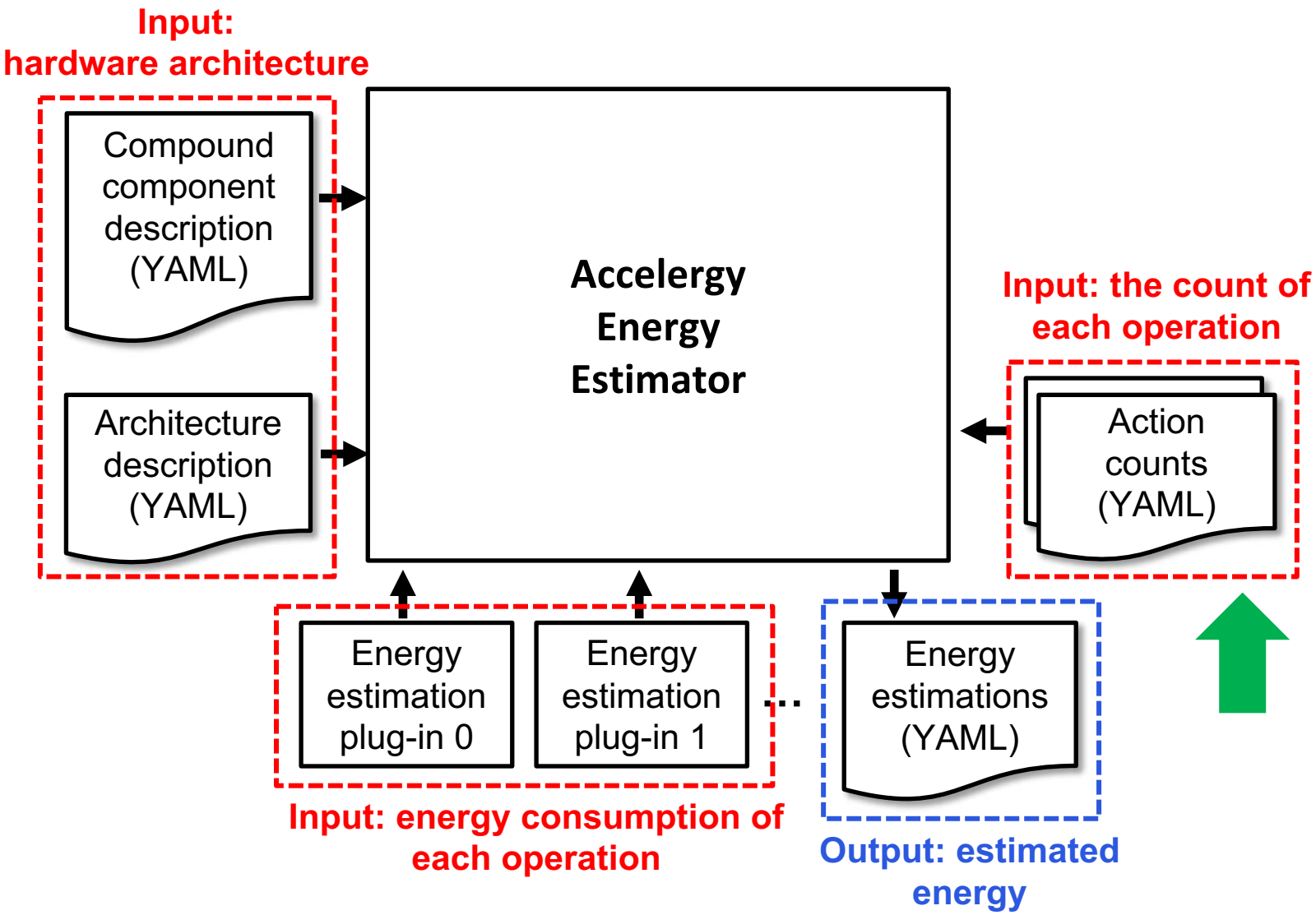
Eyeriss V1



Output DNN energy breakdown across layers



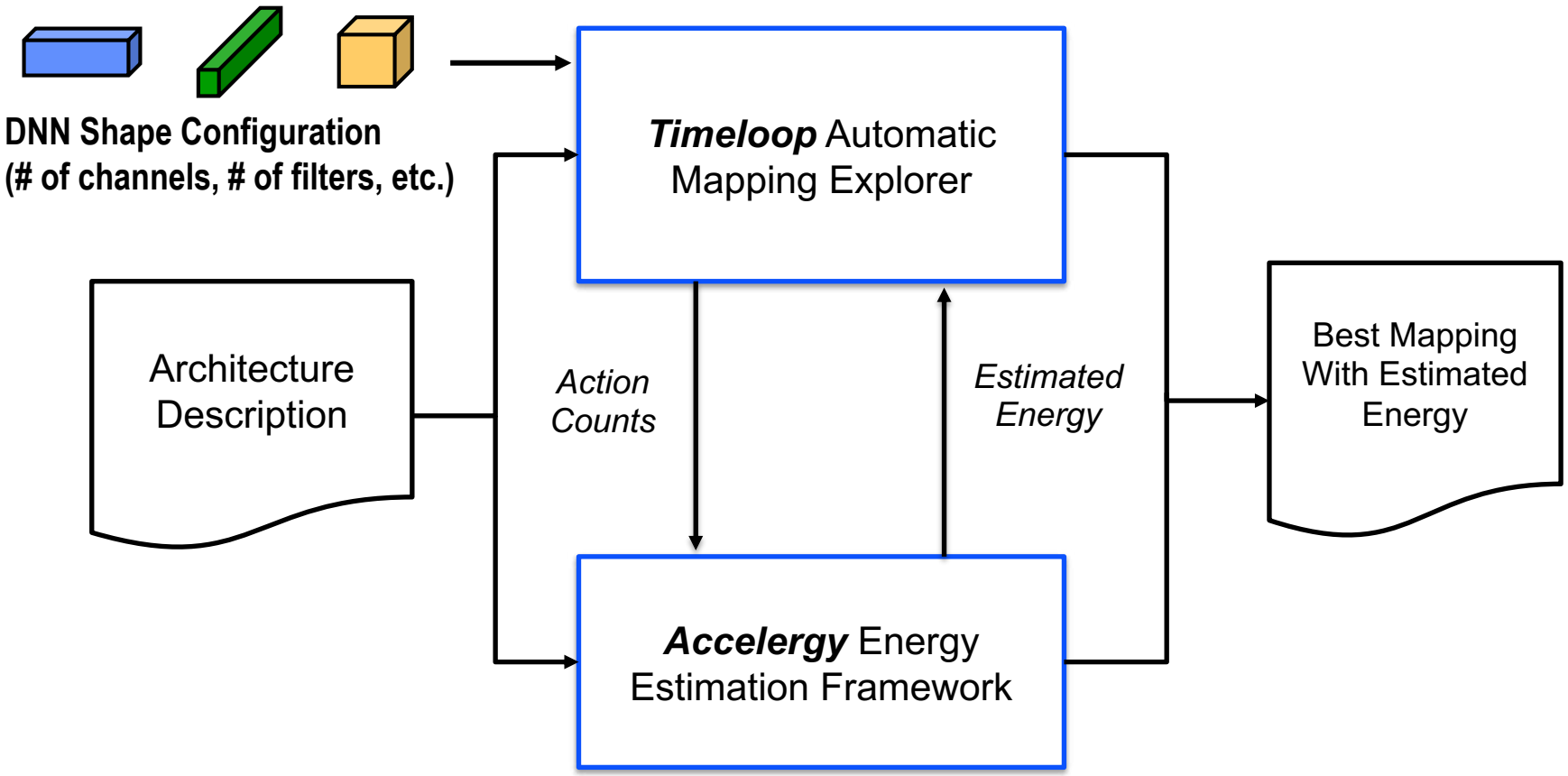
Energy Estimation Tool V2 - Accelergy



[Wu et al., ICCAD 2019]

Energy Estimation Tool V2 - Accelergy

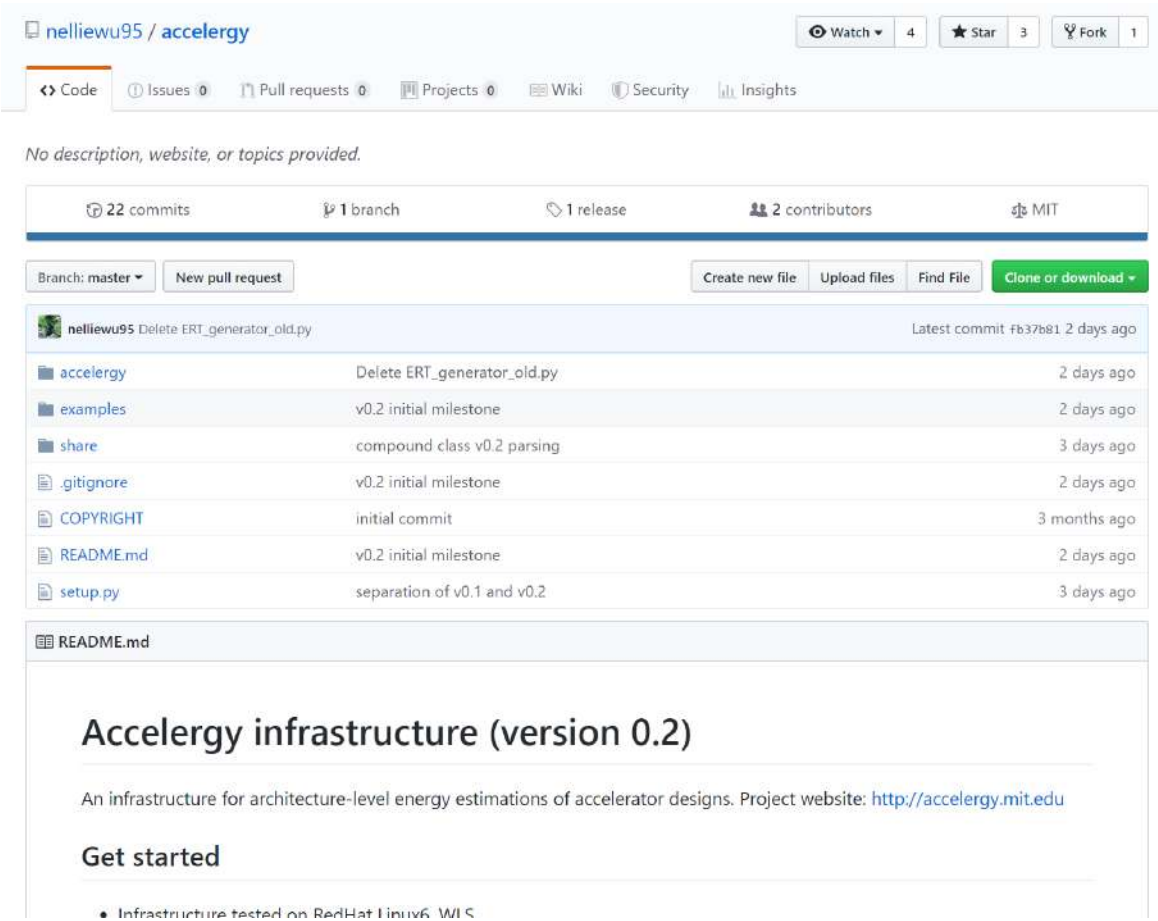
Integrate 3rd-party tools to generate the count of each operation



Tutorial at MICRO 2019: http://accelergy.mit.edu/tutorial.html

Energy Estimation Tool V2 - Accelergy

Website: https://accelergy.mit.edu/

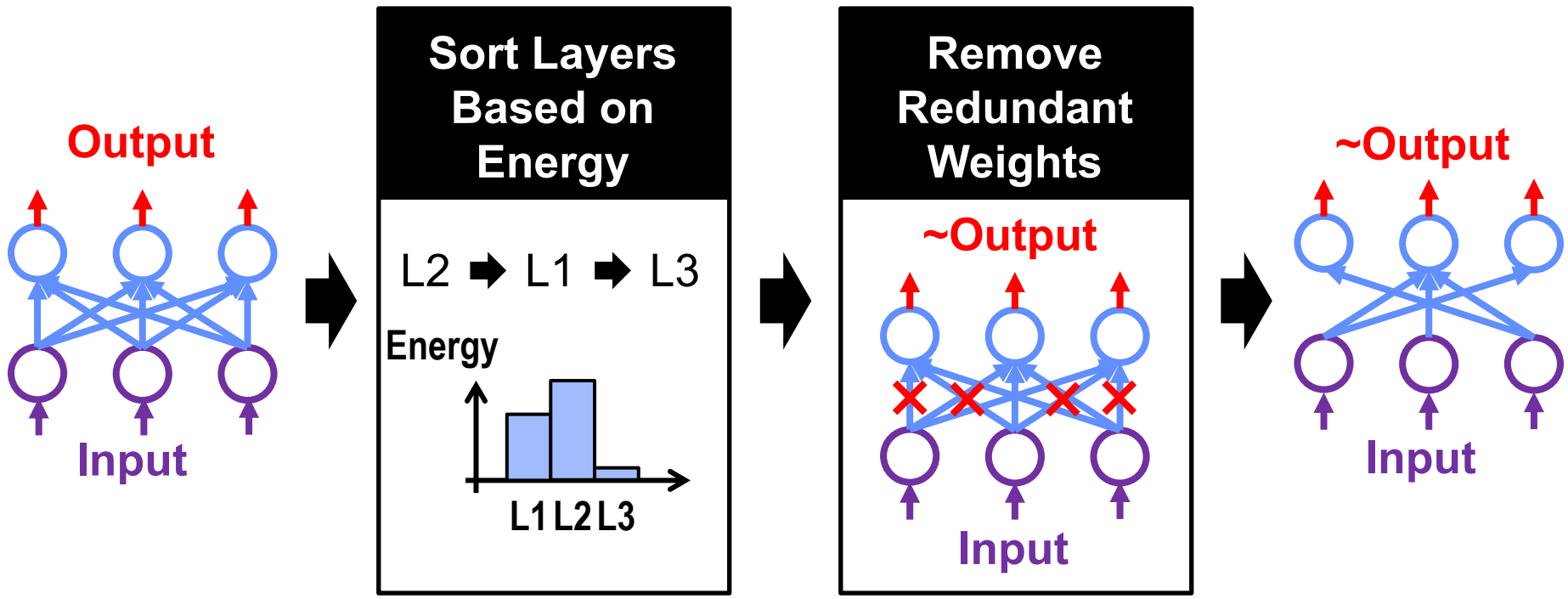


Output DNN energy breakdown across components

```
hierarchy.PE[0].ifmap_sp: 140.0
hierarchy.PE[0].mac[0]: 70.0
hierarchy.PE[0].mac[1]: 70.0
hierarchy.PE[1].ifmap_sp: 180.0
hierarchy.PE[1].mac[0]: 70.0
hierarchy.PE[1].mac[1]: 70.0
hierarchy.weights_glb: 5400.0
```

