Exploring Complexity Reduction in Deep Learning

Sourya Dey

Dissertation Defense

April 22nd, 2020

USC Viterbi
School of Engineering
Ming Hsieh Department of Electrical and Computer Engineering

Key contributions

Pre-defined Sparsity

- Reduce complexity of NNs
- Guidelines for designing sparse NNs
- Hardware architecture for on-device training and inference

Automated Machine Learning: Deep-n-Cheap

- Target performance and training complexity
- Benchmark and custom datasets, CNNs and MLPs
- Insights into search process

Dataset Engineering

- Family of synthetic datasets
- Dataset difficulty metrics

Outline


Introduction and Background


Pre-Defined Sparsity
https://github.com/souryadey/predefinedsparse-nnets


Automated Machine Learning : Deep-n-Cheap
https://github.com/souryadey/deep-n-cheap


Dataset Engineering
https://github.com/souryadey/morse-dataset

Introduction and Background

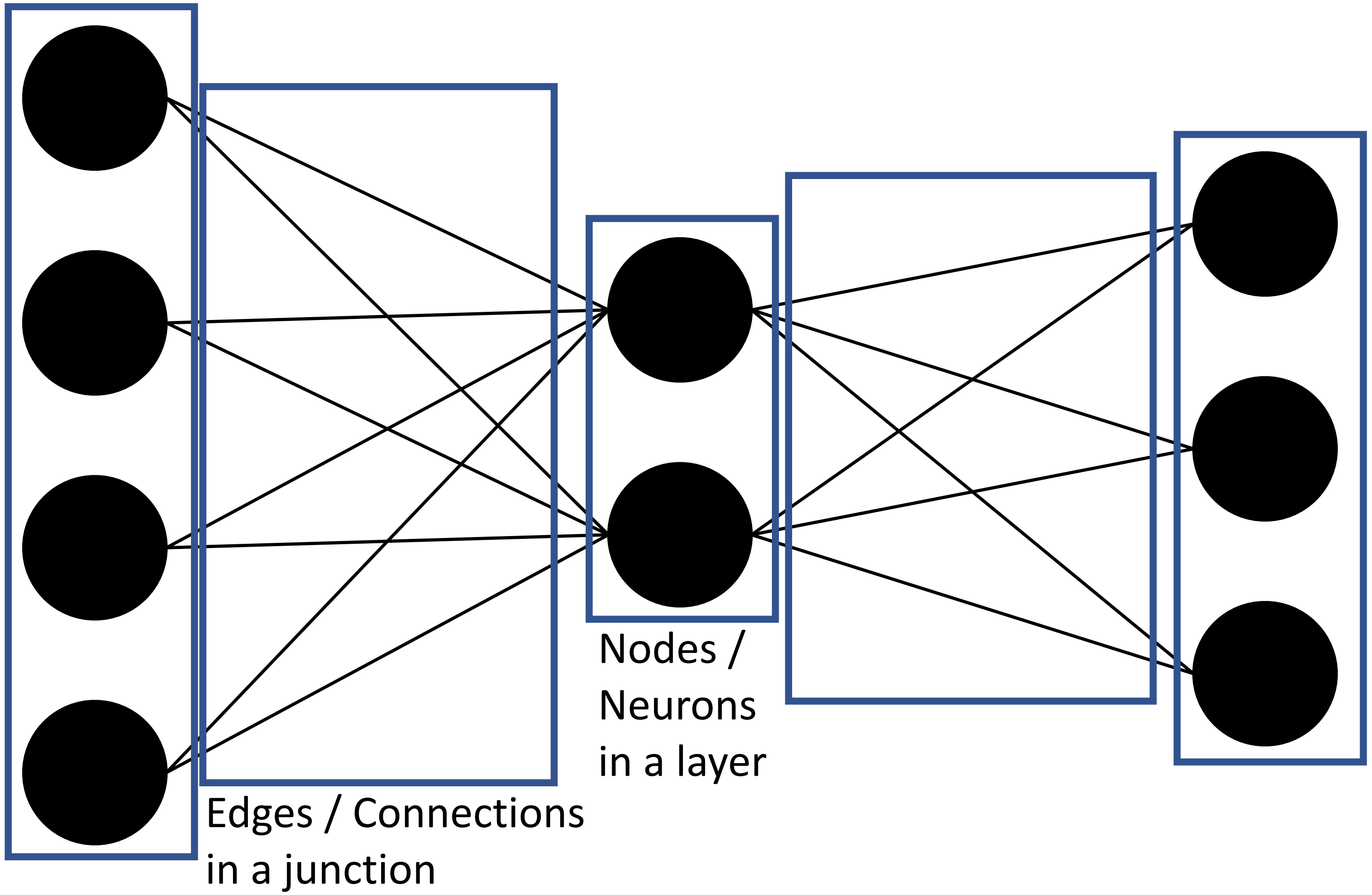
Deep Learning

Machine Learning
Neural Networks

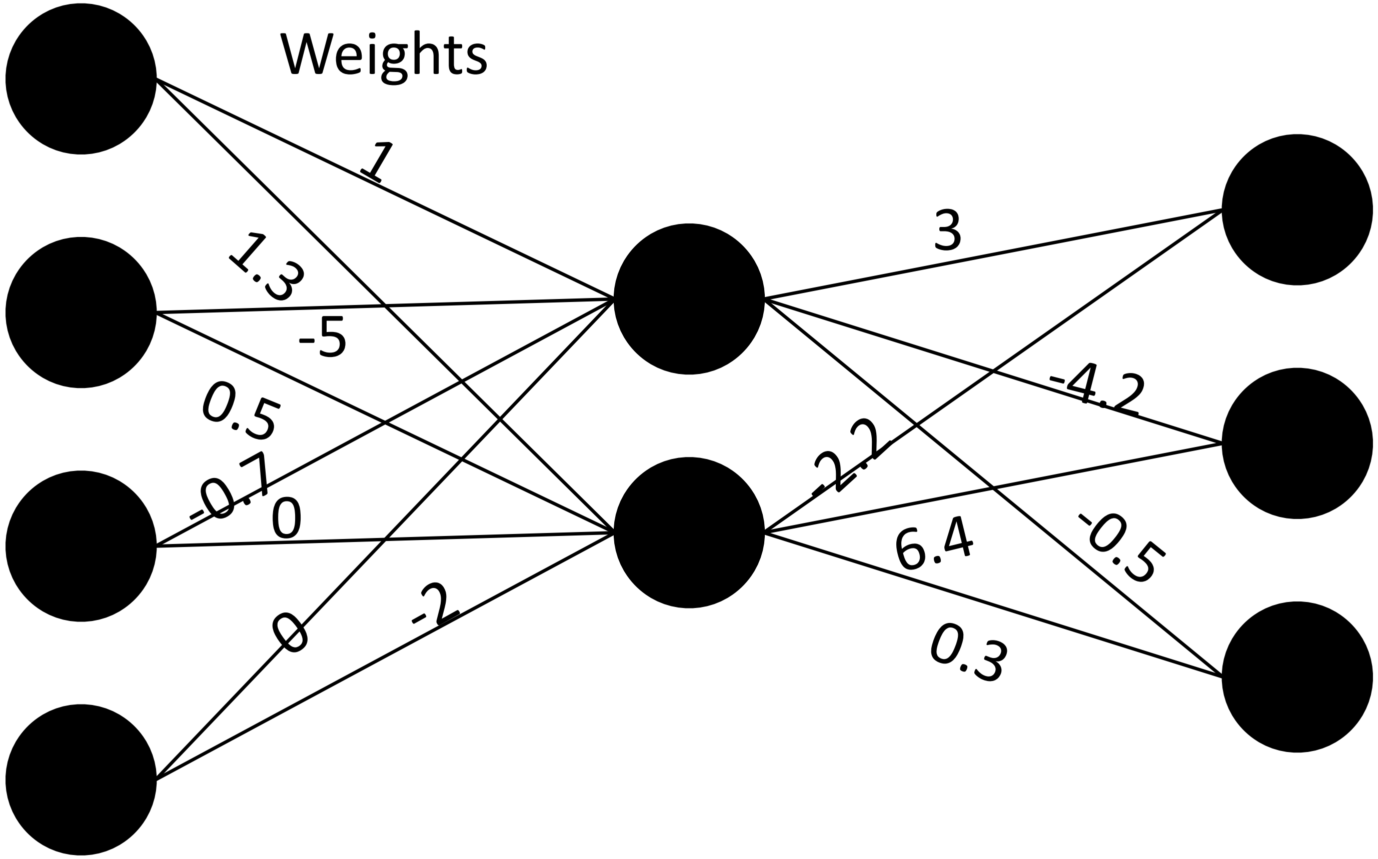
Artificial Intelligence

Smart Systems

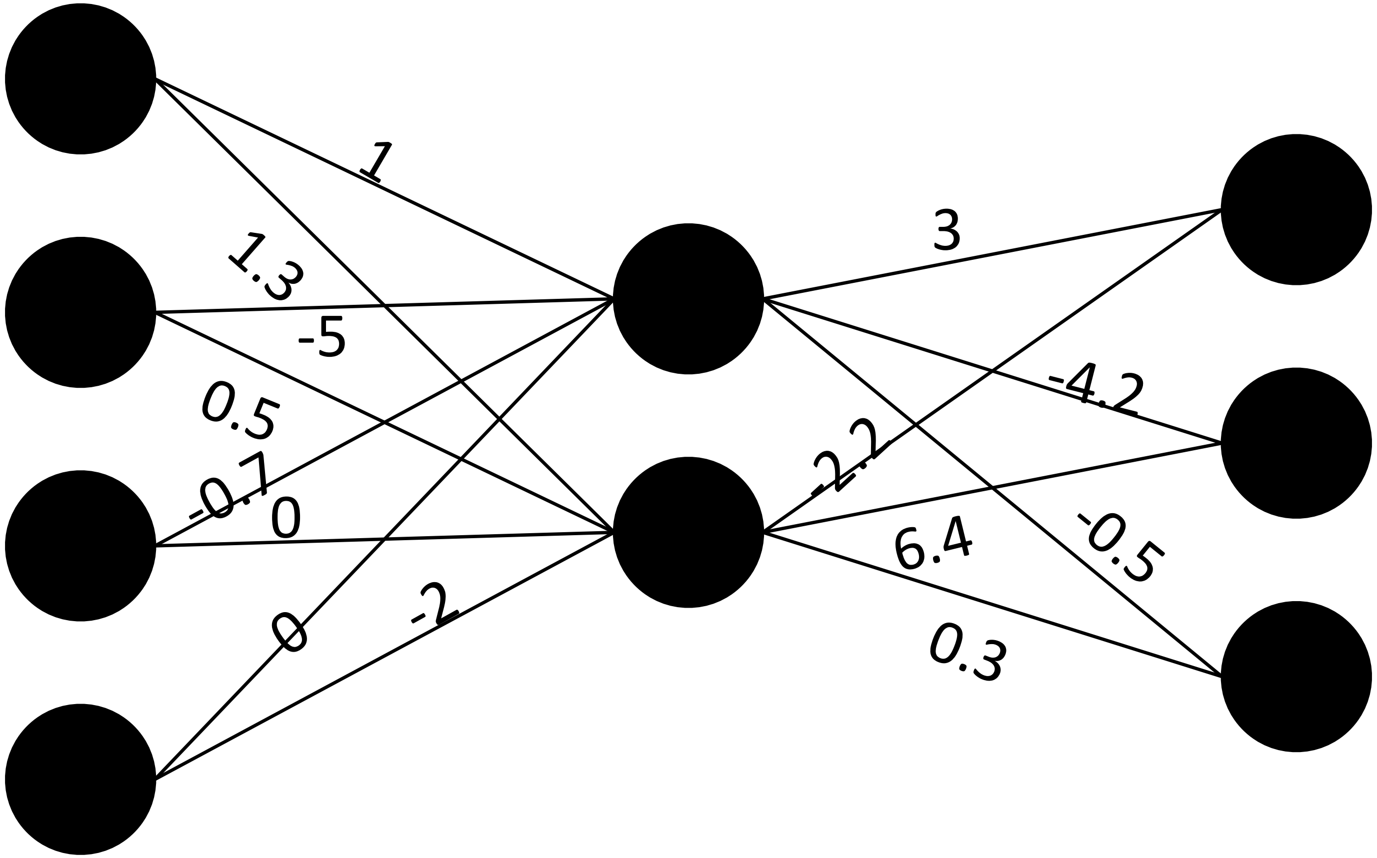
A Quick Primer on Neural Networks (NN101)