Energy-Aware Pruning

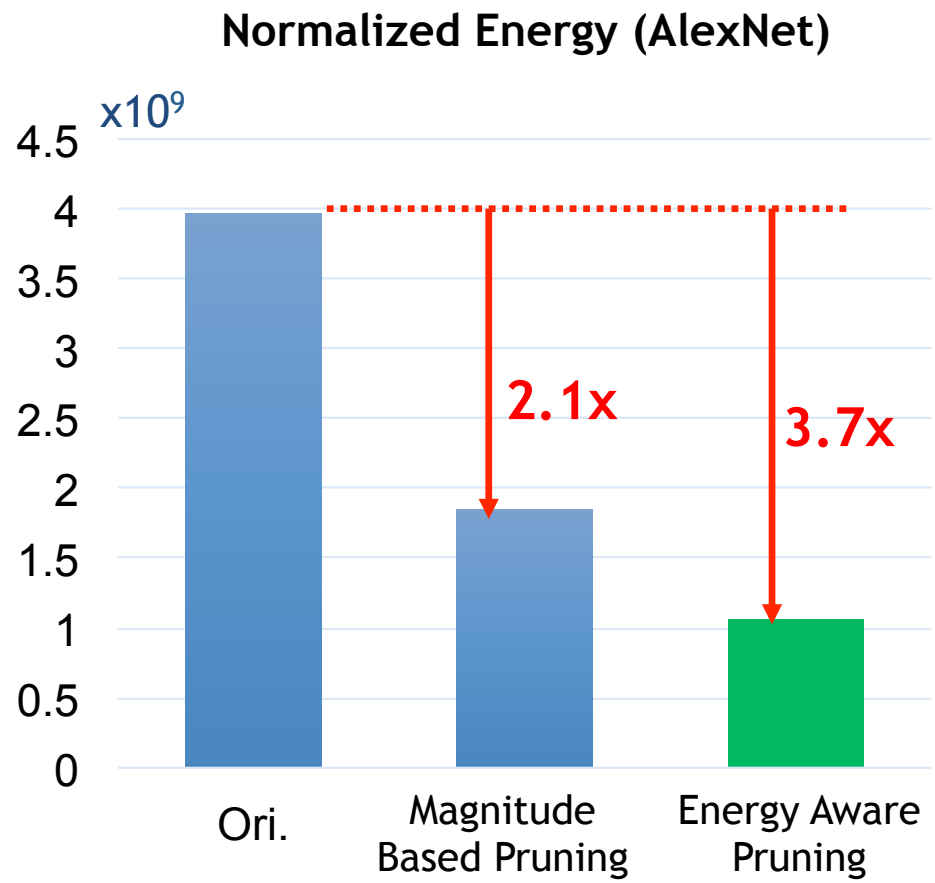
- Problem formulation: $\min_{Net} Erg(Net)\ subject\ to\ Acc(Net) \geq Th$
- Reduces energy by **removing redundant weights**
- Uses **estimated energy** to guide the layer-by-layer pruning
  - Prunes the layer that consume the most energy first



Energy-Aware Pruning

Directly target energy and incorporate it into the optimization of DNNs to provide greater energy savings

- Sort layers based on energy and prune layers that consume most energy first
- EAP reduces AlexNet energy by **3.7x** and outperforms the previous work that uses magnitude-based pruning by **1.7x**

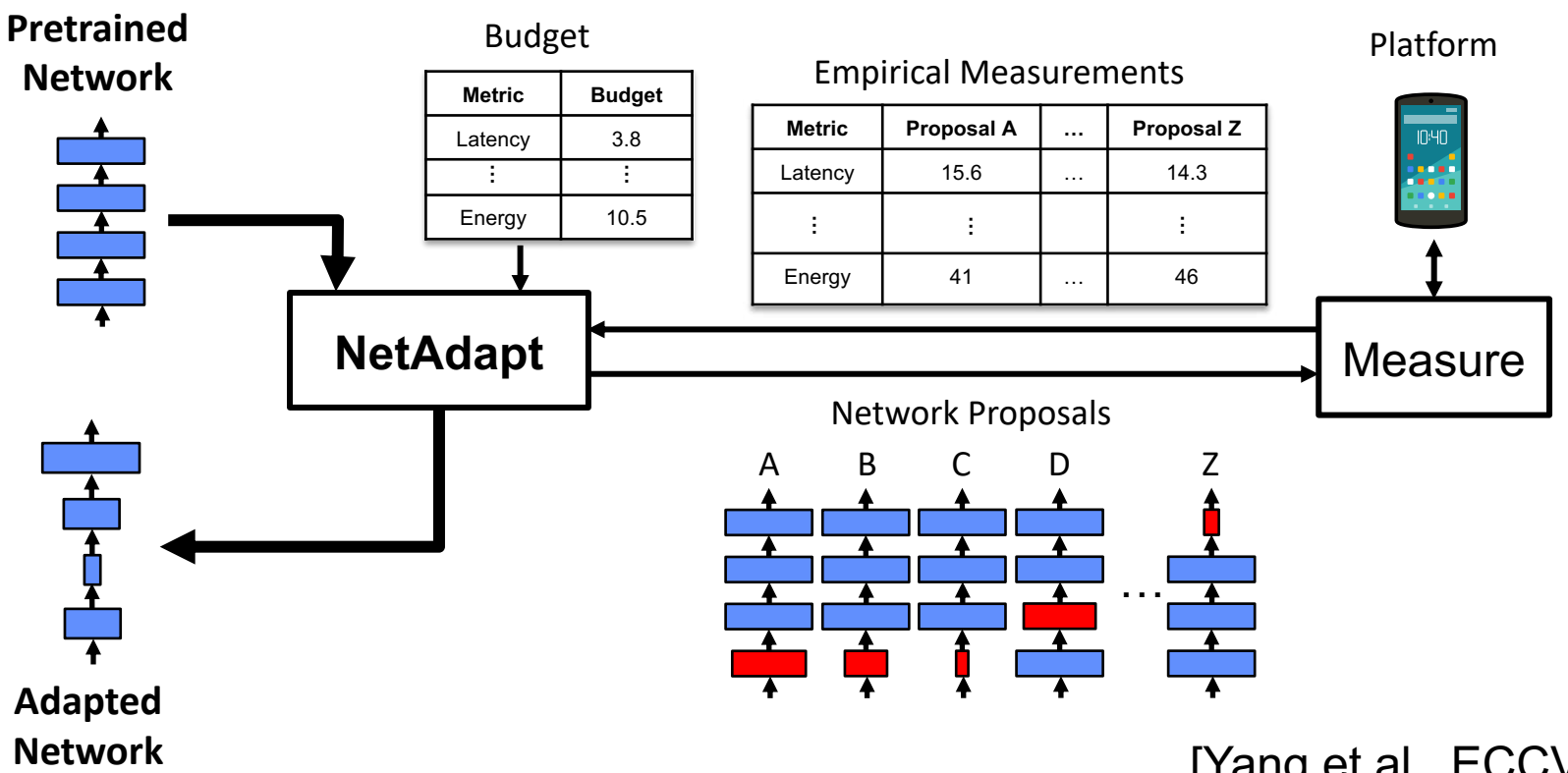


Pruned models available at http://eyeriss.mit.edu/energy.html

[Yang et al., CVPR 2017]

NetAdapt: Platform-Aware DNN Adaptation

- **Automatically adapt DNN** to a mobile platform to reach a target latency or energy budget
- Use **empirical measurements** to guide optimization (avoid modeling of tool chain or platform architecture)



[Yang et al., ECCV 2018]

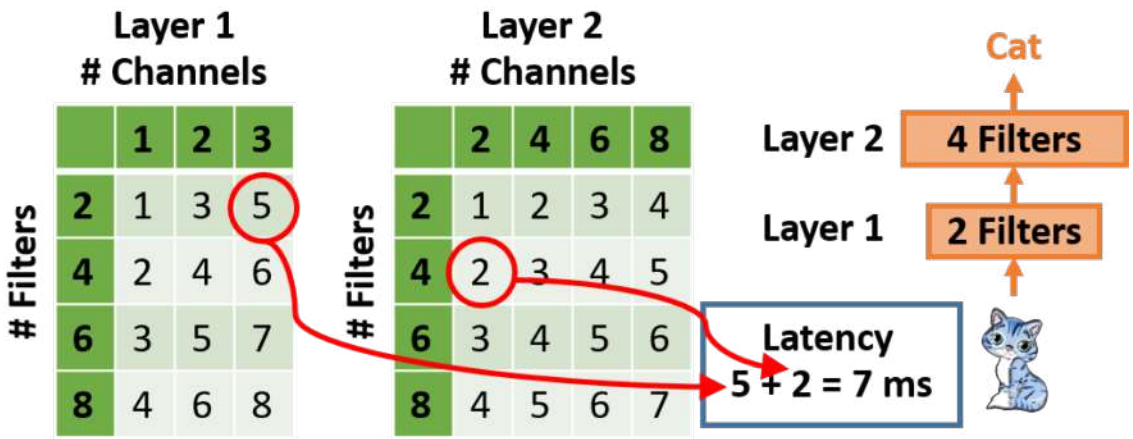
NetAdapt: Using Direct Metrics is Important

- If NetAdapt was guided by the number of MACs, it would also achieve a better accuracy-MAC trade-off
- However, it does not mean lower latency
- It is important to incorporate direct metrics rather than indirect metrics into the design of DNNs

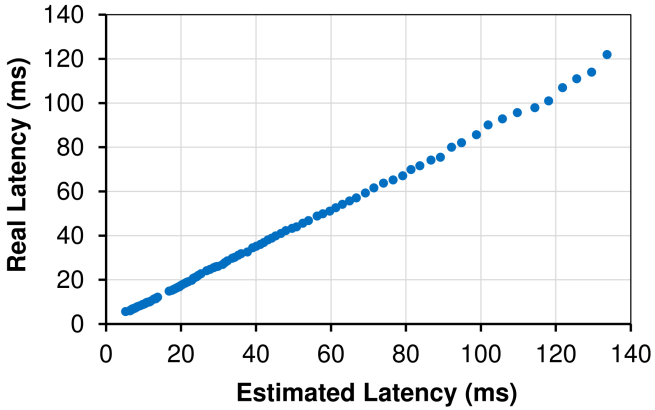
Network	Top-1 Accuracy	# of MACs (M)	Latency (ms)
Small MobileNet V1	45.1 (+0)	13.6 (100%)	4.65 (100%)
NetAdapt	46.3 (+1.2)	11.0 (81%)	6.01 (129%)
Large MobileNet V1	68.8 (+0)	325.4 (100%)	69.3 (100%)
NetAdapt	69.1 (+0.3)	284.3 (87%)	74.9 (108%)

NetAdapt: Fast Resource Consumption Estimation

- Taking measurements can be slow due to the long turn-around time and the limited number of platforms
- Solution: use per-layer lookup tables
  - The network latency can be estimated by the sum of the latency of each layer
  - The layers with the same configuration only need to be measured once
  - The network-wise lookup table grows exponentially with the number of layers



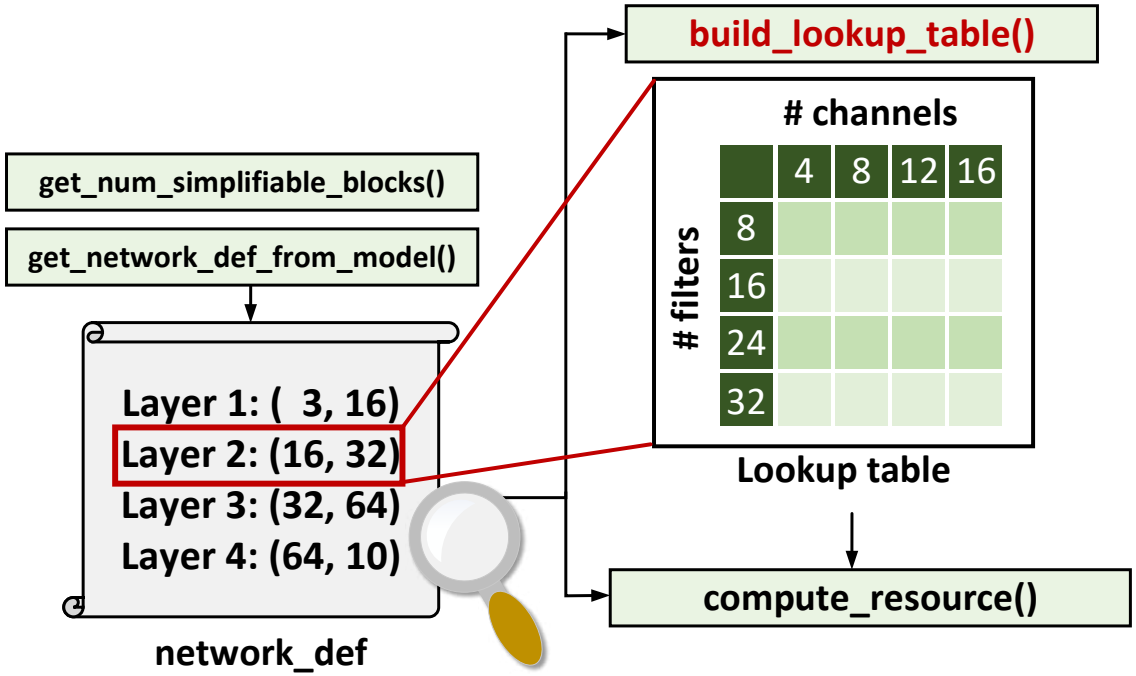
Fast Resource Consumption Estimation



Real Latency vs. Estimated Latency

NetAdapt: Code

- Support building and using lookup tables



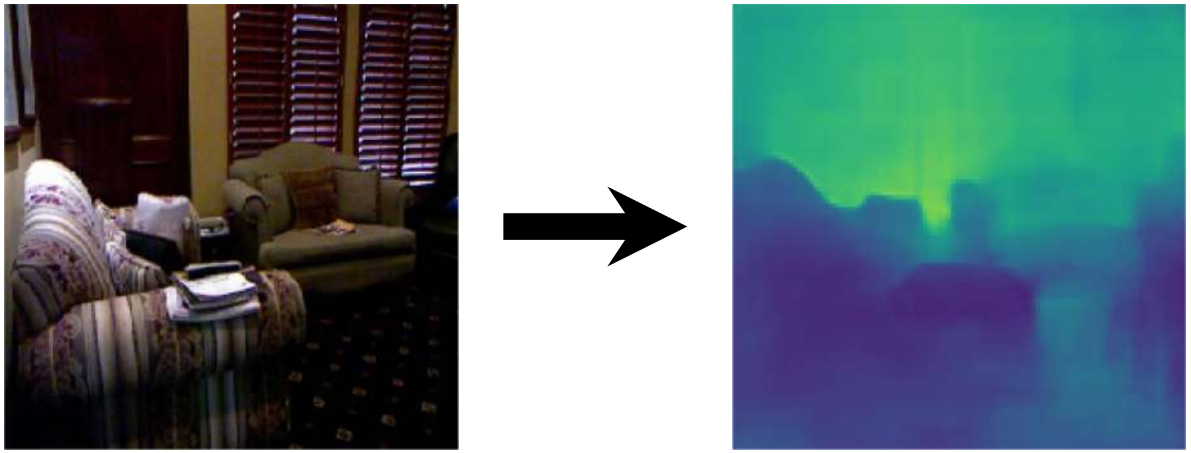
Code available at https://github.com/denru01/netadapt

Part 3: Applications (Beyond Image Classification)

FastDepth: Fast Monocular Depth Estimation

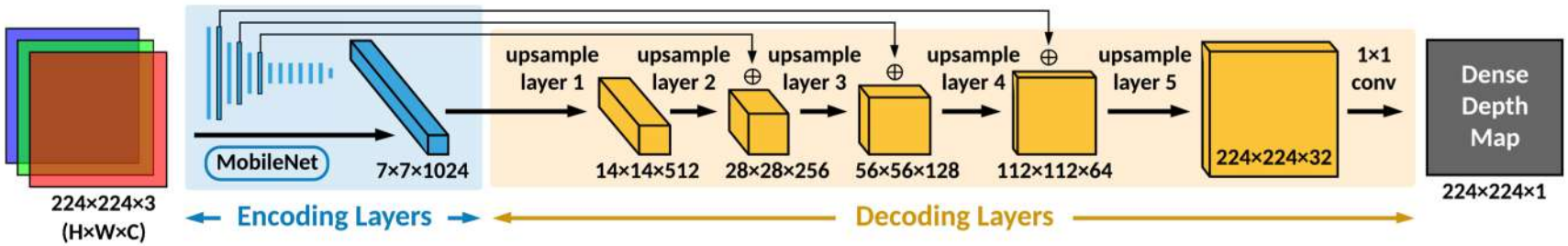
- Real-time low-power depth sensing is critical for navigation of small robotic vehicles.
- Depth estimation from a single RGB image desirable, due to the relatively low cost and size of monocular cameras.

RGB **Prediction**



Our goal is to enable **high accuracy, low latency, high throughput monocular depth estimation** on a **deployable embedded system**.

Efficient Network Design for FastDepth



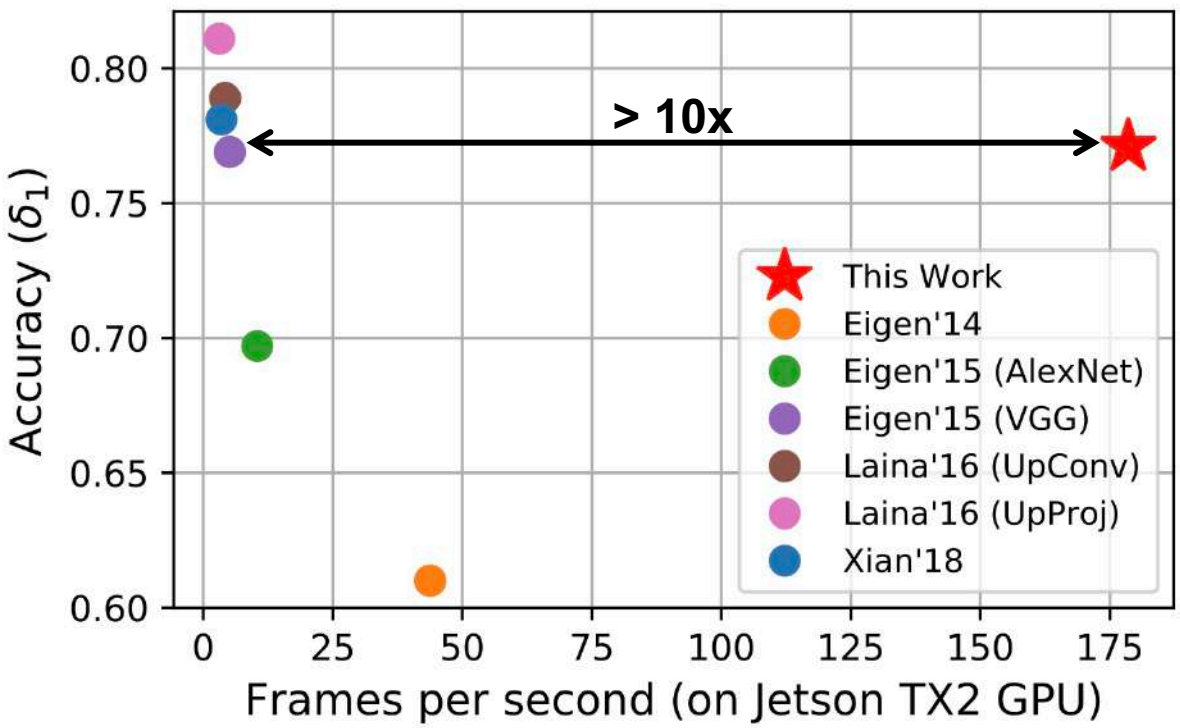
FastDepth achieves high frame rates through

- An efficient and lightweight encoder-decoder network architecture with a low-latency decoder design incorporating depthwise separable layers and additive skip connections
- Network pruning (NetAdapt) applied to whole encoder-decoder network
- Platform-specific compilation (TVM) targeting embedded systems

[Wofk*, Ma* et al., ICRA 2019]

FastDepth: Fast Monocular Depth Estimation

Depth estimation at **high frame rates on an embedded platform** (an order of magnitude faster than previous approaches) while still maintaining accuracy

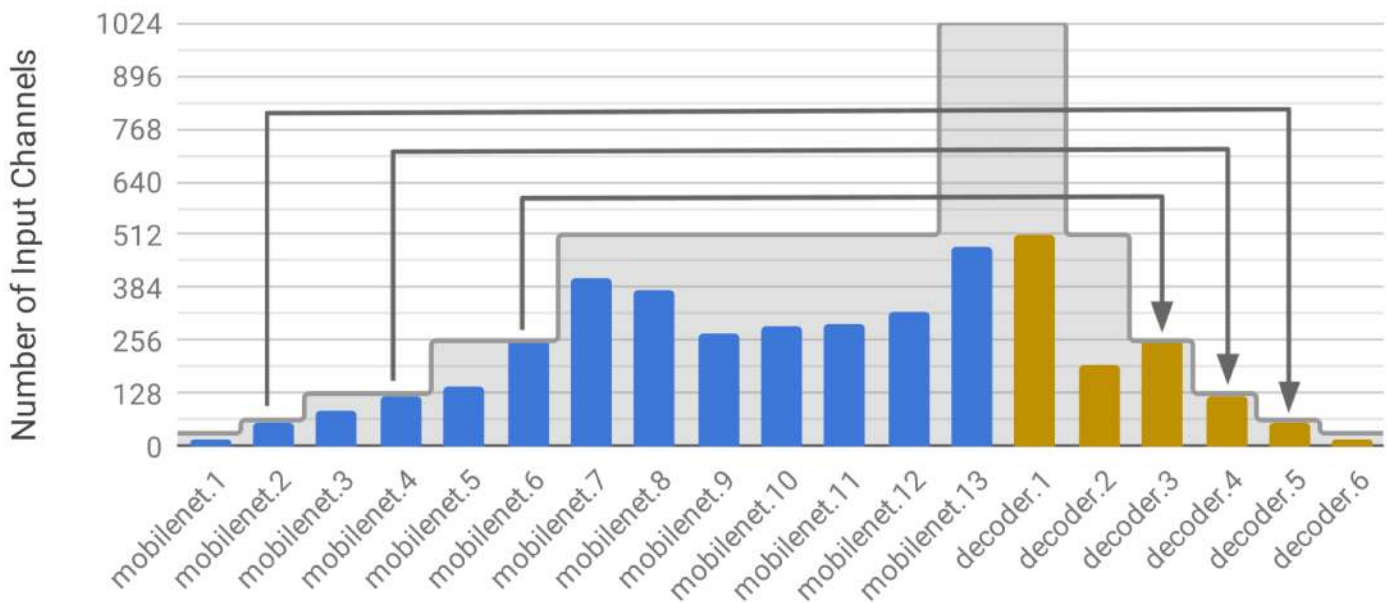


Configuration: Batch size of one (32-bit float)

~40fps on an iPhone

Simplify Network by NetAdapt

	Before Pruning	After Pruning	Reduction
Weights	3.93M	1.34M	2.9×
MACs	0.74G	0.37G	2.0×
RMSE	0.599	0.604	-
δ_1	0.775	0.771	-
CPU [ms]	66	37	1.8×
GPU [ms]	8.2	5.6	1.5×

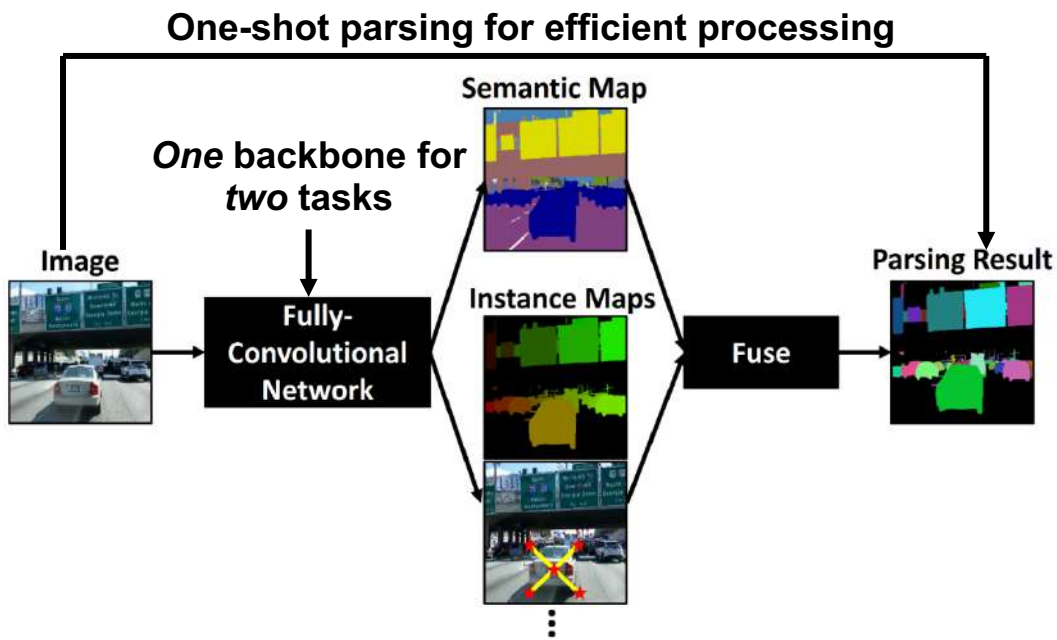


DeeperLab: Single-Shot Image Parser

Joint Semantic and Instance Segmentation (high resolution input image)

Results from Xception

Fully convolutional, one-shot parsing
(bottom-up approach)



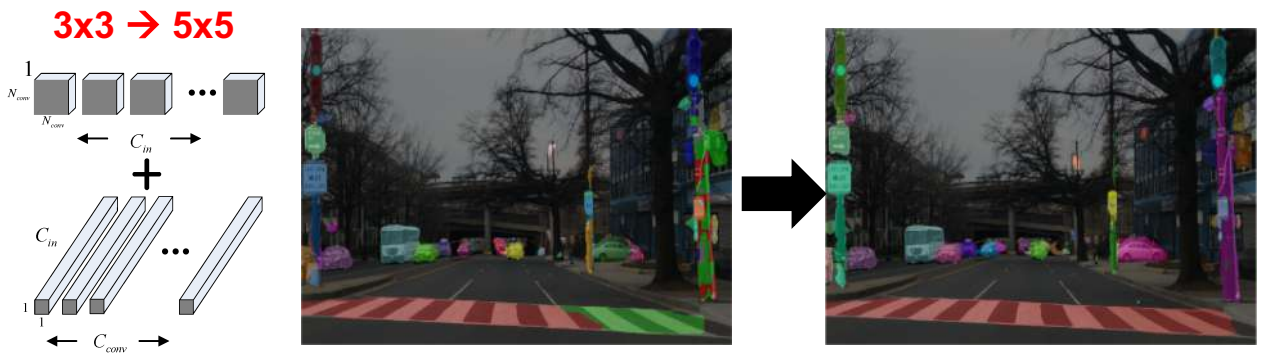
http://deeperlab.mit.edu/

[Yang et al., arXiv 2019]