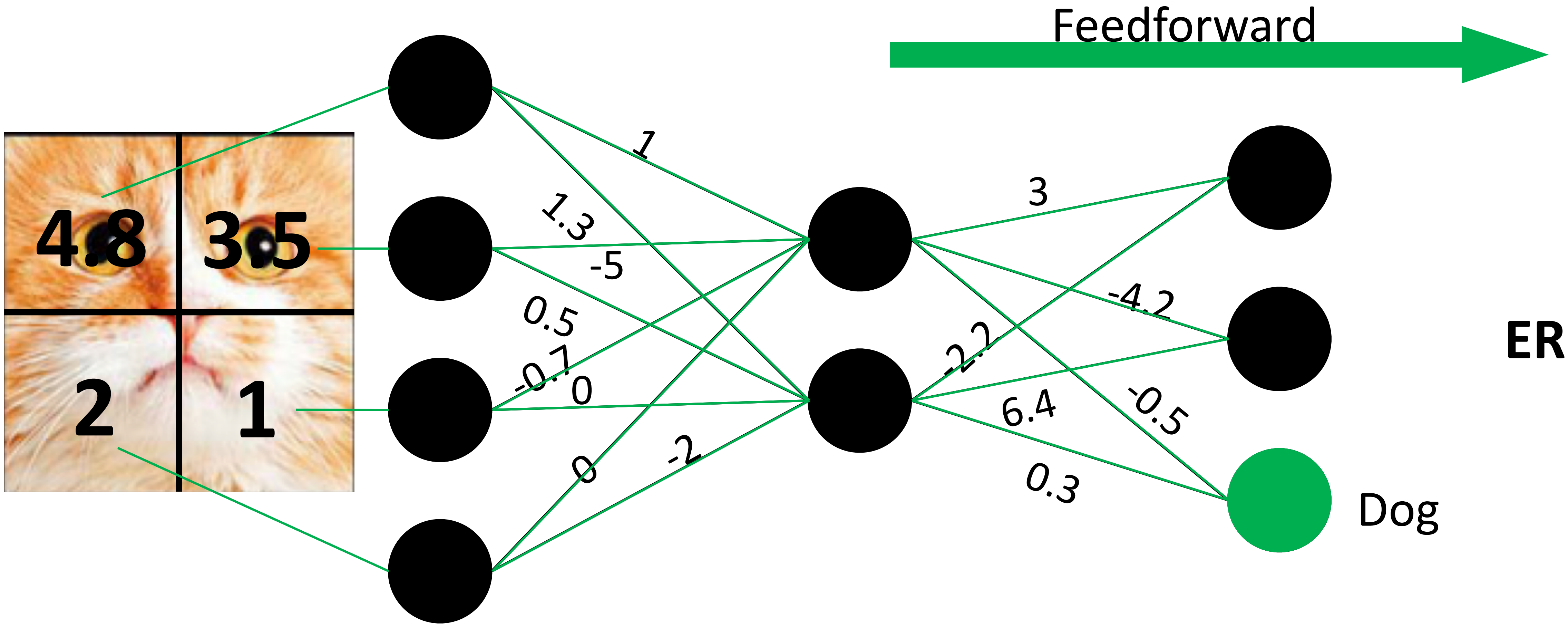
A Quick Primer on Neural Networks (NN101)



A Quick Primer on Neural Networks (NN101)



A Quick Primer on Neural Networks (NN101)

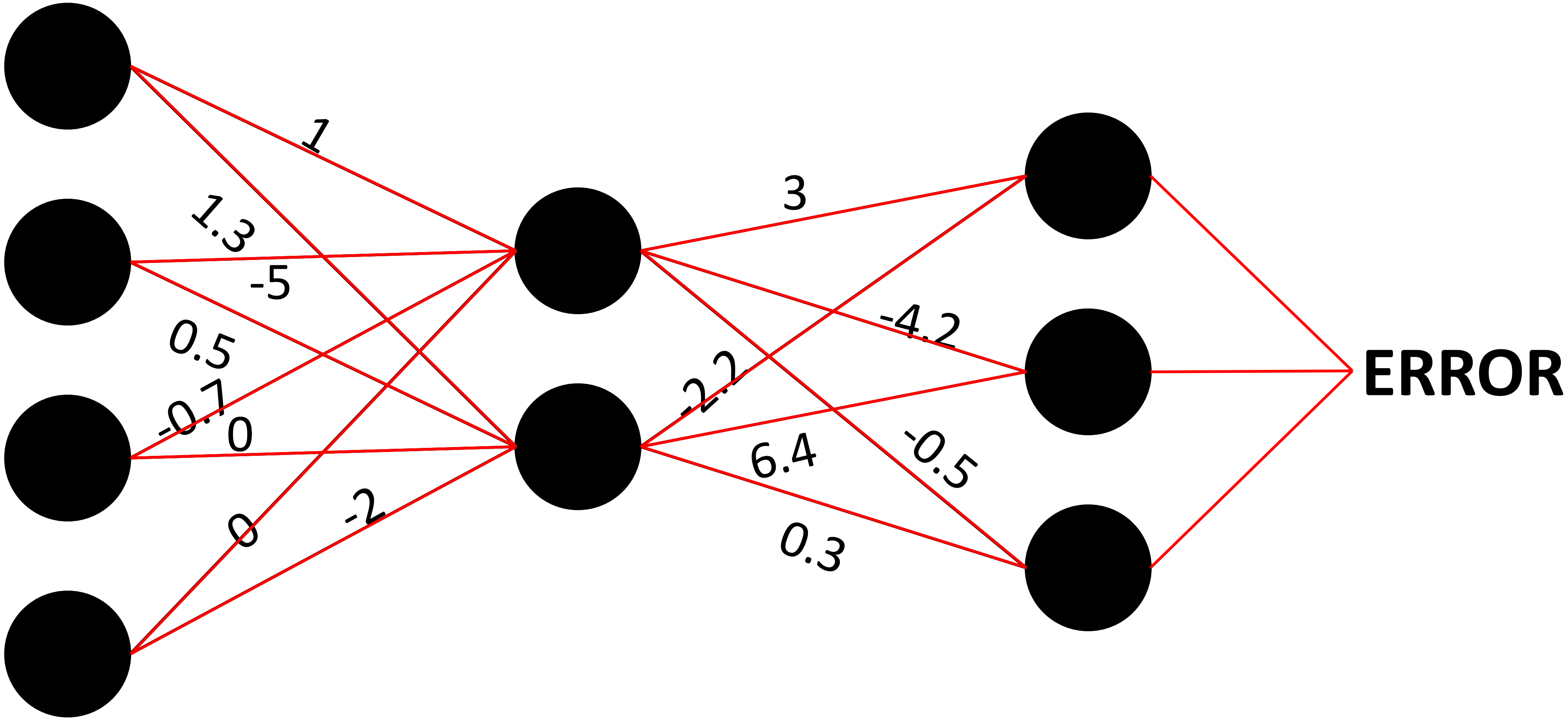


ERROR

TRAINING

Learn network parameters — weights

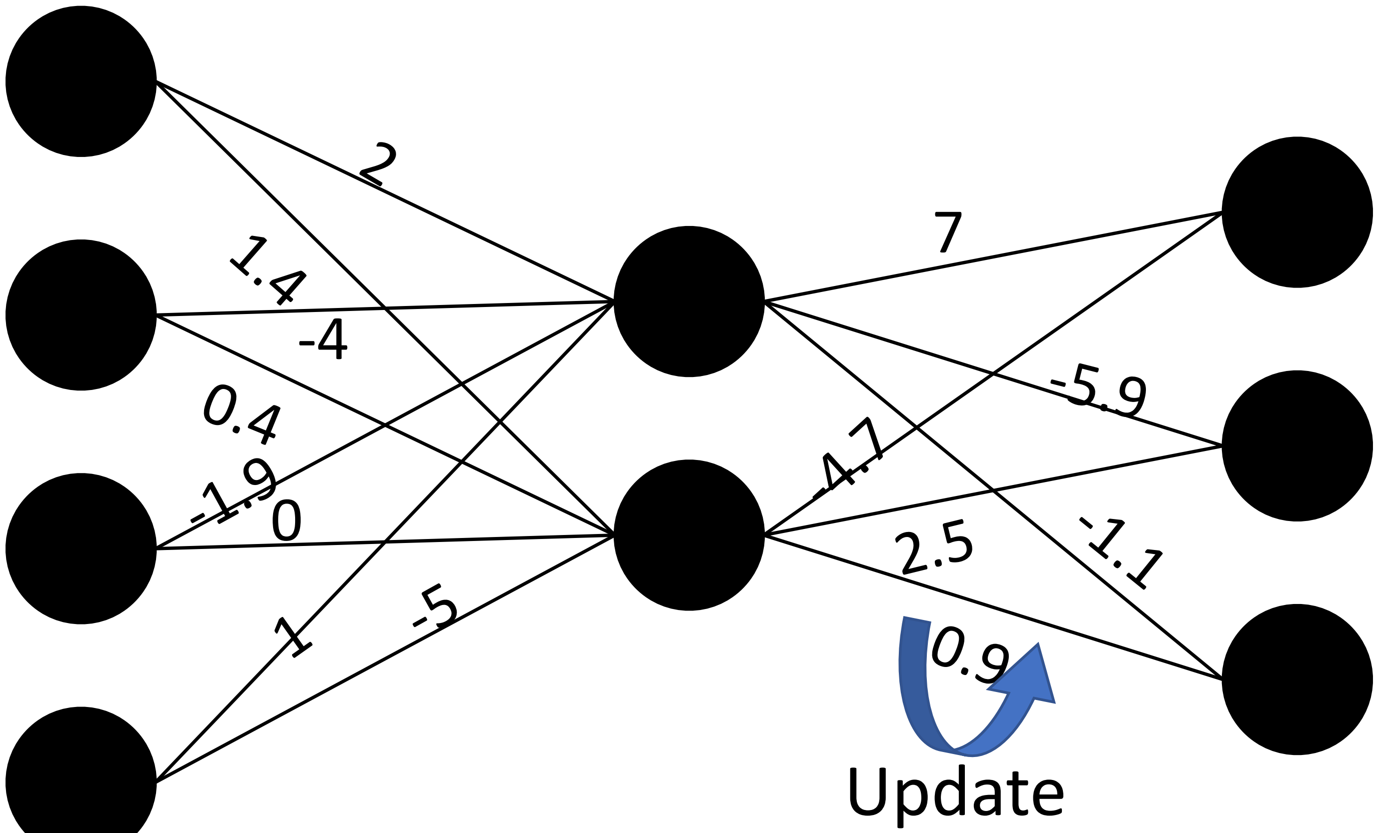
A Quick Primer on Neural Networks (NN101)