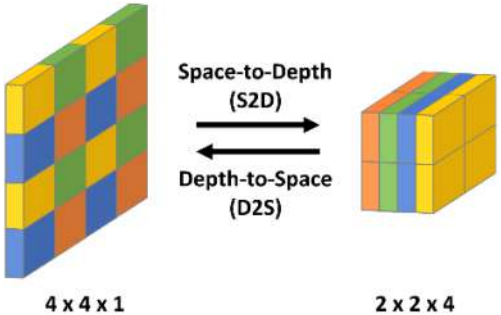
DeeperLab: Efficient Image Parsing

Address memory requirement for large feature map

1 Wide MobileNet: Increase kernel size rather than depth



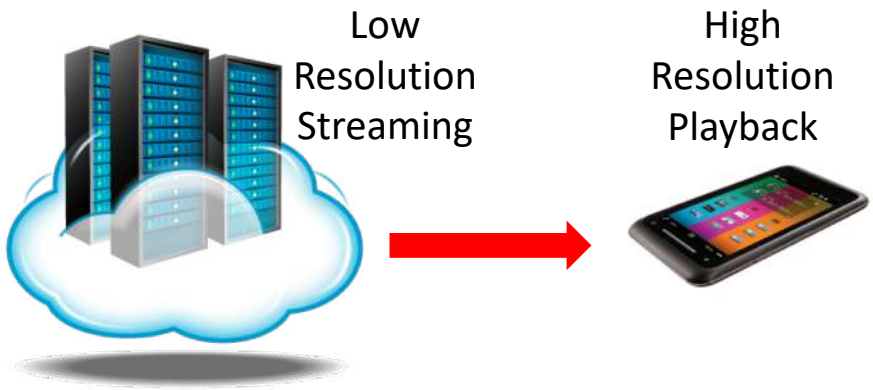
2 Space-to-depth/depth-to-space: Avoid upsampling

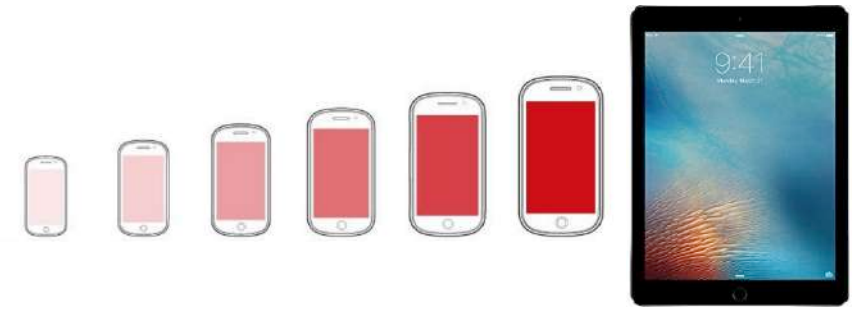


Achieves near real-time 6.19 fps on GPU (V100) with 25.2% PQ and 49.8% PC on Mapillary Vistas dataset

Applications (Beyond DNN Acceleration)

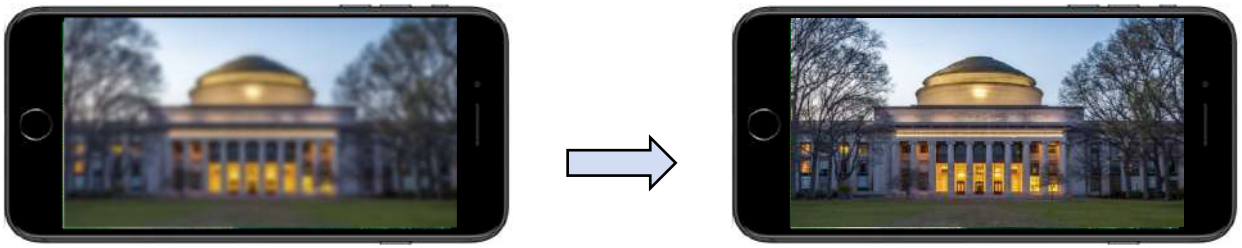
Super-Resolution on Mobile Devices





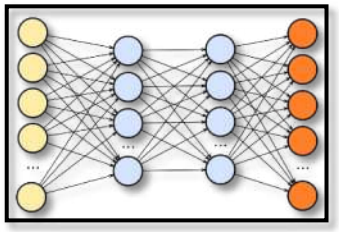
Transmit low resolution for lower bandwidth

Screens are getting larger



Use **super-resolution** to improve the viewing experience of lower-resolution content (*reduce communication bandwidth*)

FAST: A Framework to Accelerate SuperRes



SR algorithm

Compressed video

FAST

SR
15x faster

Real-time

A framework that accelerates **any SR** algorithm by up to **15x** when running on compressed videos

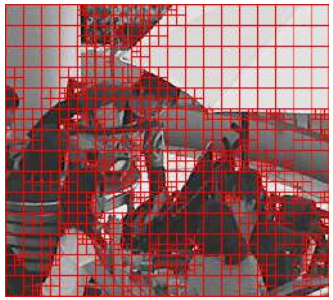
[Zhang et al., CVPRW 2017]

Free Information in Compressed Videos

Compressed video

Decode

Pixels



Block-structure

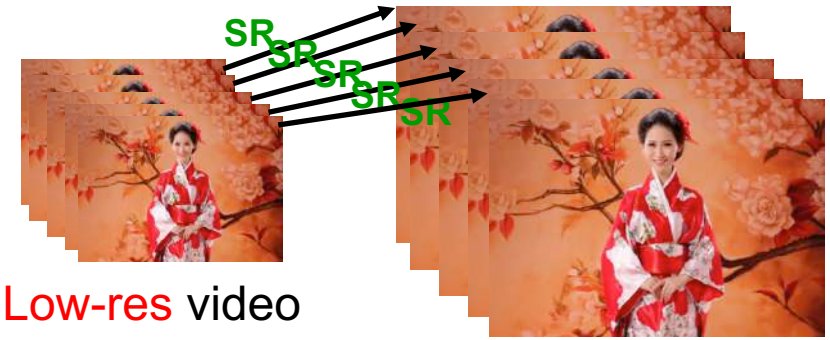
Motion-compensation

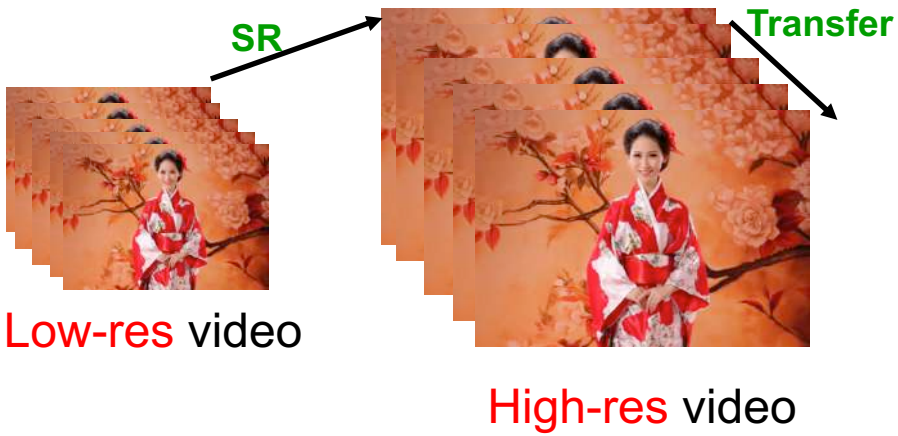
Video as a stack of pixels

Representation in compressed video

This representation can help **accelerate** super-resolution

Transfer is Lightweight





Transfer allows SR to run on only **a subset of frames**

Fractional Interpolation **Bicubic Interpolation**

Skip Flag

The complexity of the transfer is comparable to bicubic interpolation.
Transfer **N** frames, accelerate by **N**

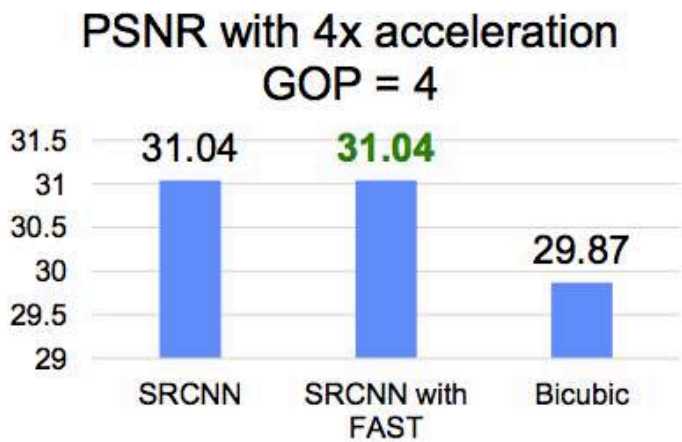
Evaluation: Accelerating SRCNN

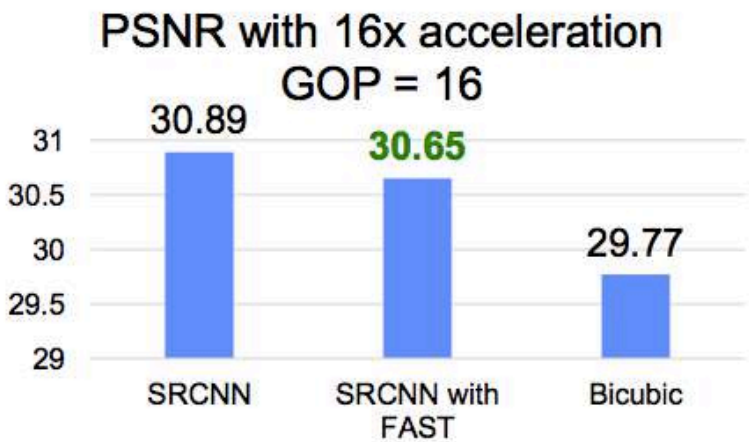
PartyScene

RaceHorse

BasketballPass

Examples of videos in the test set (20 videos for HEVC development)





4 × acceleration with NO PSNR LOSS. 16 × acceleration with 0.2 dB loss of PSNR

Visual Evaluation

SRCNN | **FAST + SRCNN** | **Bicubic**

Look ***beyond*** the DNN accelerator for opportunities to accelerate DNN processing (e.g., structure of data and temporal correlation)

Code released at www.rle.mit.edu/eems/fast

[Zhang et al., CVPRW 2017]

Summary

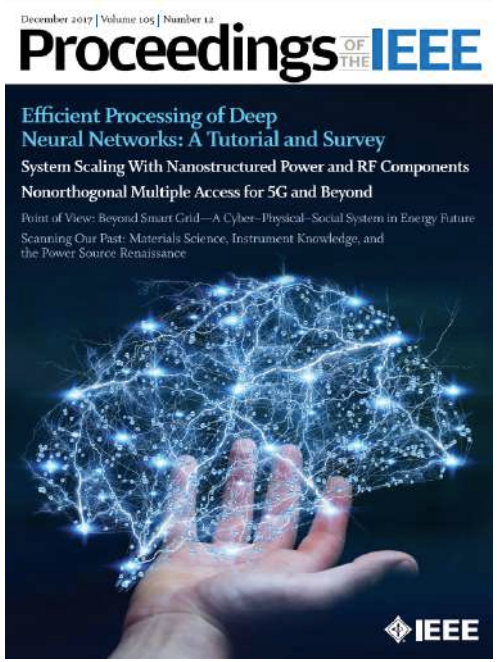
- **DNNs are a critical component in the AI revolution**, delivering record breaking accuracy on many important AI tasks for a wide range of applications; however, it comes at the cost of **high computational complexity**
- **Efficient processing of DNNs** is an important area of research with many promising opportunities for innovation at **various levels of hardware design, including algorithm co-design**
- When considering different DNN solutions it is important to **evaluate with the appropriate workload** in term of both input and model, and recognize that they are **evolving rapidly**.
- It's important to consider a **comprehensive set of metrics** when evaluating different DNN solutions: **accuracy, speed, energy, and cost**

Additional Resources

Overview Paper

V. Sze, Y.-H. Chen, T-J. Yang, J. Emer, "*Efficient Processing of Deep Neural Networks: A Tutorial and Survey,*" **Proceedings of the IEEE**, Dec. 2017

Book Coming Soon!



More info about **Tutorial on DNN Architectures**
http://eyeriss.mit.edu/tutorial.html

MIT Professional Education Course on
"Designing Efficient Deep Learning Systems"
http://professional-education.mit.edu/deeplearning

For updates Follow @eems_mit

http://mailman.mit.edu/mailman/listinfo/eems-news

References

- **Overview on DNN and Popular DNN Models**
  - *Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift," ICML 2015.*
  - ***LeNet**: LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proc. IEEE 1998.*
  - ***AlexNet**: Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." NIPS. 2012.*
  - ***VGGNet**: Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." ICLR 2015.*
  - ***GoogleNet**: Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE conference on computer vision and pattern recognition. CVPR 2015.*
  - ***ResNet**: He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. CVPR 2016.*
  - ***DenseNet**: Huang, Gao, et al. "Densely connected convolutional networks." CVPR 2017*
  - ***Wide ResNet**: Zagoruyko, Sergey, and Nikos Komodakis. "Wide residual networks." BMVC 2017.*
  - ***ResNext**: Xie, Saining, et al. "Aggregated residual transformations for deep neural networks." CVPR 2017.*

References

- **Part 1: Energy-Efficient Hardware for Deep Neural Networks**
  - **Project website:** *http://eyeriss.mit.edu*
  - *Y.-H. Chen, T. Krishna, J. Emer, V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," IEEE Journal of Solid State Circuits (JSSC), ISSCC Special Issue, Vol. 52, No. 1, pp. 127-138, January 2017.*
  - *Y.-H. Chen, J. Emer, V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," International Symposium on Computer Architecture (ISCA), pp. 367-379, June 2016.*
  - *Y.-H. Chen, T.-J. Yang, J. Emer, V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS), June 2019.*
  - *Eyexam: https://arxiv.org/abs/1807.07928*
- **Limitations of Existing Efficient DNN Approaches**
  - *Y.-H. Chen*, T.-J. Yang*, J. Emer, V. Sze, "Understanding the Limitations of Existing Energy-Efficient Design Approaches for Deep Neural Networks," SysML Conference, February 2018.*
  - *V. Sze, Y.-H. Chen, T.-J. Yang, J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," Proceedings of the IEEE, vol. 105, no. 12, pp. 2295-2329, December 2017.*
  - *Hardware Architecture for Deep Neural Networks: http://eyeriss.mit.edu/tutorial.html*

References

- **Transforms for processing on GPU and CPUs**
  - *Lavin, Andrew, and Gray, Scott, "Fast Algorithms for Convolutional Neural Networks," arXiv preprint arXiv:1509.09308 (2015)*
  - *Mathieu, Michael, Mikael Henaff, and Yann LeCun. "Fast training of convolutional networks through FFTs." arXiv preprint arXiv:1312.5851 (2013).*
  - *Cong, Jason, and Bingjun Xiao. "Minimizing computation in convolutional neural networks." International Conference on Artificial Neural Networks. Springer International Publishing, 2014.*
- **Part 2: Co-Design of Algorithms and Hardware for Deep Neural Networks**
  - *T.-J. Yang, Y.-H. Chen, V. Sze, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.*
  - *Energy estimation tool:* http://eyeriss.mit.edu/energy.html
  - *T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, V. Sze, H. Adam, "NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications," European Conference on Computer Vision (ECCV), 2018.* *http://netadapt.mit.edu*
  - *T.-J. Yang, V. Sze, "Design Considerations for Efficient Deep Neural Networks on Processing-in-Memory Accelerators," IEEE International Electron Devices Meeting (IEDM), Invited Paper, December 2019.*
  - *Y. N. Wu, J. S. Emer, V. Sze, "Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs," International Conference on Computer Aided Design (ICCAD), November 2019.* *http://accelergy.mit.edu*
  - *T.-J. Yang, Y.-H. Chen, J. Emer, V. Sze, "A Method to Estimate the Energy Consumption of Deep Neural Networks," Asilomar Conference on Signals, Systems and Computers, Invited Paper, October 2017.*

References

- **Reduced Precision**
  - *Courbariaux, Matthieu, and Yoshua Bengio. "Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1." arXiv preprint arXiv:1602.02830 (2016).*
  - *Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David. "Binaryconnect: Training deep neural networks with binary weights during propagations," NeurIPS, 2015*
  - *Rastegari, Mohammad, et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," ECCV, 2016*
  - *Judd, Patrick, Jorge Albericio, and Andreas Moshovos. "Stripes: Bit-serial deep neural network computing." IEEE Computer Architecture Letters (2016).*
  - *Lee, Edward H., et al. "LogNet: Energy-efficient neural networks using logarithmic computation." 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2017.*
  - *Han, Song, Huizi Mao, and William J. Dally. "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," ICLR, 2016.*

References

- **Exploit Sparsity**
  - *LeCun, Yann, et al. "Optimal brain damage," NIPS, 1989.*
  - *Han, Song, et al. "Learning both weights and connections for efficient neural network," NeurIPS, 2015.*
  - *T.-J. Yang, Y.-H. Chen, V. Sze, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.*
  - *Parashar, Angshuman, et al. "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks." ISCA, 2017*
  - *Han, Song, et al. "EIE: efficient inference engine on compressed deep neural network," ISCA, 2016.*
  - *Y.-H. Chen, T.-J. Yang, J. Emer, V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS), June 2019.*
- **Manual Network Design**
  - *Network in Network: Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." ICLR 2014*
  - ***MobileNet**: Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).*
  - ***ShuffleNet**: Zhang, Xiangyu, et al. "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices." arXiv preprint arXiv:1707.01083 (2017).*
  - *Yu, Fisher, et al. "Deep layer aggregation." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018.*

Reference

- **Neural Architecture Search**
  - *Learning Network Architecture: Zoph, Barret, et al. "Learning Transferable Architectures for Scalable Image Recognition." arXiv preprint arXiv:1707.07012 (2017).*
  - *T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, V. Sze, H. Adam, "NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications," European Conference on Computer Vision (ECCV), 2018. http://netadapt.mit.edu*
- **Hardware In the Loop**
  - *T.-J. Yang, Y.-H. Chen, V. Sze, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.*
  - *Energy estimation tool:* http://eyeriss.mit.edu/energy.html
  - *T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, V. Sze, H. Adam, "NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications," European Conference on Computer Vision (ECCV), 2018. http://netadapt.mit.edu*
  - *T.-J. Yang, V. Sze, "Design Considerations for Efficient Deep Neural Networks on Processing-in-Memory Accelerators," IEEE International Electron Devices Meeting (IEDM), Invited Paper, December 2019.*
  - *Y. N. Wu, J. S. Emer, V. Sze, "Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs," International Conference on Computer Aided Design (ICCAD), November 2019. http://accelergy.mit.edu*
  - *T.-J. Yang, Y.-H. Chen, J. Emer, V. Sze, "A Method to Estimate the Energy Consumption of Deep Neural Networks," Asilomar Conference on Signals, Systems and Computers, Invited Paper, October 2017.*

References

- **Part 3: Applications Beyond Image Classification**
  - *D. Wofk*, F. Ma*, T.-J. Yang, S. Karaman, V. Sze, "FastDepth: Fast Monocular Depth Estimation on Embedded Systems," IEEE International Conference on Robotics and Automation (ICRA), May 2019.* http://fastdepth.mit.edu/
  - *T.-J. Yang, M. D. Collins, Y. Zhu, J.-J. Hwang, T. Liu, X. Zhang, V. Sze, G. Papandreou, L.-C. Chen, "DeeperLab: Single-Shot Image Parser," arXiv, February 2019.* http://deeperlab.mit.edu
  - Z. Zhang, V. Sze, "FAST: A Framework to Accelerate Super-Resolution Processing on Compressed Videos," *CVPR Workshop on New Trends in Image Restoration and Enhancement*, July 2017. www.rle.mit.edu/eems/fast

Fault Injection Attack on Deep Neural Network

Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu
Department of Computer Science & Engineering, The Chinese University of Hong Kong
Shenzhen Research Institute, The Chinese University of Hong Kong
Email: {ynliu, lxwei, boluo, qxu}@cse.cuhk.edu.hk

***Abstract*—Deep neural network (DNN), being able to effectively learn from a training set and provide highly accurate classification results, has become the de-facto technique used in many mission-critical systems. The security of DNN itself is therefore of great concern. In this paper, we investigate the impact of fault injection attacks on DNN, wherein attackers try to misclassify a specified input pattern into an adversarial class by modifying the parameters used in DNN via fault injection. We propose two kinds of fault injection attacks to achieve this objective. Without considering stealthiness of the attack, single bias attack (SBA) only requires to modify one parameter in DNN for misclassification, based on the observation that the outputs of DNN may linearly depend on some parameters. Gradient descent attack (GDA) takes stealthiness into consideration. By controlling the amount of modification to DNN parameters, GDA is able to minimize the fault injection impact on input patterns other than the specified one. Experimental results demonstrate the effectiveness and efficiency of the proposed attacks.**

***Index Terms*—fault injection, neural network, misclassification**

I. Introduction

In recent years, deep neural networks (DNNs) comprising multiple hidden layers have been shown to produce state-of-the-art results on various perception tasks. Given a large training set, they are able to learn effectively and classify unseen input patterns with extremely high accuracy. As a result, DNNs are widely used in many security-sensitive systems, such as malware detection [1], autonomous driving [2], and biometric authentication [3]. The security of DNNs themselves is therefore of great concern and has received lots of attention. Several recent works [4]–[7] revealed the so-called "adversarial example" problem, i.e., a small perturbation on the input is able to fool DNN to generate adversarial output.

As the outputs of a DNN depend on both input patterns and its own internal parameters, theoretically speaking, we could misclassify a given input pattern into any specific class by manipulating some parameters in the DNN. Practically speaking, while DNNs can be implemented in different platforms (e.g., CPU/GPU [8] and dedicated accelerator [9]), the parameters are usually stored in memory and with the development of precise memory fault injection techniques such as *laser beam fault injection* [10]–[12] and *row hammer attack* [13], [14], it is possible to launch effective fault injection attacks on DNNs. To the best of our knowledge, such threats have not been investigated in the literature.

While it is simple to inject faults to modify the parameters used in DNNs, achieving misclassification for a particular input pattern is a challenging task. First, the large amount of parameters in common DNN hinder designing **efficient** attack method with minimum number of injected faults. Second, changing DNN parameters not only misclassifies the targeted input pattern, but also may misclassify some other unspecified input patterns. Hence, the attack may not be **stealthy**. Given such stealthiness is not always required in different attack scenarios, in this work, we propose two kinds of fault injection attacks on DNN.

- We propose *single bias attack* (SBA) to achieve misclassification by only modifying one parameter of DNN. Since the outputs of DNN linearly depend on the biases in the output layer, SBA can be simply implemented by enlarging the corresponding bias in output layer to the corresponding adversarial class. For DNNs with the widely-used rectified linear unit (ReLU) or similar functions as activation function, we find their outputs also linearly depend on the bias in hidden layers, thereby making SBA effective for fault injection in hidden layers. Note that, as such linear relationship is independent from the inputs of DNNs, it is possible to launch real-time SBA without knowing the exact input patterns, which cannot be achieved with other attacks that requires detailed analysis for the inputs.
- We propose *gradient descent attack* (GDA) to force stealthy misclassification, which tries to preserve the classification accuracy on input patterns other than the targeted one. Given the fact that a DNN inherently tolerates small perturbations on its parameters, GDA introduces slight changes to the DNN's parameters to achieve misclassification, thereby mitigating the impact on the other input patterns. We also propose layer-wise searching and modification compression to further improve the stealthiness and the efficiency, by searching the perturbation at finer granularity and removing insignificant part of the obtained perturbation, respectively.

In our experiments, we evaluate the proposed fault injection attack methods with the well-known MNIST and CIFAR10 datasets. We demonstrate that SBA can achieve misclassification by imposing moderate increment on single bias in any layer of a DNN using ReLU activation function. We also show that GBA can force misclassification and only degrades 3.86 percent and 2.35 percent classification accuracy on the two datasets, respectively.

The remainder of the paper is organized as follows. Sec. II presents the preliminaries of this work. In Sec. III, we give an overview of the proposed SBA and GDA attacks. Sec. IV and Sec. V detail these two kinds of attacks, respectively. Experimental results are then shown in Sec. VI. Next, we discuss future works in Sec. VII and introduce related works in Sec. VIII. Finally, we conclude this work in Sec. IX.

II. Preliminaries

In this section, we first introduce the basics of DNN. Next, we present the commonly-used techniques for memory fault injection. At last, we discuss the threat model and attack objective of this work.

A. Basics of Deep Neural Network

Deep neural networks (DNNs) are large neural networks organized into layers of neurons, corresponding to successive representations of the input data, as shown in Figure 1. The first layer and the final layer in the network are called *input layer* and *output layer* respectively, and the layers in the middle are called *hidden layers*. In the context of classification, each neuron in the output layer corresponds to one class in classification task, i.e., m output neurons for m candidate classes. To predict the probability distribution over classes, a softmax function is appended after the output layer and the class having the largest probability is reported as the final predicted class. Formally,

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

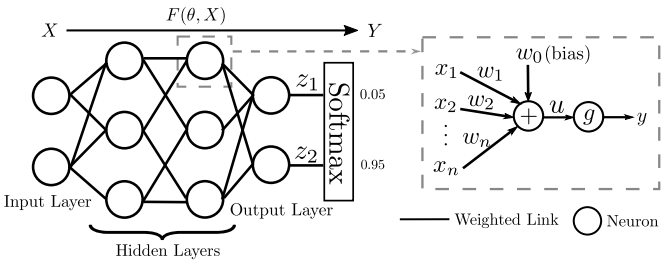


Fig. 1. A 4-layer deep neural network example and the general structure for a neuron.

the softmax function calculates the probability of given input pattern belonging to class i by

$$softmax(i) = \frac{exp(z_i)}{\sum_j exp(z_j)}, \tag{1}$$

where z_j is the output of jth neuron in output layer.

Neurons are connected by links with different *weights* and *biases*, characterizing the strength between neurons. A neuron receives inputs from many neurons in the previous layer, applies its *activation function* on these inputs, and transmits the result to neurons in next layer. Formally, a neuron's output y is given by

$$y = g(u) \quad \text{and} \quad u = \sum_{1 \le i \le n} w_i x_i + w_0, \tag{2}$$

where $x_1, x_2, \ldots, x_n$ are neuron outputs in the previous layer, $w_1, w_2, \ldots, w_n$ are weights on respective links, w_0 represents the bias, and g is the activation function. In this paper, we also use a neuron's name to indicate its output y for simplicity. If we treat the input layer and final predicted probability distribution on candidate classes as two multidimensional vectors X and Y respectively, then the whole DNN is a parameterized function F, and $Y = F(\theta, X)$, where θ represents all the weights and biases in DNN.