TRAINING

Learn network parameters — weights

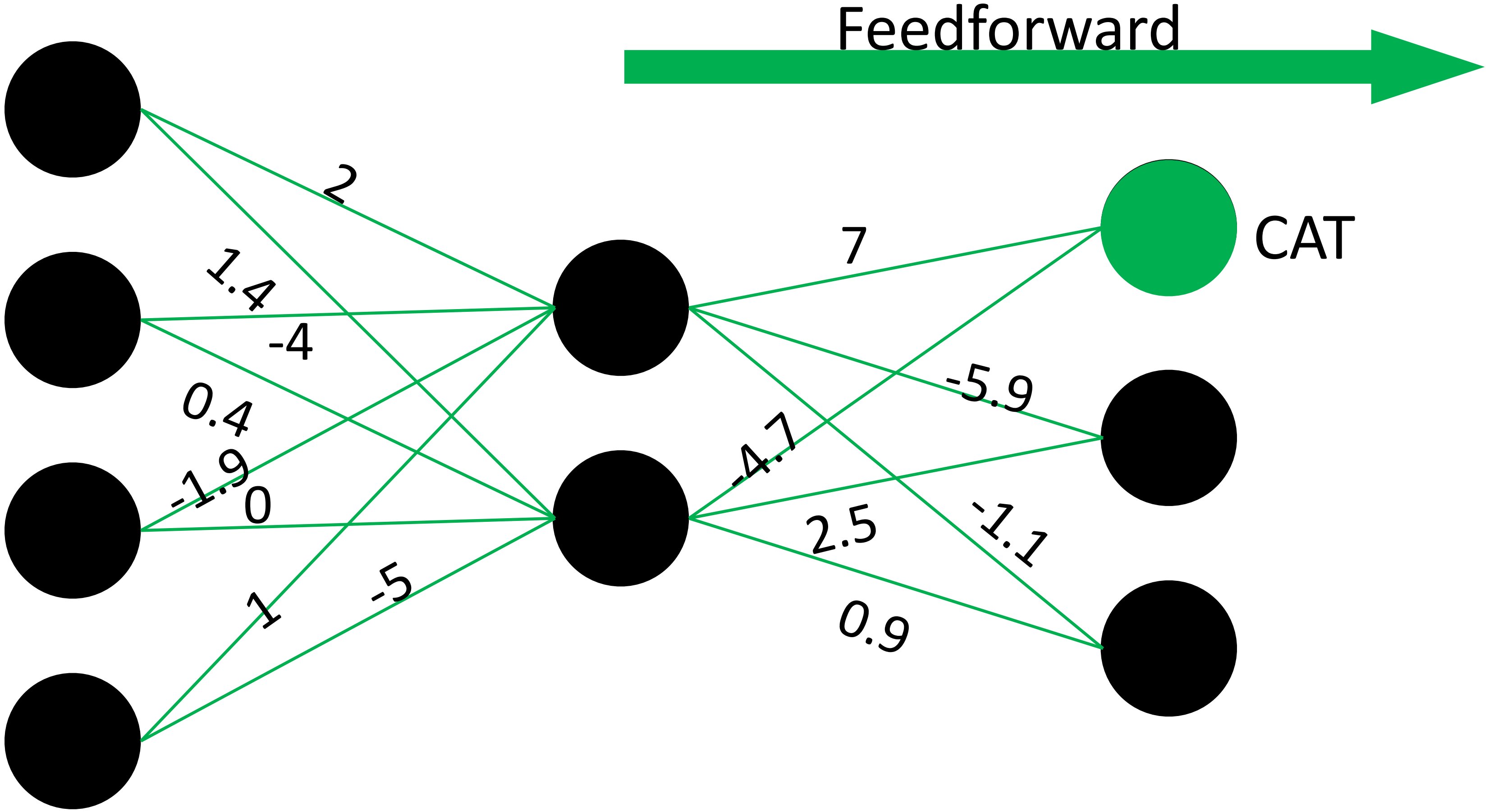
A Quick Primer on Neural Networks (NN101)



TRAINING

Learn network parameters — weights

NNs can be used for classification

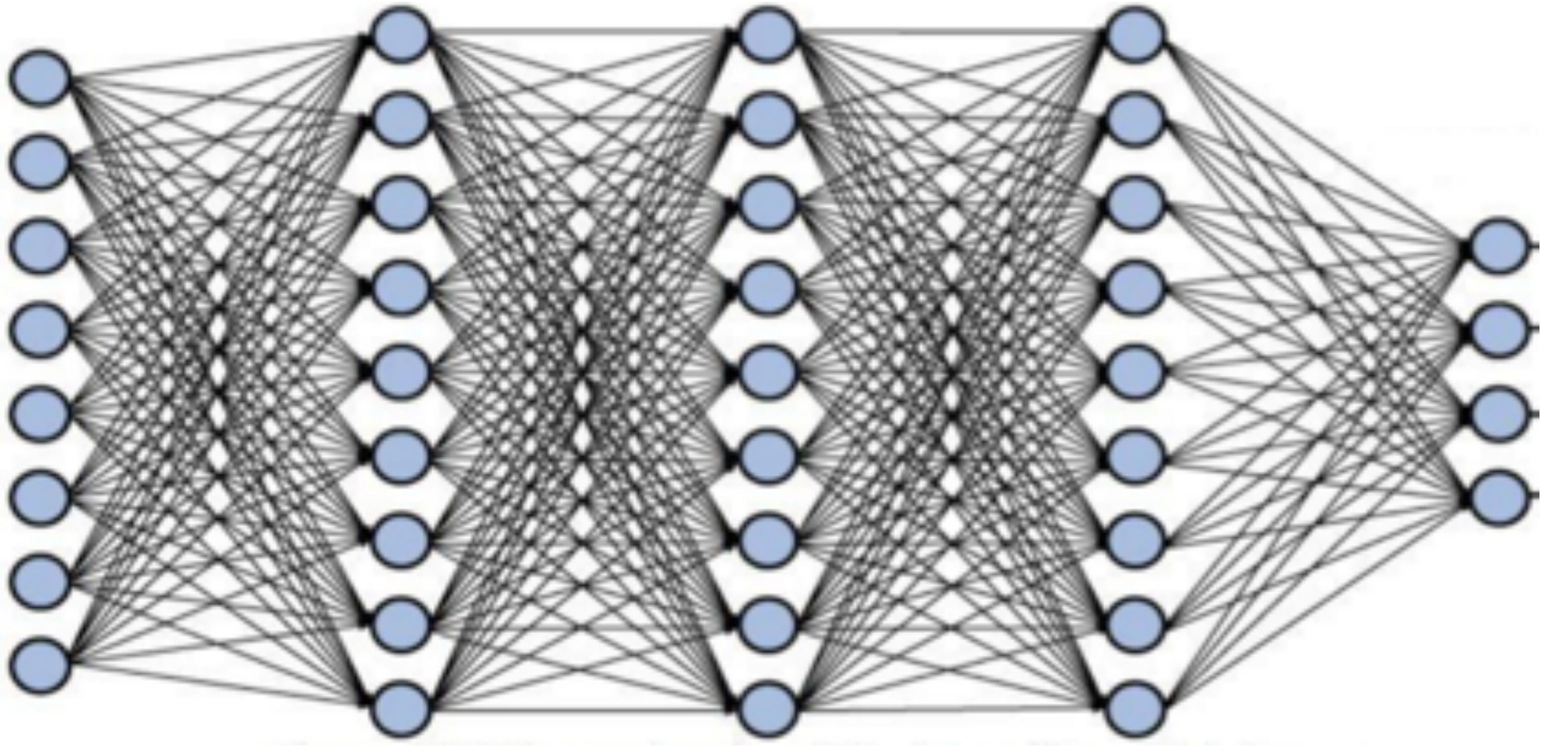


TESTING / INFERENCE

Use learned network parameters

Measure accuracy performance — % of correctly classified test samples

Types of NNs – Multilayer Perceptron (MLP)



Fully connected (FC) – every node connects to every adjacent node

Types of NNs – Convolutional Neural Network (CNN)

Filter / Kernel

1	0	1	3	-1	-3
0	1	0	-4	-2	4
1	0	1	4	-2	0
0	0	0	4	-3	2
0	2	1	1	1	0
0	1	3	-1	0	-3

Channels

4	3	-4	2
-2	2	0	4
1	4	5	2
5	5	1	3

4	4
5	5

2	1
-1	0

	1
-1	

Convolution

technically it's correlation…
but since when do engineers
bother about math?

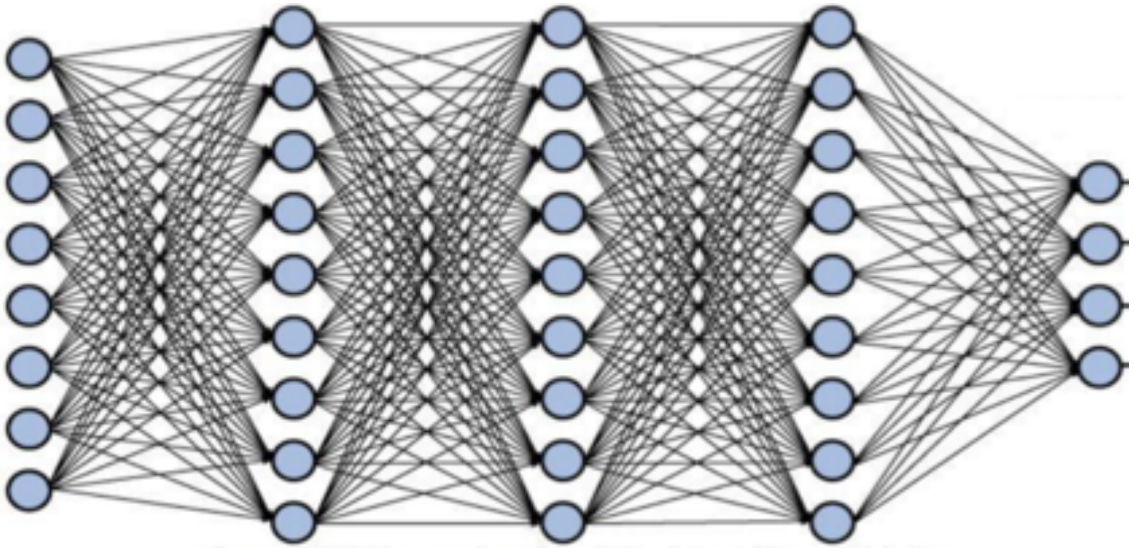
Pooling (Downsampling)

Batch Normalization

Dropout

The Complexity Conundrum…

Modern neural networks suffer from parameter explosion



Training can take weeks on CPU

Cloud GPU resources are expensive

amazon web services

Google Cloud Platform

Microsoft Azure

He 2016

… and the Design Conundrum

- Deep neural networks have a lot of **hyperparameters**
  - How many layers?
  - How many neurons?