In modern neural networks, the rectified linear unit (ReLU) [15]–[17] is the default recommendation for activation function, defined by $g(u) = max\{0, u\}$. Because the derivatives through ReLU are both large and consistent during backpropagation, it makes the training process easier compared to other kinds of activation functions like $sigmoid$ and $tanh$. There are also some varieties of ReLU which use a non-zero slope α when $u < 0$. Uniformly, all these ReLU-like activation functions can be defined as

$$g(u) = \begin{cases} u & \text{u} \ge 0 \\ \alpha u & \text{u} < 0 \end{cases}. \tag{3}$$

For instance, α is 0 for the original ReLU and α is 0.01 for leaky ReLU [20]. In DNNs for classification, to preserve enough information for softmax function, output neurons(i.e., neurons in output layer) usually use identity function, i.e., $g(u) = u$, as the activation function [18], [19]. The neurons in all hidden layers usually use the same type of activation function. We name DNN using ReLU-like activation function in hidden layers as ReLU-like DNN for short.

B. Fault Injection Techniques

Laser beam and row hammer attacks are two common techniques used for injecting faults into memory. Both of them can alter the logic values in memory with high precision, e.g., single bit flip.

Laser beam can inject fault into SRAM. By exposing the silicon under laser beam, a temporary conductive channel is formed in the dielectric, which in turn causes the transistor to switch state

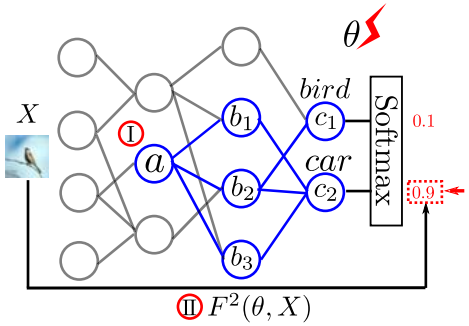


Fig. 2. Fault injection attack on DNN.

in a precise and controlled manner [10]. By carefully adjusting the parameters of laser beam, like its diameter, emitted energy, and impact coordinate, attacker can precisely change any single bit in SRAM [11], [12]. Previously, laser beam was widely used together with differential fault analysis to extract private key of cipher chips [21].

Row hammer can inject fault into DRAM. It exploits the electrical interactions between neighboring memory cells [13]. By rapidly and repeatedly accessing a given physical memory location, a bit in its adjacent location may flip. By profiling the bit flip patterns in DRAM module and abusing the memory manage features, row hammer can reliably flip a single bit at any address in the software stack [14]. The row hammer has been used in many attacks, e.g., cracking the memory isolation in virtualization [22] and obtaining root privileges in Android system [23].

With the above, existing techniques can inject faults into memory with extremely high precision. However, to inject multiple faults, the laser beam fault injection needs to readjust the laser beam setup [12] and row hammer attack needs to relocate the targeted data in memory [23]. Given the cost of readjustment and relocation, minimizing the number of injected faults is essential to make the attack practical.

C. Threat Model and Attack Objective

Threat Model. In this work, we consider an attacker aiming at changing the output of a DNN-based classification system. We consider white-box attack in this work, that is the attacker has the knowledge of targeted DNN's network structure, the benign parameters and low-level implementation details, e.g., parameters' location in memory. Note that, we do not assume the attacker know the data samples for training or testing. We also assume the attacker can modify any parameter in DNN (i.e., any weight or any bias in DNN) by injecting faults into the memory where the parameters are stored. Given existing fault injection techniques can precisely flip any bit of the data in memory, we assume the attacker can modify any parameter in DNN to any adversarial value that is in the valid range of used arithmetic format.

Attack Objective. Our objective is to find the modification on parameters to misclassify a given input pattern into a specific adversarial class, e.g., misclassifying the bird in the given image as a car as described in Fig. 2. Upon achieving misclassification, we also consider the following two issues:

- **Efficiency.** To make the attack practical, we should modify as few parameters as possible. Though DNN can be implemented on different platforms recording parameters in different arithmetic formats, modifying more parameters tends to require

injecting more faults. Hence, minimizing the number of modified parameters can reduce the fault injection cost on all platforms.

- **Stealthiness.** Parameter modification may not only misclassify the given input pattern (e.g., the given image), but also misclassify the rest unspecified input patterns (e.g., all the other images). Keeping the attack stealthy for these input patterns is equivalent to preserving original DNN's classification accuracy on these unspecified input patterns as much as possible.

In various attack scenarios, the adversaries may have different requirements on the efficiency and stealthiness. For instance, when adversaries intend to evade malware detection and crash the system, they do not care about the stealthiness and only concern the efficiency of the attack. When adversaries intend to evade a face recognition system without affecting other users, the stealthiness becomes the main concern for the attack. Motivated by the above, we design respective attack methods.

III. Overview

Given attackers may have different requirements on the efficiency and stealthiness, we propose two attack methods, i.e., SBA and GDA. SBA concentrates on achieving high efficiency during attack, while GDA focuses on achieving high stealthiness. In this section, we introduce the basic idea behind each method and the main challenges.

A. Single Bias Attack

According to Eq. 1, the output neuron with the largest output has the largest probability after softmax function. So we can force the output of DNN to be class i, by making the ith output neuron has the largest output among all output neurons. In DNNs for classification, because identify function is used as activation function in output layer, each output neuron's output linearly depends on its bias. Hence, for any input pattern, we can change DNN's output to class i by enlarging the bias of the ith output neuron.

Moreover, we can attack not only the biases in output layer but also the biases in hidden layers, if the activation function g used in hidden layers is a linear function. Consider the attack scenario indicated by ① in Figure 2, where a is a hidden neuron, c_1 and c_2 are the output neurons, and b_1, b_2, and b_3 are the neurons on the paths from a to output layer. Supposing an attacker increasing a's bias, then the outputs of neurons marked by blue are all affected. According to Eq. 2, if g is linear, a neuron's output y linearly depends on the inputs, weights and bias. In this case, a linearly depends on its bias, b_1, b_2, and b_3 also linearly depend on a's bias, and finally c_1 and c_2 linearly depend a's bias as well. In this case, if the derivate of c_2 w.r.t. a's bias is larger than c_1's, c_2 must be larger than c_1 when a's bias becomes sufficient large, and DNN's output is changed to class 2.

Challenges. The practical activation function used in the hidden layers is non-linear, which makes extending SBA to hidden layers non-trivial. However, we find the output neurons in ReLU-like DNN linearly depend on a bias in hidden layers when the bias is sufficiently large, which we call *one side linear phenomenon* as detailed in Section IV-A. Such property enables us to attack biases in hidden layers.

B. Gradient Descent Attack

Because of the inherent fault tolerance capability of DNN, small perturbation on DNN's parameters will not degrade DNN's classification accuracy significantly. Motivated by this, to achieve high stealthiness, we should minimize the perturbation on parameters during attack.

To find such small perturbation, one naive method is using gradient descent to gradually impose perturbation on parameters. To be specific, we can first initialize θ (i.e., all parameters in DNN) with its benign value, denoted by θ_b, and gradually modify θ in the direction of gradient $\frac{dF^i(\theta,x)}{d\theta}$ to enlarge $F^i(\theta,x)$, where $F^i(\theta,x)$ represents the output probability of class i as indicated by ⓫ in Figure 2. Moreover, we can use L1-norm regulator to restrict the amount of perturbation during searching. Formally, the naive method aims at maximizing the following objective function with gradient descent,

$$J(\theta) = F^i(\theta,x) - \lambda|\theta - \theta_b|. \tag{4}$$

Challenges. Because this naive method operates in a greedy way, the obtained result is usually not optimal. First, we find such naive method trends to modify parameters in layer having few parameters and optimizes the objective function in coarse granularity. To tackle this, we propose layer-wise searching to search perturbation in a hidden layer with abundant tunable parameters, so that we can optimize $J(\theta)$ at finer granularity and achieve better stealthiness, detailed in Section V-A. Second, huge number of parameters are modified in naive method. Because the gradient value for most parameters in DNN is not zero, many parameters are likely to be modified with naive method. To tackle this, we propose modification compression to remove insignificant part in obtained perturbation, so that the number of modified parameters is reduced and the stealthiness is improved, detailed in Section V-B.

IV. Single Bias Attack

In this section, we first introduce the one side linear phenomenon and prove that vulnerable biases widely exist in ReLU-like DNNs. Then, we show the overall flow of SBA. At last, we show SBA is easy to launch, analyze SBA's stealthiness, and discuss how to force DNN generating specific adversarial output without knowing the input pattern.

A. One Side Linear and Sink Class

Let us first give the formal definition of one side linear.

Definition 1. *Given two variables x and y, if there exist two constants ϵ and δ such that $\frac{dy}{dx} = \delta$ when $x > \epsilon$, we say y is* **one side linear** *to x and the one side linear slope is δ.*

If multiple variables $y_1, y_2, \ldots, y_n$ are all one side linear to variable x with different slopes, then $y_1, y_2, \ldots, y_n$ will change at different rates when x increases. If the increment on x is larger enough, the variable in $y_1, y_2, \ldots, y_n$ increasing at the fastest rate would finally become the largest one, which is formally proved by Theorem 1.

Theorem 1. *Given variables $y_1, y_2, \ldots, y_n$ which are all one side liner to variable x with slopes $\delta_1, \delta_2, \ldots, \delta_n$ respectively. If $\delta_n > \delta_i$ for $\forall i \neq n$, there must exist a constant ϵ_{sink} such that $y_n > y_i$ for $\forall i \neq n$ when $x > \epsilon_{sink}$.*

Proof. According to Definition 1, for any $i \in [1, n]$, there must exist a constant ϵ_i such that $\frac{dy_i}{dx} = \delta_i$ when $x > \epsilon_i$. Suppose $\epsilon_{max} = max\{\epsilon_1, \ldots, \epsilon_n\}$, and the values of $y_1, y_2, \ldots, y_n$ are $v_1, v_2, \ldots, v_n$ respectively when $x = \epsilon_{max}$. When $x > \epsilon_{max}$, we have $y_i = v_i + \delta_i(x - \epsilon_{max})$.

In this case, to make $y_n > y_i$ for $\forall i \neq n$, we have

$$v_n + \delta_n(x - \epsilon_{max}) > v_i + \delta_i(x - \epsilon_{max})$$
$$x > \epsilon_{max} + \frac{v_i - v_n}{(\delta_n - \delta_i)}.$$

So when $x > \epsilon_{sink} = max\{\epsilon_{max} + \frac{v_i - v_n}{(\delta_n - \delta_i)} | i \in [1, n-1]\}$, $y_n > y_i$ for $\forall i \neq n$. With above, Theorem 1 is proved. □

In the context of ReLU-like DNN, we have two observations related to one side linear phenomenon: 1) each neuron's output is one side linear to its bias, given by Lemma 1, and 2) if a neuron's all inputs are one side linear to certain bias, then this neuron's output is also one side linear to the bias, given by lemma 2.

Lemma 1. *If h is a neuron in the hidden layers of ReLU-like DNN, then h is one side linear to its bias m with one side linear slope 1.*

Proof. According to Eq. 2, we have $h = g(\sum_{1 \leq i \leq n} w_i x_i + m)$. Given g is ReLU-like function defined in Eq. 3, if $m > |\sum_{1 \leq i \leq n} w_i x_i|$, $h = \sum_{1 \leq i \leq n} w_i x_i + m$. Hence, h is one side linear to m and the slope is 1. □

Lemma 2. *Given a neuron h and a bias m in ReLU-like DNN, if all h's inputs $x_1, x_2, \ldots, x_n$ are one side linear to m with slopes $\delta_1, \delta_2, \ldots, \delta_n$ respectively, then h is also one side linear to m with slope*

$$g\left(\sum_{1 \leq i \leq n} w_i \delta_i\right),$$

where g is the activation function and w_i is the weight corresponding to i^{th} input.

Proof. According to Definition 1, for any $i \in [1, n]$, there must exist a constant ϵ_i such that $\frac{dx_i}{dm} = \delta_i$ when $m > \epsilon_i$. Suppose $\epsilon_{max} = max\{\epsilon_1, \ldots, \epsilon_n\}$ and $u = \sum_{1 \leq i \leq n} w_i x_i + w_0$, where w_0 is h's bias. When $m > \epsilon_{max}$, we have

$$\frac{du}{dm} = \frac{d\left(\sum_{1 \leq i \leq n} w_i x_i + w_0\right)}{m} = \sum_{1 \leq i \leq n} w_i \cdot \frac{dx_i}{dm} = \sum_{1 \leq i \leq n} w_i \delta_i.$$

Suppose $u = u'$ when $m = \epsilon_{max}$. When $m > \epsilon_{max}$, we have

$$u = u' + (m - \epsilon_{max}) \sum_{1 \leq i \leq n} w_i \delta_i.$$

If g is identity function in output layer, $h = u$. Apparently, h is one side linear to m with slope $g(\sum_{1 \leq i \leq n} w_i \delta_i)$. If g is ReLU-like function in hidden layer, we discuss the case that $\sum_{1 \leq i \leq n} w_i \delta_i$ is positive, negative and zero separately.

When $\sum_{1 \leq i \leq n} w_i \delta_i > 0$ and $m > \epsilon_{max} + |\frac{u'}{\sum_{1 \leq i \leq n} w_i \delta_i}|$, we have $u = u' + (m - \epsilon_{max}) \sum_{1 \leq i \leq n} w_i \delta_i > 0$ and $h = u$. Then h is one side linear to m and the slope is $g(\sum_{1 \leq i \leq n} w_i \delta_i)$.

When $\sum_{1 \leq i \leq n} w_i \delta_i < 0$ and $m > \epsilon_{max} + |\frac{u'}{\sum_{1 \leq i \leq n} w_i \delta_i}|$, we have $u = u' + (m - \epsilon_{max}) \sum_{1 \leq i \leq n} w_i \delta_i < 0$ and $h = \alpha u$. So h is one side linear to m and the slope is $g(\sum_{1 \leq i \leq n} w_i \delta_i)$

At last, when $\sum_{1 \leq i \leq n} w_i \delta_i = 0$, apparently h is also one side linear to m and the slope is also $g(\sum_{1 \leq i \leq n} w_i \delta_i)$.

With above, we show h is one side linear to m in all cases and the slope is $g(\sum_{1 \leq i \leq n} w_i \delta_i)$ □

In fact, conclusion similar to Lemma 2 holds for other types of activation function. For instance, the neuron h in widely used max pooling layer simply outputs its largest input, i.e., $h = max(x_1, x_2, \ldots, x_n)$. Suppose $x_1, x_2, \ldots, x_n$ are all one side linear to a bias m and x_n has the largest one side linear slope. According to Theorem 1, there must exist ϵ_{sink} such that x_n is consistently larger than $x_1, x_2, \ldots, x_{n-1}$ when $m > \epsilon_{sink}$, which means $h = x_n$ in this case. So h is also one side linear to m and the slope is the same as x_n's.

With above, we show the one side linear property can propagate from previous layer to next layer. By induction, the output neurons are one side linear to any bias.

Theorem 2. *In ReLU-like DNN, for a bias m in hidden layers, every output neuron is one side linear to m.*

Proof. Suppose the DNN has N layers and m is in the ith layer. The neurons in the same layer are independent. Given a neuron in the ith hidden layer, it is either independent with m or using m as its bias. A neuron independent with m is one side linear to m with slope 0, and a neuron using m as bias is one side linear to m given Lemma 1.

With above, all the neurons in i^{th} layer are one side linear to m. Because the neurons in $(i+1)^{th}$ layer only depends on neurons in i^{th} layer, all neurons in $(i+1)^{th}$ layer are also one side linear to m according to Lemma 2. By induction, all neurons in the output layer are one side linear to m. □

Sink Class. With Theorem 1 and Theorem 2, if one output neuron has the largest one side linear slope w.r.t. a given bias among all the output neurons, increasing this bias would finally make the DNN's output converge at the class indicated by this output neuron. We use **sink class** to represent the output class at which DNN's output converges when a bias increases. During proving Theorem 2 by induction, we find the one side line slope of a output neuron w.r.t. a bias is independent with the input pattern. Hence, one bias' sink class is the same for different input patterns.

B. Overall Attack Flow

The overall flow of single bias attack consists of two steps, as shown in Figure 3(a). First, we profile the sink class for each bias for the given DNN. Next, according to the specific adversarial mapping, we only need to choose one bias whose sink class is the same as the targeted class in the adversarial mapping and find the required increment on this bias to saturate DNN's output.

Profiling Sink Class. To obtain the sink class for a bias, we only need each output neuron's one side linear slope w.r.t. this bias and check whether unique largest slope exists. We can calculate the slope for each output neuron w.r.t. a given bias m by induction. At first, we need to initialize the slopes for all neurons in i^{th} layer where m is located. If a neuron uses m as its bias, we set the slope for this neuron to be 1, otherwise it is 0. Then we can calculate the slopes layer by layer according to Lemma 2. Finally, we can obtain the one side linear slope for each output neuron. If unique largest slope exists in the output layer, we can determine the sink class for m.

Saturating DNN. To achieve the adversarial mapping, we can choose any bias m whose sink class is the same as the targeted class in the specific adversarial mapping. Then to make DNN output the m's sink class, we only need to modify m so that m is larger the ϵ_{sink} value defined in Theorem 1. We can determine the exact ϵ_{sink} value for a given input pattern and selected bias m, by scanning m values and monitoring when DNN's output converges. Given we can check the ϵ_{sink} for each candidate bias, we can also use the bias with the smallest ϵ_{sink} during attack.

C. Discussion

During practical attack, we do not need to precisely modify the bias, i.e., enlarging the bias beyond ϵ_{sink} is sufficient. To achieve this, we may only need to inject several single bit faults on the most significant bits of the bias. For instance, when floating point format

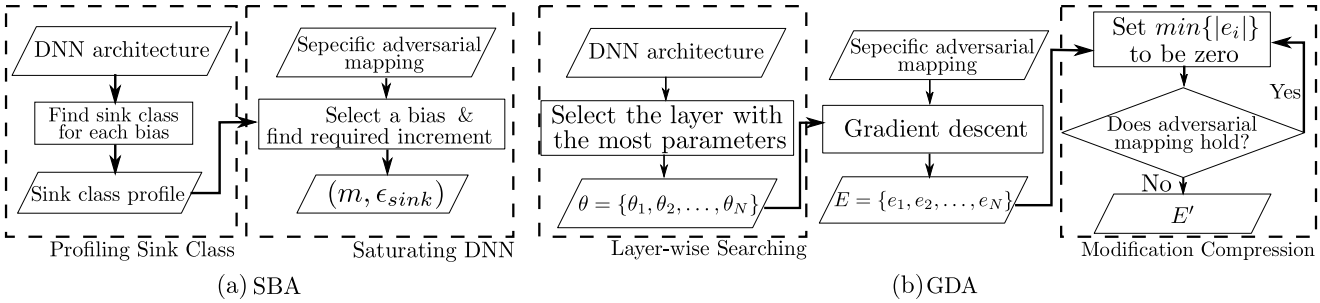


Fig. 3. The overall flow for (a)SBA and (b)GDA.

is used to store parameter, we can efficiently enlarge the bias by magnitudes via making its sign bit positive and setting several most significant bits in the exponent field.

The stealthiness of single bias attack depends on the ϵ_{sink} value for the input pattern. Suppose an attacker can precisely modify the bias to be slightly larger than the ϵ_{sink}. Different input patterns may require different ϵ_{sink} values to saturate DNN's output when attacking the same bias. When misclassifying a input pattern with $\epsilon_{sink} = k$, the outputs for other input patterns with $\epsilon_{sink} < k$ on this bias must converge and the outputs for input patterns with $\epsilon_{sink} > k$ may be also affected although not converge. Hence, the accuracy degradation is larger when attacking input pattern with larger ϵ_{sink} on a bias. To mitigate this, one possible solution is to recover the DNN to benign state with additional faults in addition to the few faults injected during SBA.

Note that, even without knowing the exact input pattern, it is still possible to force DNN generating specific adversarial output with SBA. Because the sink class of a bias is independent from input patterns, we could saturate DNN by modifying the bias to be a very large valid value allowed by the arithmetic format used. When it is larger than the required ϵ_{sink} for an input pattern, we can successfully change the DNN's output. Such attack can be launched at real-time (without necessarily conducting input pattern analysis), leading to possible denial-of-service.

V. Gradient Descent Attack

The overall flow of gradient descent attack consists of three steps, as shown in Figure 3(b). At first, we determine the set of parameters to be updated in gradient descent. Instead of updating all the parameters in DNN and searching the modification global-wise, we search modification within a single layer, i.e., layer-wise searching. Next, we optimize Eq. 4 with gradient descent to find the modification on each selected parameter, denoted by $E = \{e_1, e_2, \ldots, e_N\}$. At last, we repeatedly replace the smallest element in E with zero until the final parameter modification E' is obtained, i.e, modification compression.

In the following two subsections, we discuss layer-wise searching and modification compression in detail.

A. Layer-wise Searching

We found the magnitude of parameter's gradient is not uniformly distributed among layers, and parameters in layer having fewer parameters trend to have gradient of larger magnitude. For instance, Figure 4 shows the average magnitude of parameter's gradient and parameter count in each layer for the MNIST model detailed in Section VI-A, and `layer2` with the fewest parameters has the largest gradient. Because gradient descent modifies each parameter according

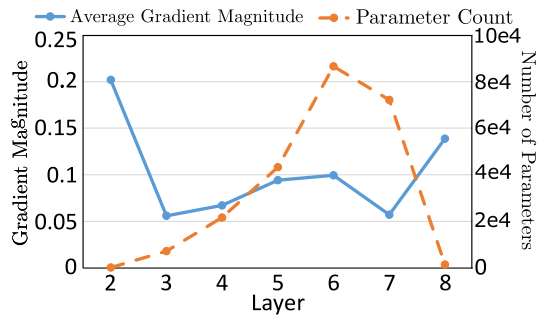


Fig. 4. The distribution of parameter's gradient and parameter count.

to its gradient, parameter with gradient of larger magnitude receives larger modification during searching. As a consequence, the naive method which is supposed to find optimal modification global-wise in fact focuses on modifying the few parameters in the smallest layer, e.g., `layer2` in Figure 4. In this way, the naive method optimizes the objective function by adjusting the mapping F in coarse granularity and the obtained solution is not optimal.

To tackle this problem, we propose layer-wise searching. To be specific, instead of finding the modification global-wise, we manually restrict the θ in Eq. 4 to be the set of all parameters within one layer. Intuitively, a layer with more parameters allows us to adjust the mapping F at finer granularity and optimize the objective function better, i.e., introducing smaller perturbation. While the naive global-wise searching in fact focuses on adjusting the smallest layer, we can conduct layer-wise searching on the layer with the most parameters to search the solution at the finest granularity. In Section VI-C, we demonstrate conducting layer-wise searching on any layer, except the smallest layer, can achieve higher stealthiness than the global-wise searching.

B. Modification Compression

Because gradient search is a greedy method, the obtained perturbation list E is not optimal even with our layer-wise searching technique, which implies some perturbations in E are redundant. Motivated by this, modification compression aims at removing the insignificant part of E to further reduce the overall perturbation, that is replacing some non-zero elements in E with zero.

Considering changing E too much may destroy the required adversarial mapping, we achieve the replacement by iteration as shown in Figure 3(b). At each iteration step, we only replace the element with smallest absolute value in E to be zero and check whether the adversarial mapping is still preserved. If it is preserved, we continue

TABLE I. The architectures of the MNIST model and the CIFAR model.

Layer	MNIST	CIFAR
1	28*28 image	32*32 RGB image
2	3*3 conv. 20 $tanh$	3*3 conv. 96 ReLU
3	3*3 conv. 40 $tanh$ 2*2 max pooling	3*3 conv. 96 ReLU 2*2 max pooling
4	3*3 conv. 60 $tanh$	3*3 conv. 192 ReLU
5	3*3 conv. 80 $tanh$ 2*2 max pooling	3*3 conv. 192 ReLU 2*2 max polling
6	3*3 conv. 120 $tanh$	3*3 conv. 192 ReLU
7	fully connect. 150 $tanh$	1*1 conv. 192 ReLU
8	fully connect. 10 identity	1*1 conv. 10 identity global average pooling
softmax function		

to explore further replacement. Otherwise, only previously found replacements are finally used.

With modification compression, we can not only reduce the number of modified parameters but also improve the stealthiness, because replacing the elements in E with zero reduces the amount of perturbation imposed.

VI. Experimental Results

A. Experimental Setup

Dataset. All the experiments are performed on two machine learning datasets: MNIST and CIFAR10. The MNIST dataset is a collection of 70000 black-white images of handwritten digit, where each pixel is encoded as a real number in $[0, 1]$. The classification goal is to determine the digit written in the given image. So the candidate classes range from 0 to 9. The CIFAR10 dataset is a collection of 60000 nature color images in RGB format, where the value of each pixel on each channel is encoded as an integer in $[0, 255]$. The images are to be classified in one of the 10 mutually exclusive classes, e.g., airplane, bird, and cat. We also label these 10 classes from 0 to 9 for short.

DNN Model. For each dataset, we implement one DNN model, detailed in Table I. The MNIST model from [18] is an 8 layer DNN for MINST dataset, which uses $tanh$ as activation function. The CIFAR model from [19] is an 8 layer DNN for CIFAR10 dataset, which uses ReLU as activation function. In each model, the `Layer1` and `Layer8` are the input layer and output layer respectively. The MNIST model and CIFAR model can achieve 99.06% and 84.01% classification accuracy with benign parameter respectively, which are comparable to the state-of-the-art results.

Platform. All the experiments are conducted on a machine equipped with an Intel Xeon E5 CPU and a NVIDIA TitanX GPU. We implement the DNN models with Theano [8] library, and Theano uses default float32 precision to store the parameters and intermediate results.

B. Evaluation for SBA

In this section, we evaluate SBA on the CIFAR model which is a ReLU-like DNN. We first demonstrate biases having sink class widely exist in DNN. Then we show how DNN's output converges as we increase a bias. At last, we evaluate the magnitude of ϵ_{sink} required to saturate the DNN and the stealthiness of the attack.

At first, we found every bias in the CIFAR model had a sink class, and the distribution of sink class within each layer is shown in Figure 5(a). We observe that every layer in the CIFAR model can provide all the ten kinds of sink classes. Hence, once we can inject fault into one layer, we can achieve any specific maliciously mapping.

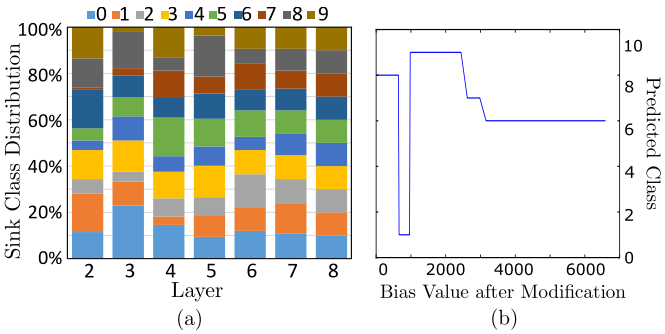


Fig. 5. (a) Sink class distribution in the CIFAR model, where each color represents one kind of sink class, and (b) the convergence of CIFAR model's output as a bias increases.

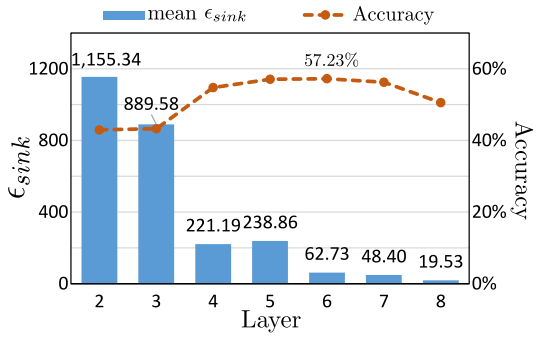


Fig. 6. The mean ϵ_{sink} in each layer and the corresponding classification accuracy after attack.