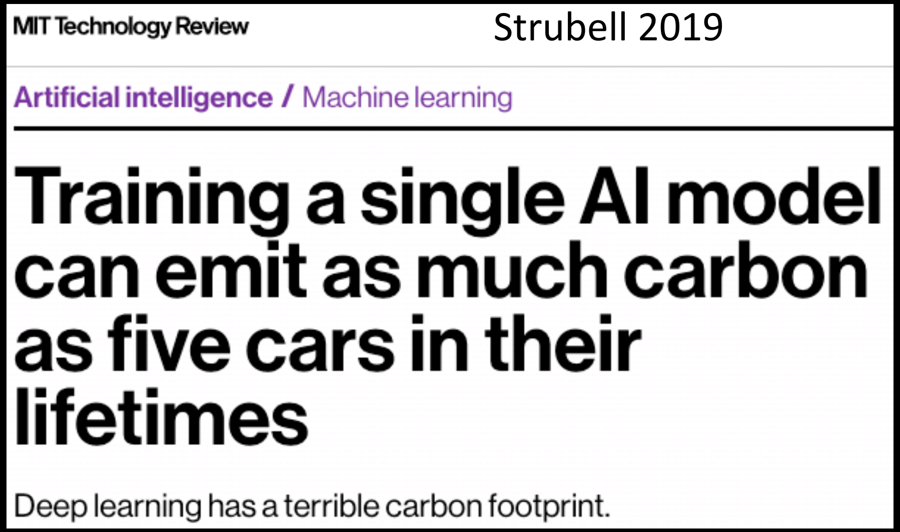
    Architecture Hyperparameters
  - Learning rate
  - Batch size

    Training Hyperparameters
  - and more…

- Our understanding of NNs is at best vague, at worst, zero!

The big question my research aims to answer

Can we reduce the storage and computational (which translate to temporal, financial and environmental) burden of deploying NNs, particularly the training phase, while minimizing performance degradation?



MIT Technology Review

Strubell 2019

Artificial intelligence / Machine learning

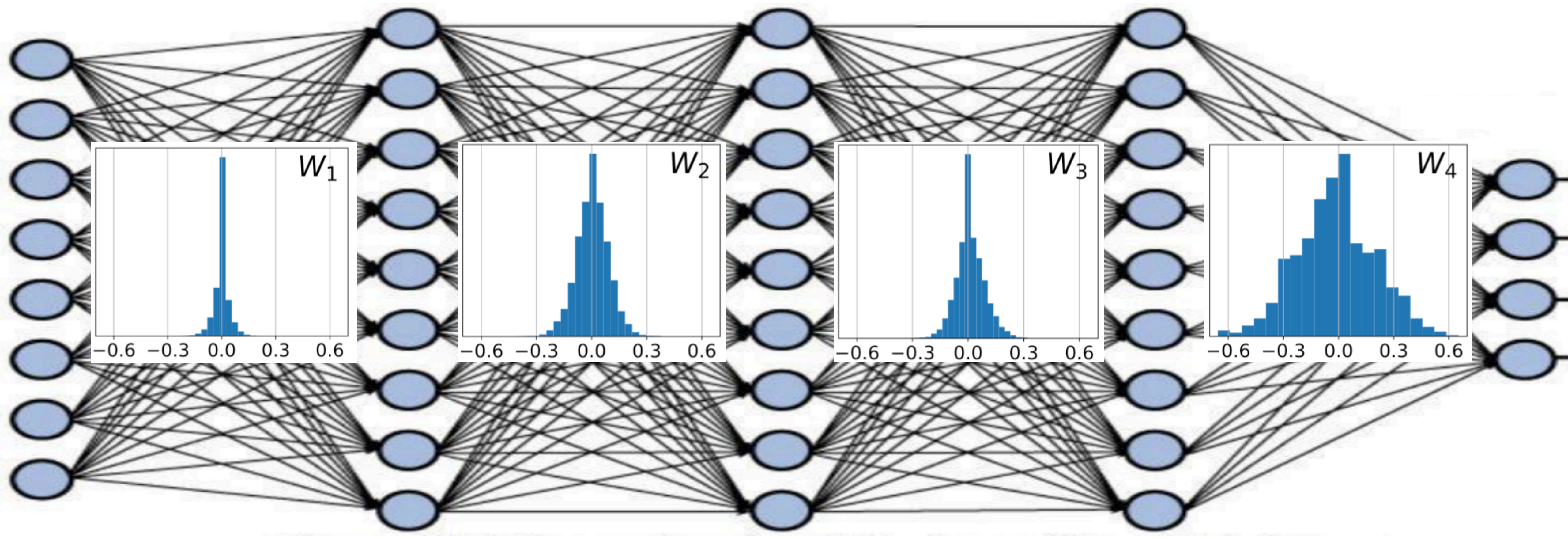
Training a single AI model can emit as much carbon as five cars in their lifetimes

Deep learning has a terrible carbon footprint.

Pre-Defined Sparsity

https://github.com/souryadey/predefinedsparse-nnets

Motivation behind pre-defined sparsity



In a FC MLP network, most weights are small in magnitude after training

Pre-defined Sparsity

Pre-define a sparse connection pattern **prior to training**

Use this sparse network for both training and inference

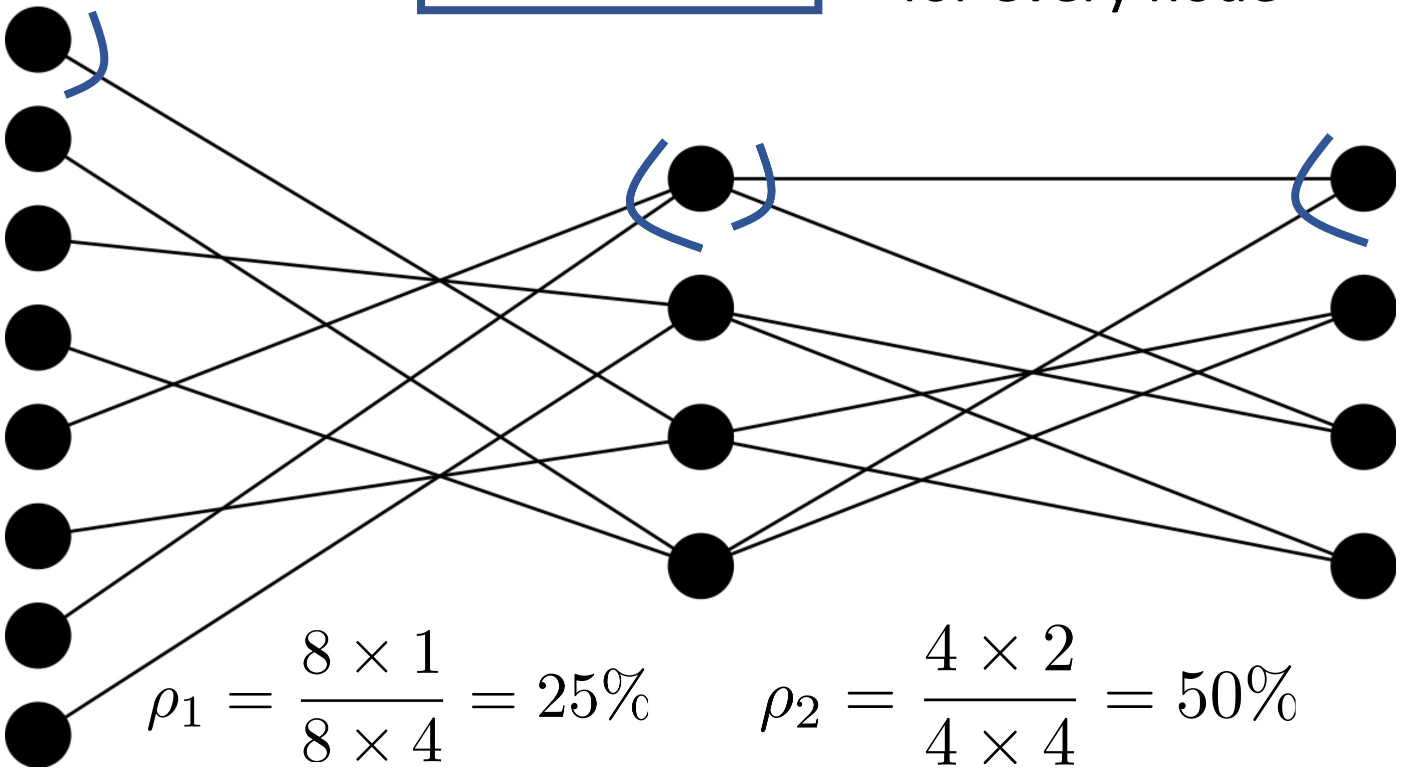
Reduced training *and* inference complexity

$$\boldsymbol{N}_{\text{net}} = (8, 4, 4)$$

$$\boldsymbol{d}_{\text{net}}^{\text{out}} = (1, 2)$$

$$\boldsymbol{d}_{\text{net}}^{\text{in}} = (2, 2)$$

Structured Constraints:
Fixed in-, out-degrees for every node

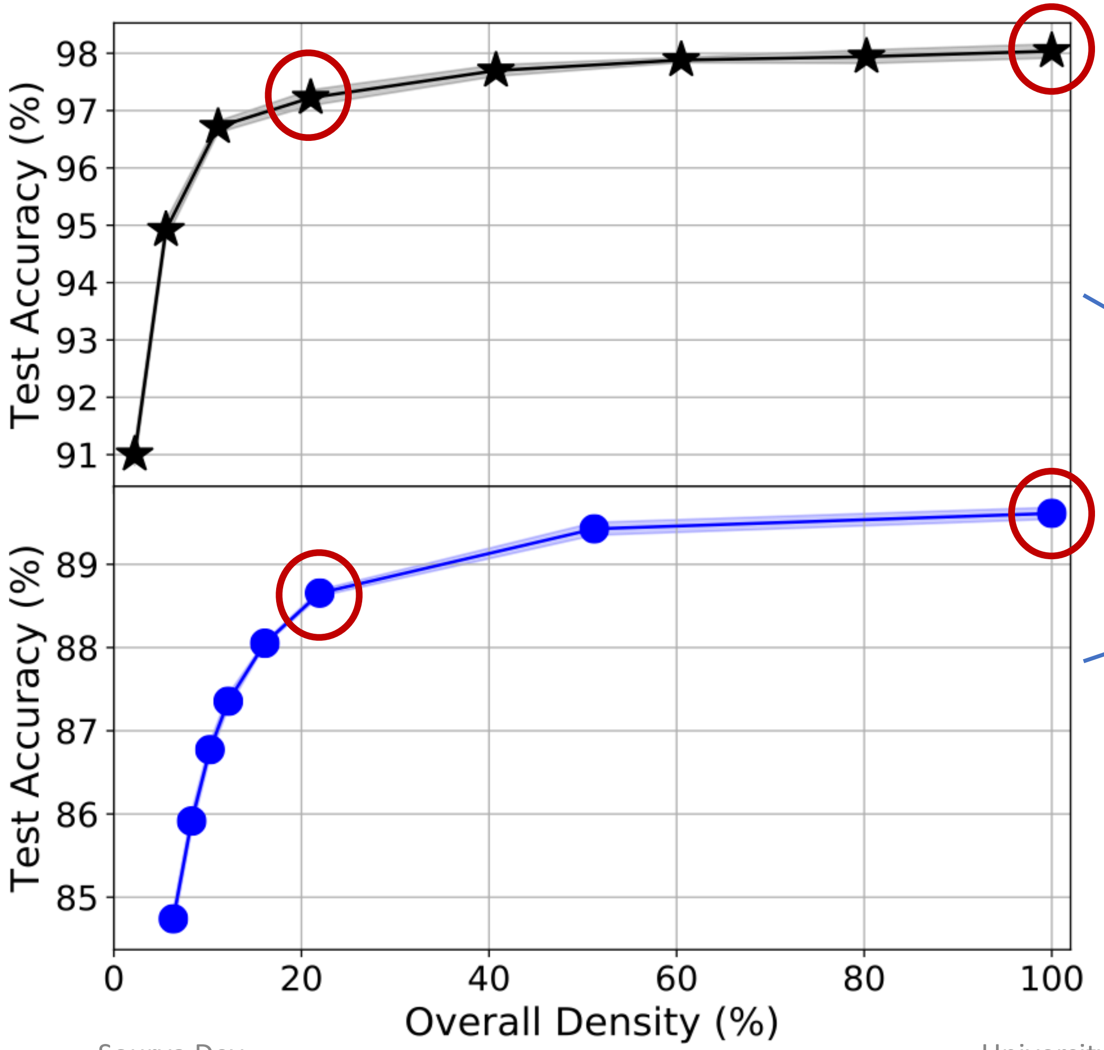


$$\rho_1 = \frac{8 \times 1}{8 \times 4} = 25\% \qquad \rho_2 = \frac{4 \times 2}{4 \times 4} = 50\%$$

$$\rho_{\text{net}} = \frac{8 + 8}{32 + 16} = 33\%$$

Overall Density compared to FC

Pre-defined sparsity performance on MLPs



Starting with only 20% of parameters reduces test accuracy by just 1%

MNIST handwritten digits

Reuters news articles

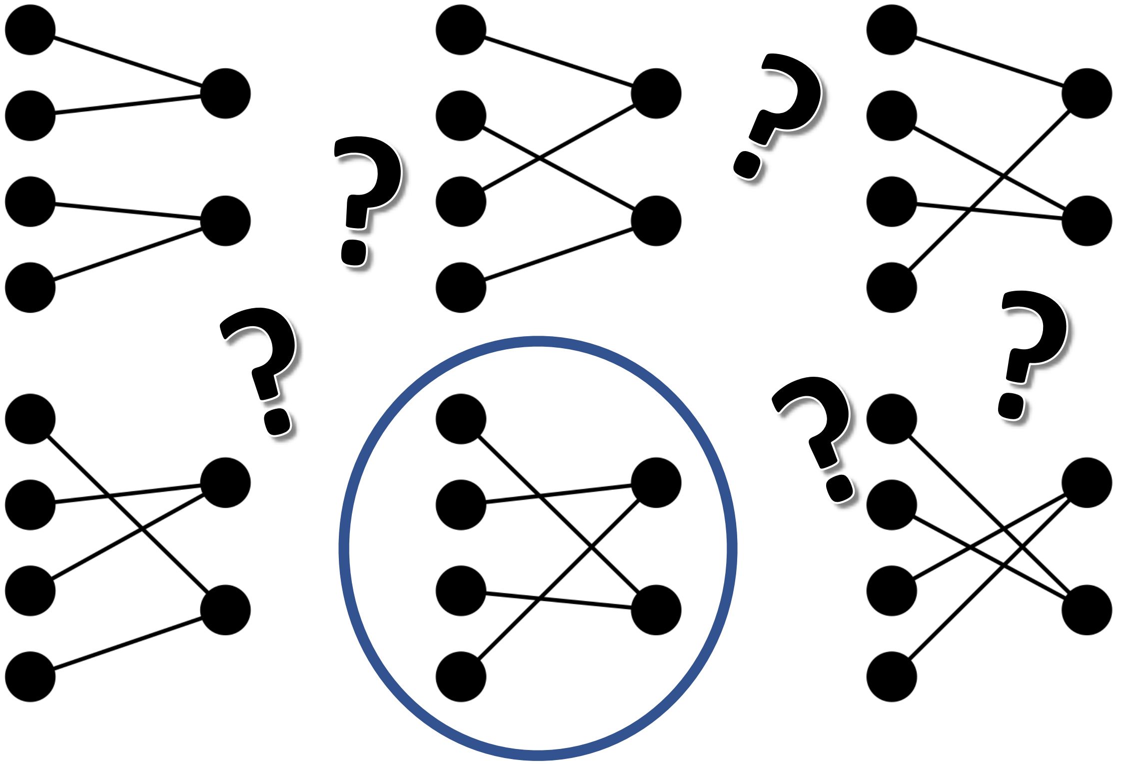
TIMIT phonemes

CIFAR images

Morse symbols

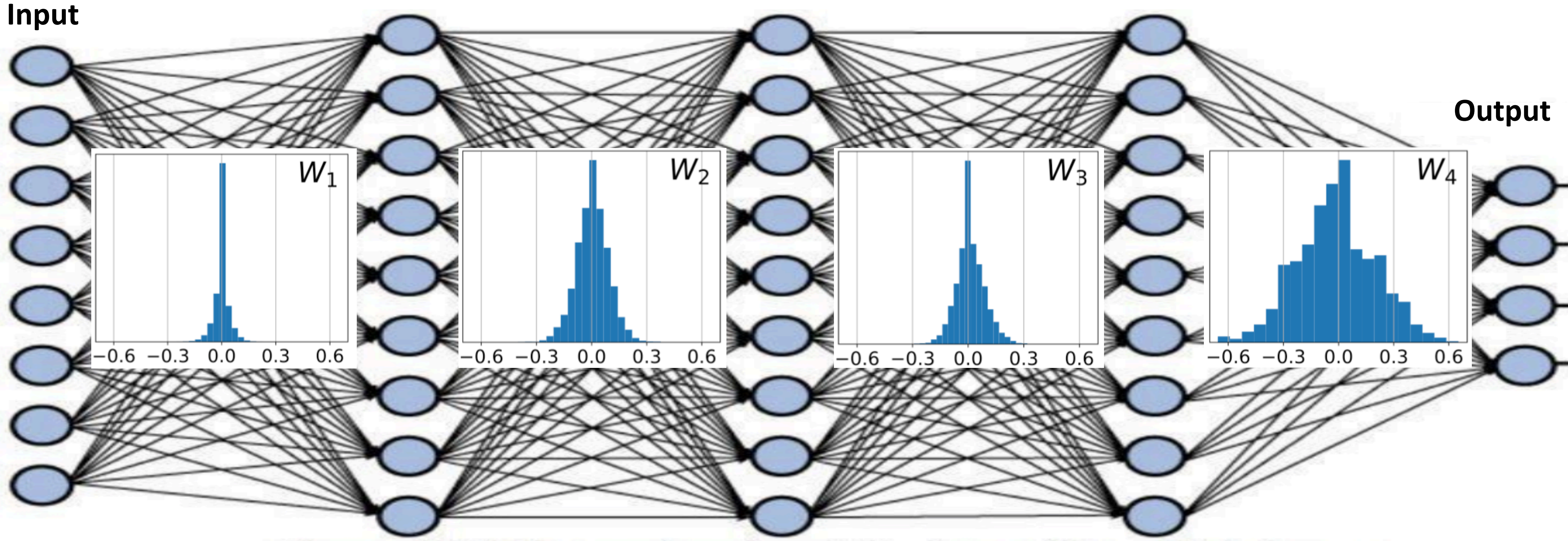
Designing pre-defined sparse networks

*A pre-defined sparse connection pattern is a **hyperparameter** to be set prior to training*



Find trends and guidelines to optimize pre-defined sparse patterns

1. Individual junction densities



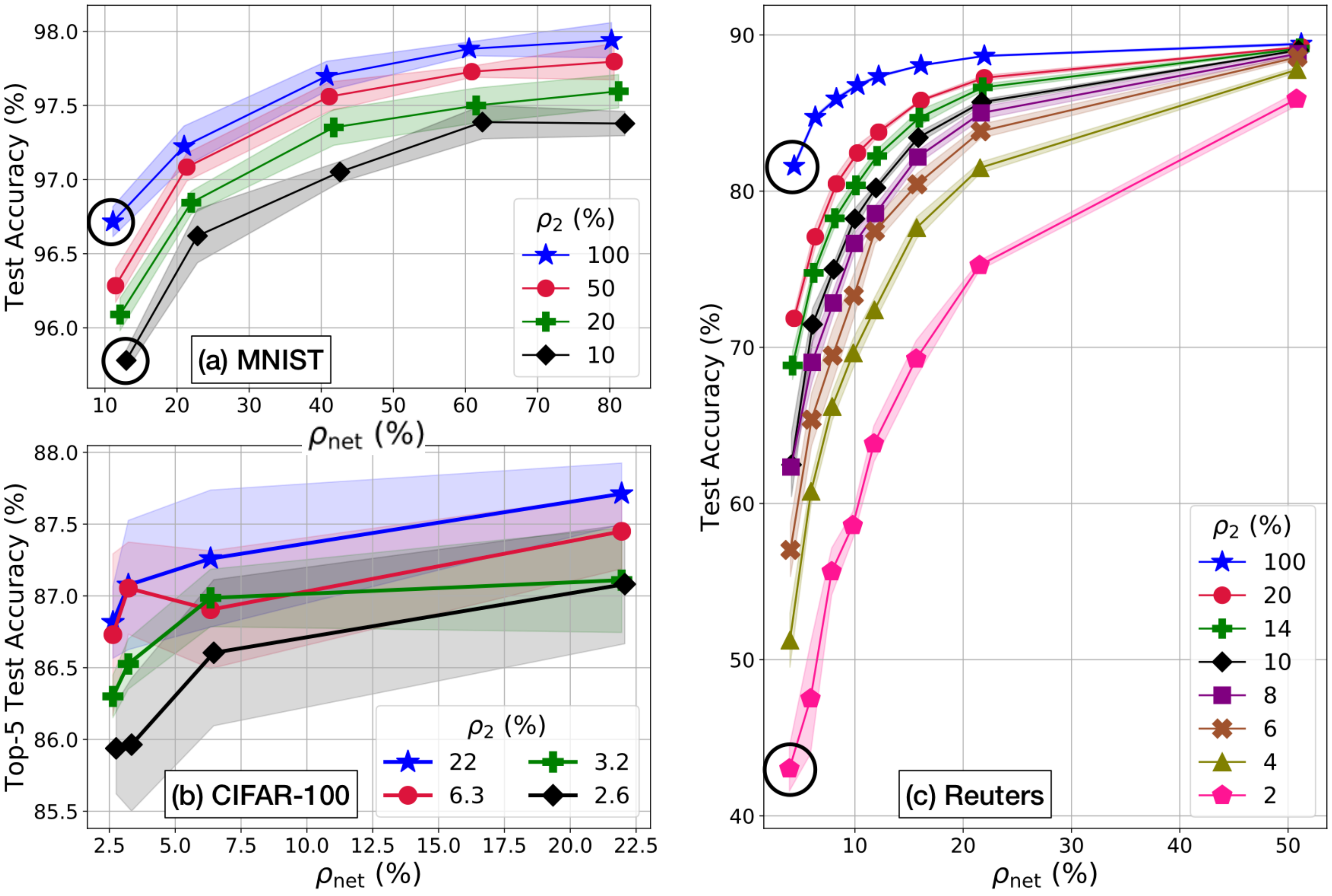
Latter junctions (closer to the output) learn higher-order, more complicated representations => They need to be denser