Next, to verify CIFAR model's output would converge at a bias's sink class as we increase the bias, we test every bias in the CIFAR model with about 100 randomly selected input patterns. In all the cases, the CIFAR model's output successfully converges at the expected sink class. Figure 5(b) shows the convergence procedure of CIFAR's output in a random selected case, which can be divided into two stages. In the early stage, the model's output oscillates when the bias increases, because the classification function intended to be learned by the DNN is non-linear. Then, as we increase the bias beyond the ϵ_{sink} value, the output converges at the sink class of the bias.

At last, to show the increment on bias required by SBA is practical, we examine the biases for each layer individually. For each layer, we randomly select about 1000 input patterns for misclassification, and we record the minimal ϵ_{sink} for each pattern when using biases in this layer. In Figure 6, the bar chart shows the mean of such minimal ϵ_{sink} values for each layer, and the dashed curve plots the average classification accuracy after attack where we modify the bias slightly larger than the ϵ_{sink}. Given Figure 6, we have two observations. First, the average ϵ_{sink} for each layer is definitely within the valid range of the *float32* precision, which can represent number as large as $3e38$. Especially for `Layer6` $\sim$ `Layer8`, the average ϵ_{sink}s are smaller than 70, which are exploitable on platforms using simpler arithmetic format, e.g., 16-bit fixed-point number with 8-bit fraction. Second, we find the highest accuracy 57.23% is achieved by attacking `Lyaer6`. However, attacking `Layer6` in fact imposes larger perturbation on the bias compared to attacking `Layer8`, given the magnitude of every bias's benign value in our CIFAR model is smaller than one. One possible reason is that, the gradient of the output w.r.t. a bias in `Layer6` is smaller than that for `Layer8`, similar to the case indicated by Figure 4. Hence, although attacking `Layer6` imposes

TABLE II. The classification accuracy(CA) after attack and the number of modified parameters(# of MP) during attack, where 'LW+digit' represents layer-wise searching on corresponding layer, 'MC' is short for modification compression, and '§' indicates the results for naive gradient descent method.

	MNIST				CIFAR			
	CA		# of MP		CA		# of MP	
	w/o MC	MC	w/o MC	MC	w/o MC	MC	w/o MC	MC
LW 2	46.38%	59.89%	200	19	12.98%	25.06%	2334	283
LW 3	56.22%	68.62%	7240	221	12.98%	54.54%	57009	1354
LW 4	58.80%	84.93%	21660	1077	25.34%	76.45%	129759	697
LW 5	46.07%	90.44%	43280	1215	23.39%	73.73%	195502	2321
LW 6	65.23%	95.20%	86520	2345	11.68%	81.66%	115127	198
LW 7	89.88%	97.01%	72150	5734	13.87%	80.57%	19109	43
LW 8	95.12%	96.86%	1439	125	13.02%	80.32%	1147	2
Global-wise	26.68%(§)	63.70%	232559(§)	1170	10.00%(§)	50.97%	519691(§)	425

larger perturbation on a bias, it changes DNN's output less.

C. Evaluation for GDA

In this section, we evaluate GDA on both MNIST model and CIFAR model. We examine conducting layer-wise searching on each layer, while modification compression is present or absent. Table II shows the number of modified parameters and the classification accuracy after attack in all the cases.

At first, to evaluate the effectiveness of layer-wise searching, we compare the data within each column in Table II. First, by comparing the data in third column and seventh column respectively, we observe layer-wise searching on any layer except `Layer2` can achieve higher accuracy than the global-wise searching. According to Table I, in both models, any layer in `Layer3`∼`Layer8` has more parameters than `Layer2`. Hence, attacking a layer with abundant parameters is more stealthy than simply using global-wise searching. Second, by comparing the data in fifth column and ninth column respectively, we observe layer-wise searching on layers close to input layer or output layer modifies fewer parameters than layers in the middle of DNN. This is because the layers close to the two ends have much fewer parameters than layers in the middle according to Table I, hence fewer parameters are modified during gradient descent.

Next, to evaluate the effectiveness of modification compression, we examine the data within each row in Table II. On the one hand, we observe, no matter layer-wise searching or global-wise searching is used, conducting modification compression can always significantly reduce the number of modified parameters. Without modification compression, almost all tunable parameters are modified in each case. On the other hand, we observe modification compression can always improve the classification accuracy, which also demonstrates minimizing perturbation on parameters indeed helps improve the stealthiness.

Finally, by conducting layer-wise searching on `Layer6`, the largest layer containing the most parameters in both models, together with modification compression, GDA can achieve classification accuracy as high as 95.20% and 81.66% for each model, which only degrades the benign accuracy by 3.86 percent and 2.35 percent respectively. Compared to the naive method discussed in Section III-B, whose results are indicated by '§' in Table II, GDA can averagely reduce the number of modified parameters by 99.50%, and improve the classification accuracy by 67.97 percent.

D. SBA VS. GDA

We compare SBA and GDA based on the results for CIFAR model in Figure 6 and Table II. First, we observe SBA is more efficient than GDA, because SBA only modifies one parameter and GDA modifies 198 parameters when attacking `Layer6`. Second, by examining the accuracy curve in Figure 6, we observe GDA is more stealthy than SBA, given GDA can achieve 81.66% accuracy while SBA's average accuracy is below 60%. One possible reason is that, GDA imposes small perturbations on multiple parameters instead of imposing large perturbation on a single parameter like SBA, and the total perturbation on parameters imposed by GDA is smaller than SBA.

VII. Discussion and Future Works

In this section, we discuss potential countermeasures to the proposed attacks and possible future work.

We can detect whether DNN suffers fault injection attack by checking whether the stored parameters are modified. To achieve this, we can apply error detection code (EDC) on parameters, e.g., parity check. Because DNNs generally contain a very large number of parameters, however, the cost of checking all parameters' status via EDC could be prohibitively high. As a direction of future study, one can investigate the vulnerabilities of different parameters under attack and only protect those more vulnerable ones with EDC, which can achieve a good balance between fault detection overhead and fault detection capability.

We can mitigate SBA's threat on ReLU-like DNN by optimizing DNN's architecture. SBA relies on the one side linear property of ReLU-like function to propagate the large increment on internal bias to DNN's final output, so that the output neuron with the largest slope would finally become the largest one. However, such large value propagation is not possible for activation function like *tanh*, because *tanh*'s output is always between -1 and 1. Hence, as a direction of future work, one can study how to allocate ReLU-like and other types of activation functions in the same DNN, so that we can not only preserve ReLU-like function's advantage in simplifying training procedure but also mitigate SBA.

We may reduce the required increment on a bias in SBA by modifying more biases instead of one, which may also improve the stealthiness of SBA. In Section VI-D, we found GDA can achieve higher stealthiness than SBA by imposing smaller perturbations on multiple parameters. Similarly, we may enhance SBA by imposing small perturbations on multiple biases instead of one large perturbation on single bias. As a direction of future work, one can study how to select the set of multiple biases and allocate perturbations on these biases to efficiently saturate DNN's output.

It is also possible to attack DNN with low-precision fault injection techniques. With low-precision fault injection, attacker cannot modify the parameter precisely. For instance, the bits maybe flipped with uncertainty within the fault-affected area. Because SBA doesn't need to modify the bias exactly and increasing the bias beyond its ϵ_{sink} is sufficient, SBA may still succeed in this case if we can guarantee

the bias is increased during attack, e.g., injecting faults into a zero byte. As a direction of future study, one can investigate how to attack DNN with low-precision fault injection techniques in practice.

VIII. Related Works

In this section, we briefly discuss the related works on adversarial example and techniques for improving DNN's fault-tolerance capability against random faults.

The state-of-the-art neural networks are vulnerable to adversarial examples [4]. That is, small but intentionally perturbation on input forces neural network to provide adversary output, and such perturbation for an input pattern can be crafted by methods like L-BFGS [4] and fast-gradient-sign [5]. Given adversarial example and fault injection attack proposed in this work target on DNN's input and internal parameters respectively, adversarial example is orthogonal to our proposed attacks. To defeat adversarial example problem, recent countermeasures aim at improving the generalization capability of neural network by manipulating DNN's training procedure. For instance, Papernot *et al.* [6] proposed to train the same neural network twice and the soft labels used in the second train process are able to improve the generalization capability. Gu and Rigazion [7], and Goodfellow *et al.* [5] proposed to regularize the objective function in training procedure to manually restrict the predicted class for data points around training samples.

Though no previous works have studied fault injection attack on DNN, there are many works improving DNN's fault-tolerance capability against random faults on the weights. These works can be mainly divided into three groups, i.e., training with artificial fault [24], [25], neuron duplication [26], [27], and weight restriction [28], [29]. Training with artificial fault intentionally injects faults into DNN during training process, so that obtained DNN can perform correctly even in faulty cases [24], [25]. However, the cost of enumerating all faulty cases is prohibitive when the number of faulty cases is exponentially larger, like enumerating multi-bit faults. Hence, such method is inefficient for defeating SBA and GDA as well, where the number of exploitable faulty cases is large. Neuron duplication improves the redundancy of DNN by duplicating internal neurons and scaling down corresponding weights [26], [27]. Such method is specially designed for stuck@0 fault. Because the magnitude of each weight is scaled down, the perturbation imposed by stuck@0 fault is reduced and expected to be tolerated. However, adversaries can intentionally change a weight's value and control the amount of perturbation introduced, so such method is inefficient for malicious faults. At last, weight restriction determines a range to which weights should belong during training, and any weight being outside the range is forced to be its upper limit or lower limit [28], [29]. In this way, weight restriction can mitigate the interference from fault imposing large perturbation on a weight. Intuitively, weight restriction can mitigate SBA, but it is less effective for defeating GDA. This is because SBA need to impose large perturbation on one parameter, while GDA imposes small perturbations on multiple parameters.

IX. Conclusion

In this paper, we present two kinds of fault injection attacks on DNN. SBA is able to achieve misclassification by modifying only one bias in the network. By combining DNN's fault tolerance capability with the proposed layer-wise searching and modification compression techniques, GDA is able to achieve high stealthiness and efficiency at the same time. Experimental results demonstrate the effectiveness of the proposed attacks on DNN.

Acknowledgement

This work is supported in part by Project No. 61432017 and Project No. 61532017 by National Natural Science Foundation of China (NSFC), in part by Huawei Technologies Co. Ltd., and in part by Shanghai Inbestech Co. Ltd.

References

[1] G. E. Dahl, *et al*. Large-scale malware classification using random projections and neural networks. In *Proc. of International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[2] D. C. Ciresan, *et al*. Multi-column deep neural network for traffic sign classification. In *Neural Networks*, 2012.

[3] Y. Taigman, *et al*. Deepface: Closing the gap to human-level performance in face verification. In *Proc. of Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[4] C. Szegedy, *et al*. Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199, 2013.

[5] I. J. Goodfellow, *et al*. Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572, 2014.

[6] N. Papernot, *et al*. Distillation as a defense to adversarial perturbations against deep neural networks. In *Proc. of Symposium on Security and Privacy (SP)*, 2016.

[7] S. Gu and L. Rigazio. Towards deep neural network architectures robust to adversarial examples. arXiv preprint arXiv:1412.5068, 2014.

[8] J. Bergstra, *et al*. Theano: A CPU and GPU math compiler in python. In *Proc. of Python in Science Conf*, 2010.

[9] T. Chen, *et al*. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[10] A. Barenghi, *et al*. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. In *Proc. of the IEEE*, 2012.

[11] B. Selmke, *et al*. Precise laser fault injections into 90 nm and 45 nm sram-cells. In *Proc. of Smart Card Research and Advanced Applications (CARDIS)*, 2015.

[12] M. Agoyan, *et al*. How to flip a bit?. In *Proc. of International On-Line Testing Symposium (IOLTS)*, 2010.

[13] Y. Kim, *et al*. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proc. of International Symposium on Computer Architecture (ISCA)*, 2014.

[14] K. Razavi, *et al*. Flip feng shui: Hammering a needle in the software stack. In *Proc. of USENIX Security Symposium*, 2016.

[15] I. Goodfellow, *et al*. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] K. Jarrett, *et al*. What is the best multi-stage architecture for object recognition? In *Proc. of International Conference on Computer Vision (ICCV),* 2009.

[17] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proc. of International Conference on Machine Learning (ICML)*, 2010.

[18] D. C. Ciresan, *et al*. Flexible, high performance convolutional neural networks for image classification. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[19] J. T. Springenberg. *et al*. Striving for simplicity: The all convolutional net. arXiv preprint arXiv:1412.6806, 2014.

[20] A. L. Maas, *et al*. Rectifier nonlinearities improve neural network acoustic models. In *Proc. of International Conference on Machine Learning (ICML)*, 2013.

[21] C. Giraud. DFA on AES. In *Advanced Encryption Standard - AES*, 2004.

[22] Y. Xiao, *et al*. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *Proc. of USENIX Security Symposium*, 2016.

[23] V. van der Veen, *et al*. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proc. of Conference on Computer and Communications Security (CCS)*, 2016.

[24] T. Ito and I. Takanami. On fault injection approaches for fault tolerance of feedforward neural networks. In *Proc. of Asian Test Symposium (ATS)*, 1997.

[25] I. Takanami, *et al*. A fault-value injection approach for multiple-weight-fault tolerance of mnns. In *Proc. of International Joint Conference on Neural Networks*, 2000.

[26] D. Phatak and I. Koren. Fault tolerance of feedforward neural nets for classification tasks. In *Proc. of International Joint Conference on Neural Networks*, 1992.

[27] D. S. Phatak and I. Koren. Complete and partial fault tolerance of feedforward neural nets. In *IEEE Transactions on Neural Networks*, 1995.

[28] N. Kamiura, *et al*. An improvement in weight-fault tolerance of feedforward neural networks. In *Proc. of Asian Test Symposium (ATS)*, 2001.

[29] N. Kamiura, *et al*. Learning based on fault injection and weight restriction for fault-tolerant hopfield neural networks. In *Proc. of International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, 2004.