Results

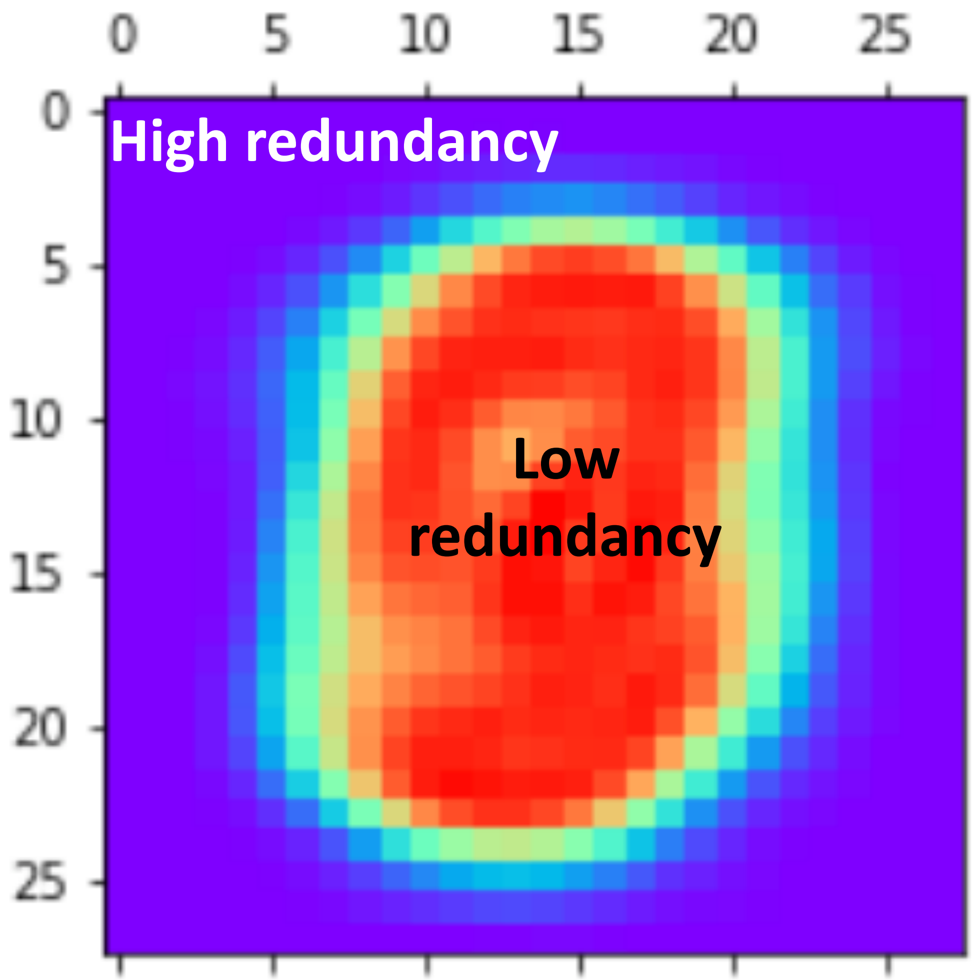
Each curve keeps ρ_2 fixed and varies ρ_{net} by varying ρ_1

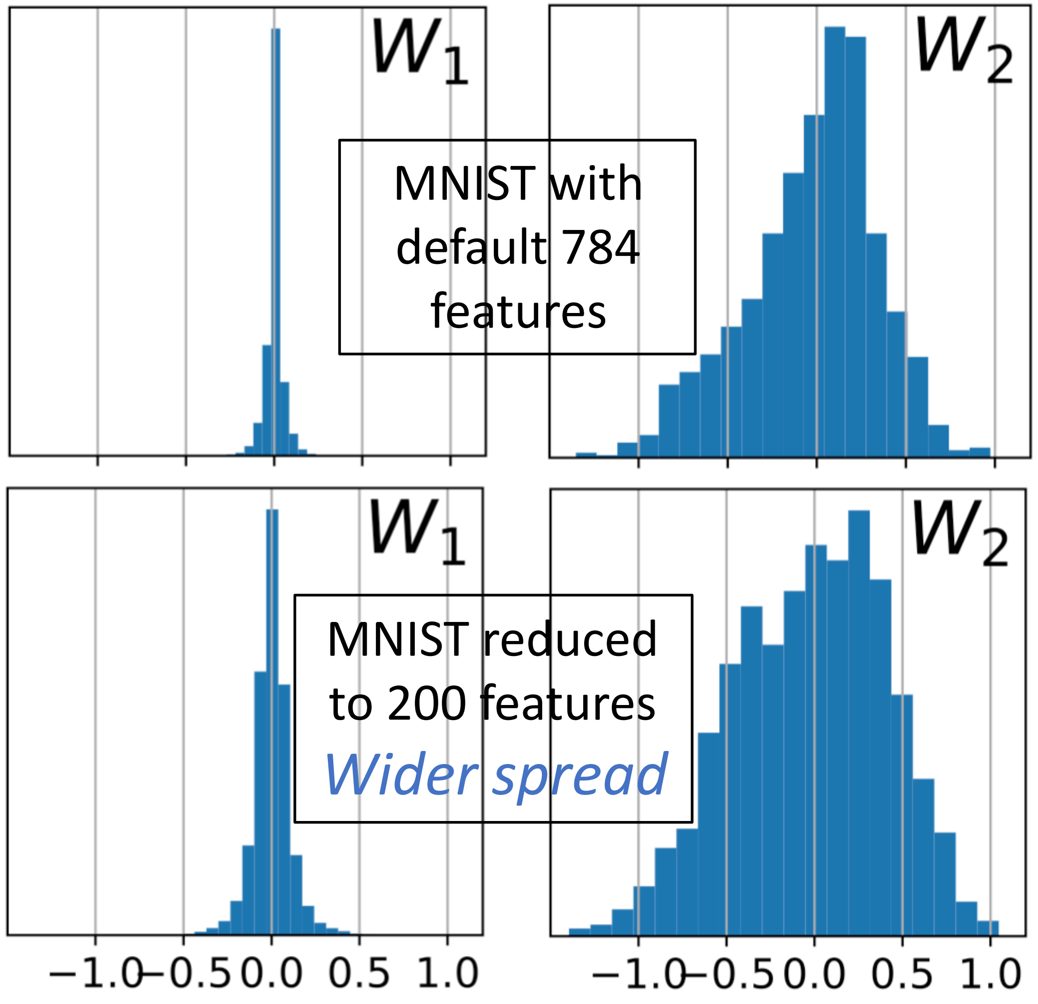
For the same ρ_{net}, $\rho_2 > \rho_1$ improves performance

Mostly similar trends observed for deeper networks



2. Dataset redundancy



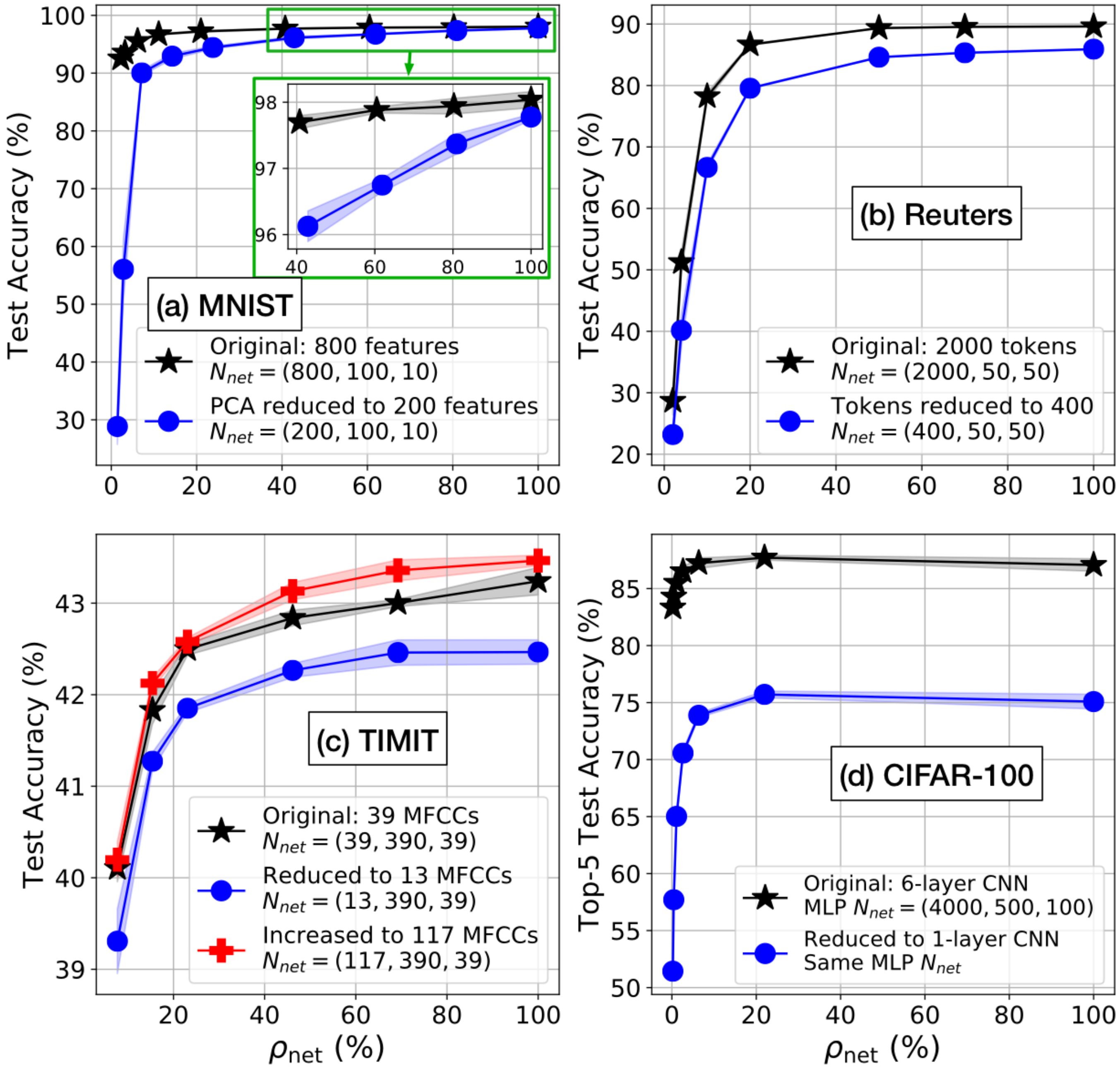


Less redundancy => Less sparsification possible

Results

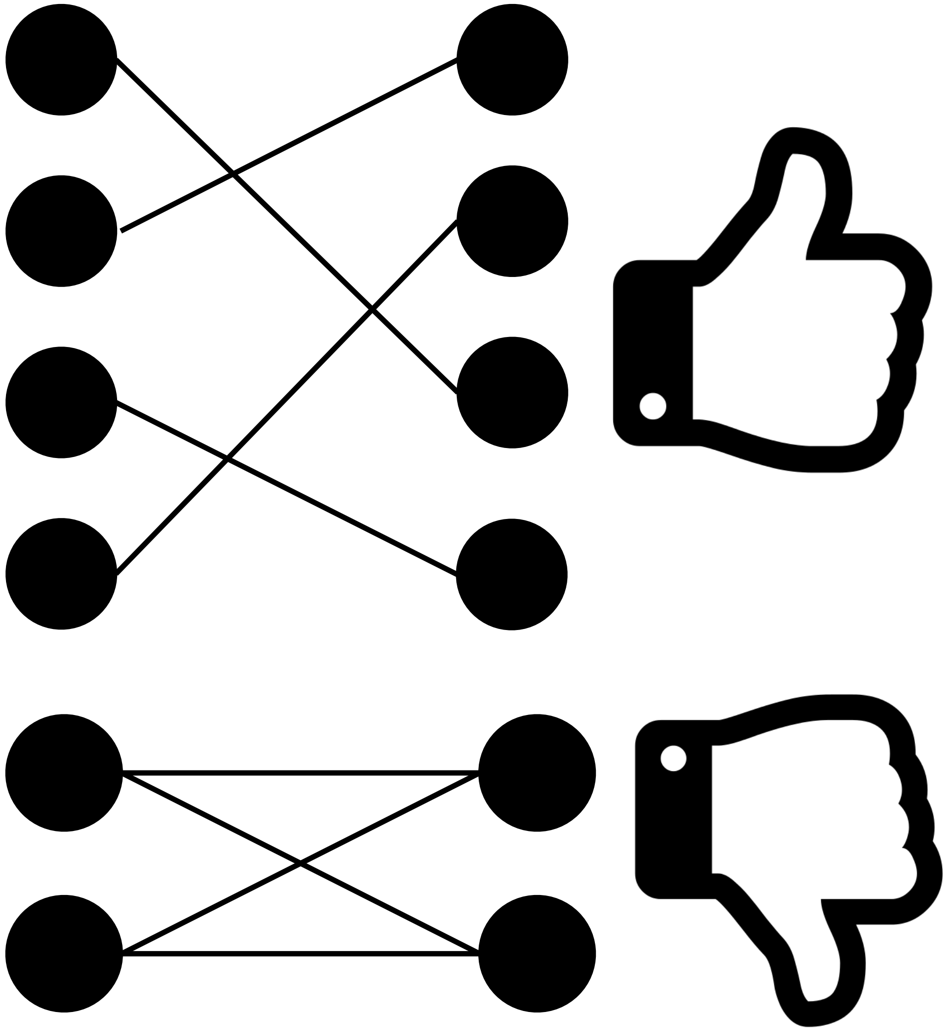
Reducing redundancy leads to increased performance degradation on sparsification

Pre-defined sparse design is problem-dependent



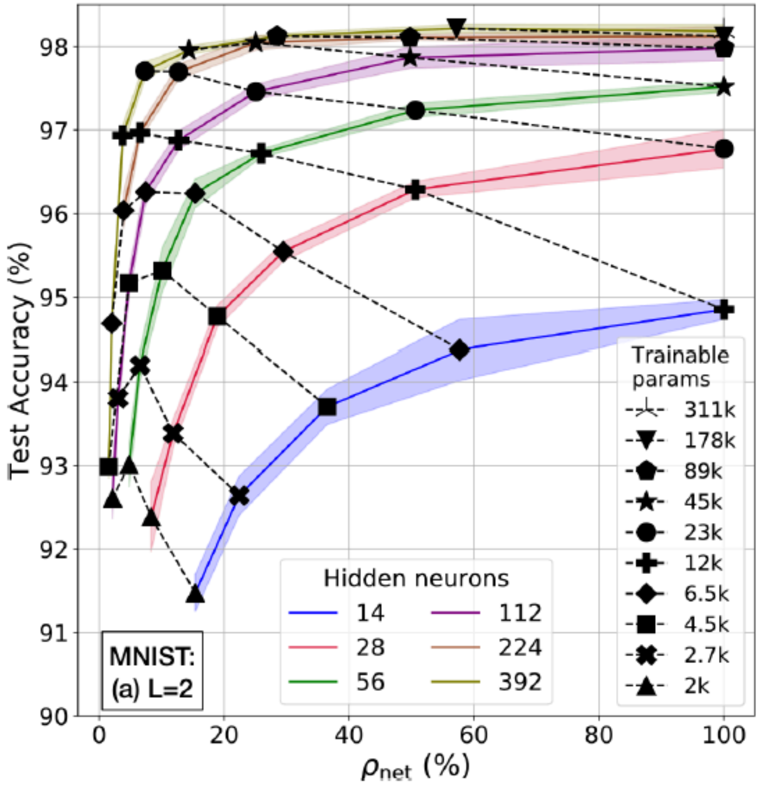
3. ‘Large sparse’ vs ‘small dense’ networks

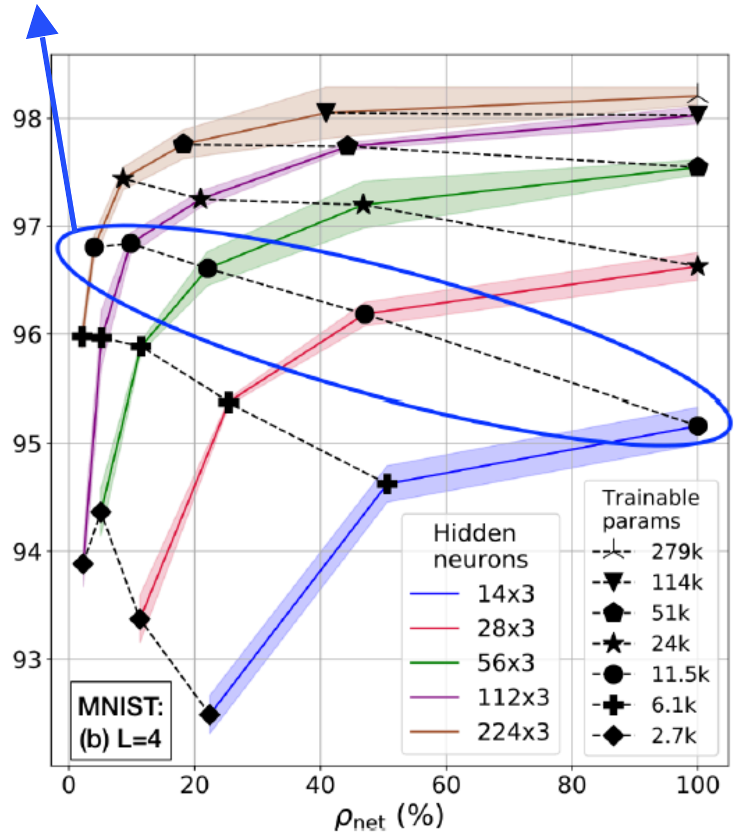
A sparser network with more hidden nodes will outperform a denser network with less hidden nodes, when both have same number of weights

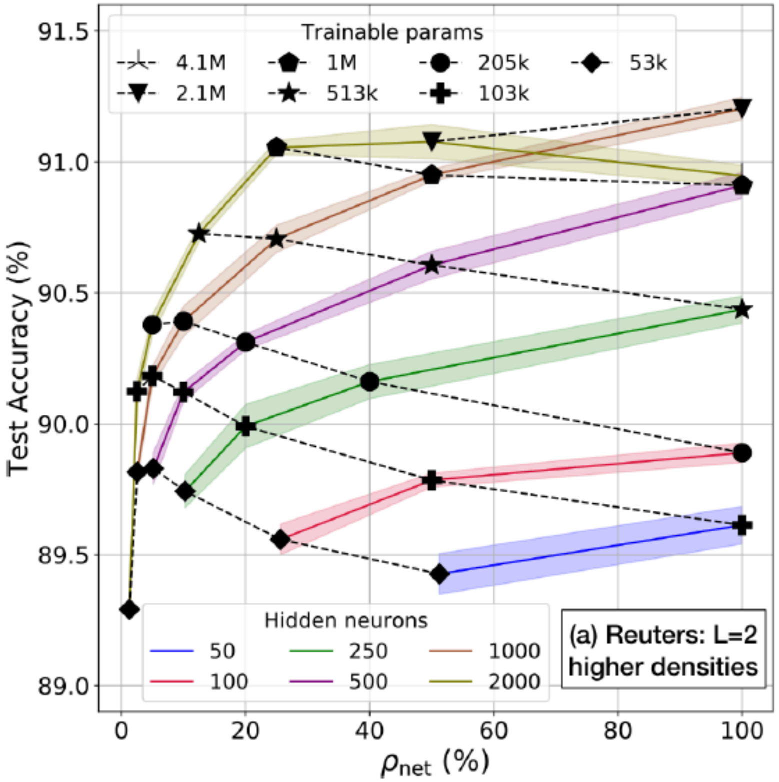


Results

Networks with same number of parameters go from bad to good as #nodes in hidden layers is increased







4. Regularization – Why does pre-defined sparsity work?

$$C(\boldsymbol{w}) = C_0(\boldsymbol{w}) + \lambda \left\| \boldsymbol{w} \right\|_2^2$$

Regularized cost

Original unregularized cost (like cross-entropy)

Regularization term

Pre-defined sparse networks need smaller λ (as determined by validation)

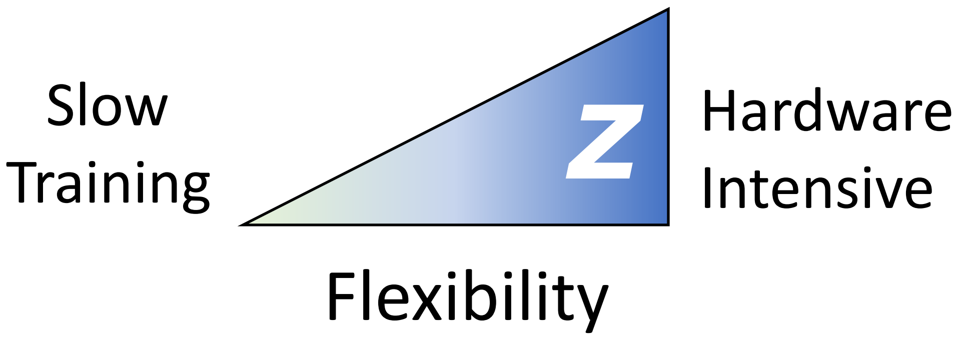
Overall Density	λ
100 %	1.1×10^{-4}
40 %	5.5×10^{-5}
11 %	0

Example for MNIST 2-junction networks

Pre-defined sparsity reduces the overfitting problem stemming from over-parametrization in big networks

Quick Overview of Hardware Architecture

Degree of parallelism (z) = Number of weights processed in parallel in a junction

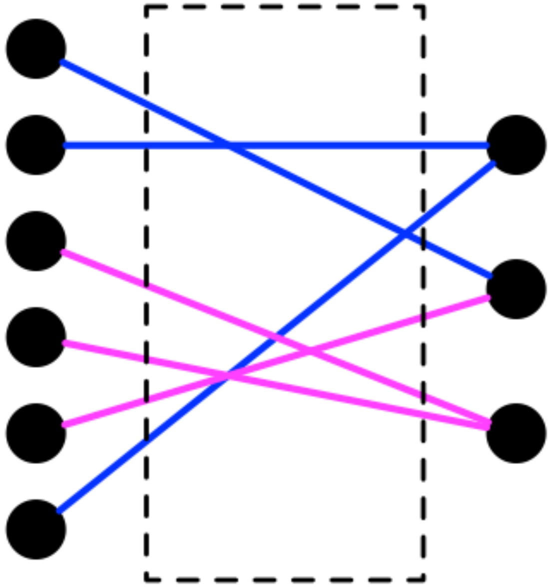


Quick Overview of Hardware Architecture

Degree of parallelism (z) = Number of weights processed in parallel in a junction

Connections designed for clash-free memory accesses to prevent stalling

$z = 3$

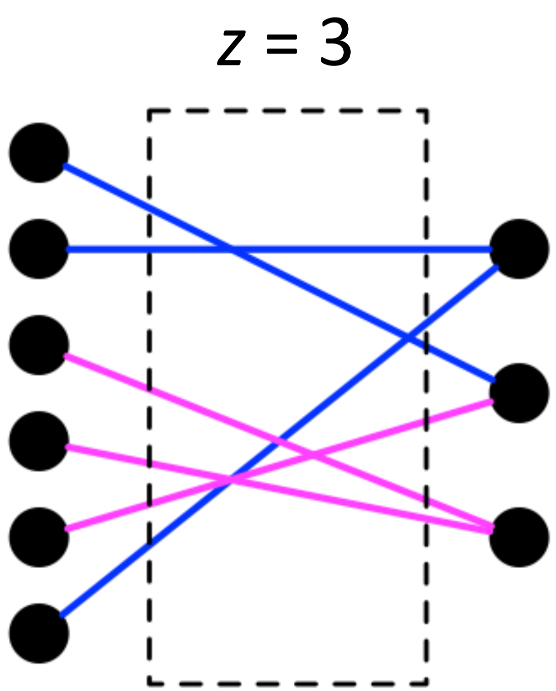


Quick Overview of Hardware Architecture

Degree of parallelism (z) = Number of weights processed in parallel in a junction

Connections designed for clash-free memory accesses to prevent stalling

Clash-free pre-defined sparsity leads to no performance degradation



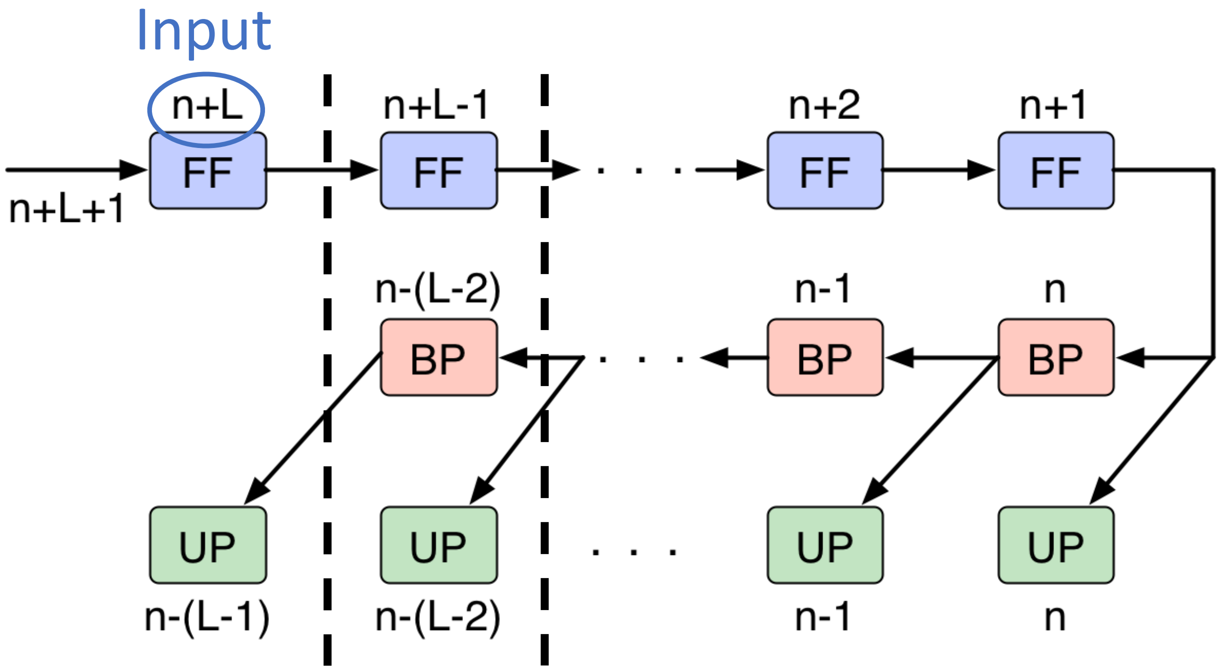
Quick Overview of Hardware Architecture

Degree of parallelism (z) = Number of weights processed in parallel in a junction

Connections designed for clash-free memory accesses to prevent stalling

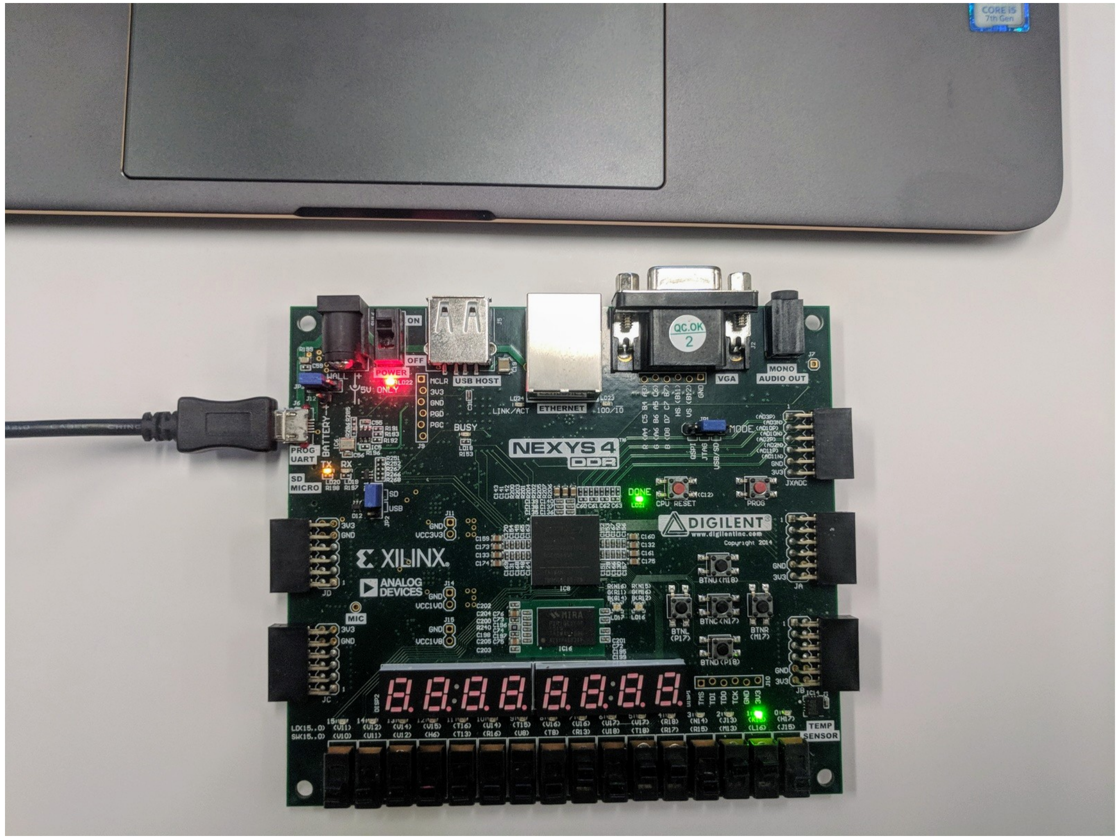
Clash-free pre-defined sparsity leads to no performance degradation

Operational parallelization and junction pipelining



Quick Overview of Hardware Architecture

- Degree of parallelism (z) = Number of weights processed in parallel in a junction
- Connections designed for clash-free memory accesses to prevent stalling
- Clash-free pre-defined sparsity leads to no performance degradation
- Operational parallelization and junction pipelining
- Prototype implemented on FPGA



Quick Overview of Hardware Architecture

Degree of parallelism (z) = Number of weights processed in parallel in a junction

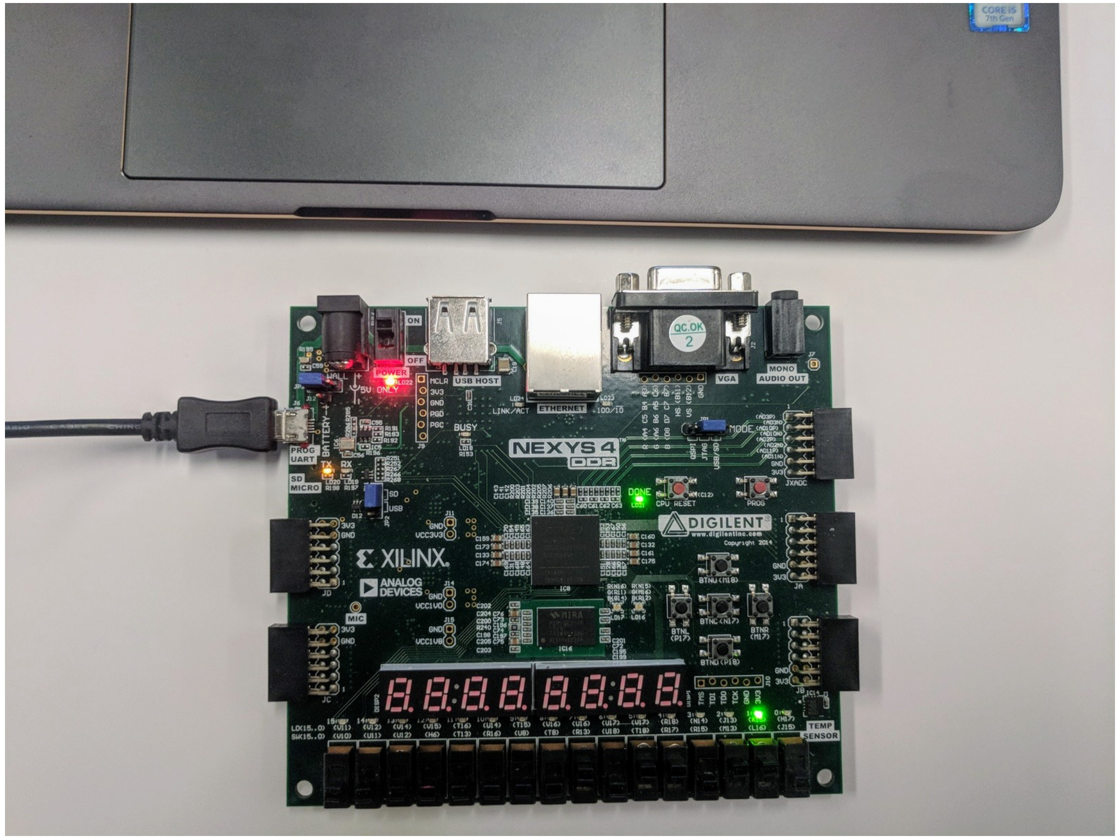
Connections designed for clash-free memory accesses to prevent stalling

Clash-free pre-defined sparsity leads to no performance degradation

Operational parallelization and junction pipelining

Prototype implemented on FPGA

Transferred to and currently being developed by team SAPIENT, in collaboration with DTRA and USC ISI.



Automated Machine Learning : Deep-n-Cheap

https://github.com/souryadey/deep-n-cheap

AutoML (Automated Machine Learning)

- Software frameworks that make design decisions
- Given a problem, **search** for NN models

AutoKeras

Jin 2019 – Auto-Keras

GLUON

AWSLabs 2020 – AutoGluon

AutoML.org Freiburg-Hannover

Mendoza 2018 – Auto-PyTorch

Our Work

DnC Deep-n-Cheap

Low Complexity AutoML framework

Reduce training complexity

Target custom datasets and user requirements

Supports CNNs and MLPs

Framework	Architecture search space	Training hyp search	Adjust model complexity
Auto-Keras	Only pre-existing architectures	No	No
AutoGluon	Only pre-existing architectures	Yes	No
Auto-PyTorch	Customizable by user	Yes	No
Deep-n-Cheap	Customizable by user	Yes	Penalize t_{tr}, N_p

t_{tr} = Training time / epoch

N_p = # Trainable parameters

Search Objective

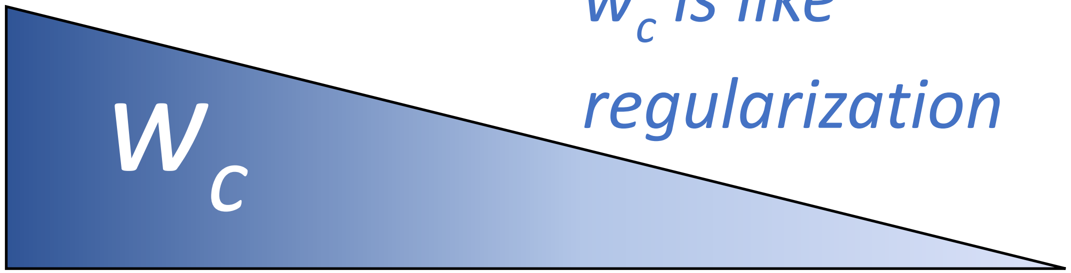
Optimize performance and complexity

Modified loss function: $f(\text{ NN Config } \mathbf{x}\ ) = \log(\ f_p + w_c * f_c\ )$

Example config ***x***:
[#layers, #channels] = [3, (29,40,77)]

f_p = 1 - (Best Validation Accuracy)
f_c = Normalized t_{tr} or N_p
= t_{tr}(config) / t_{tr}(baseline)

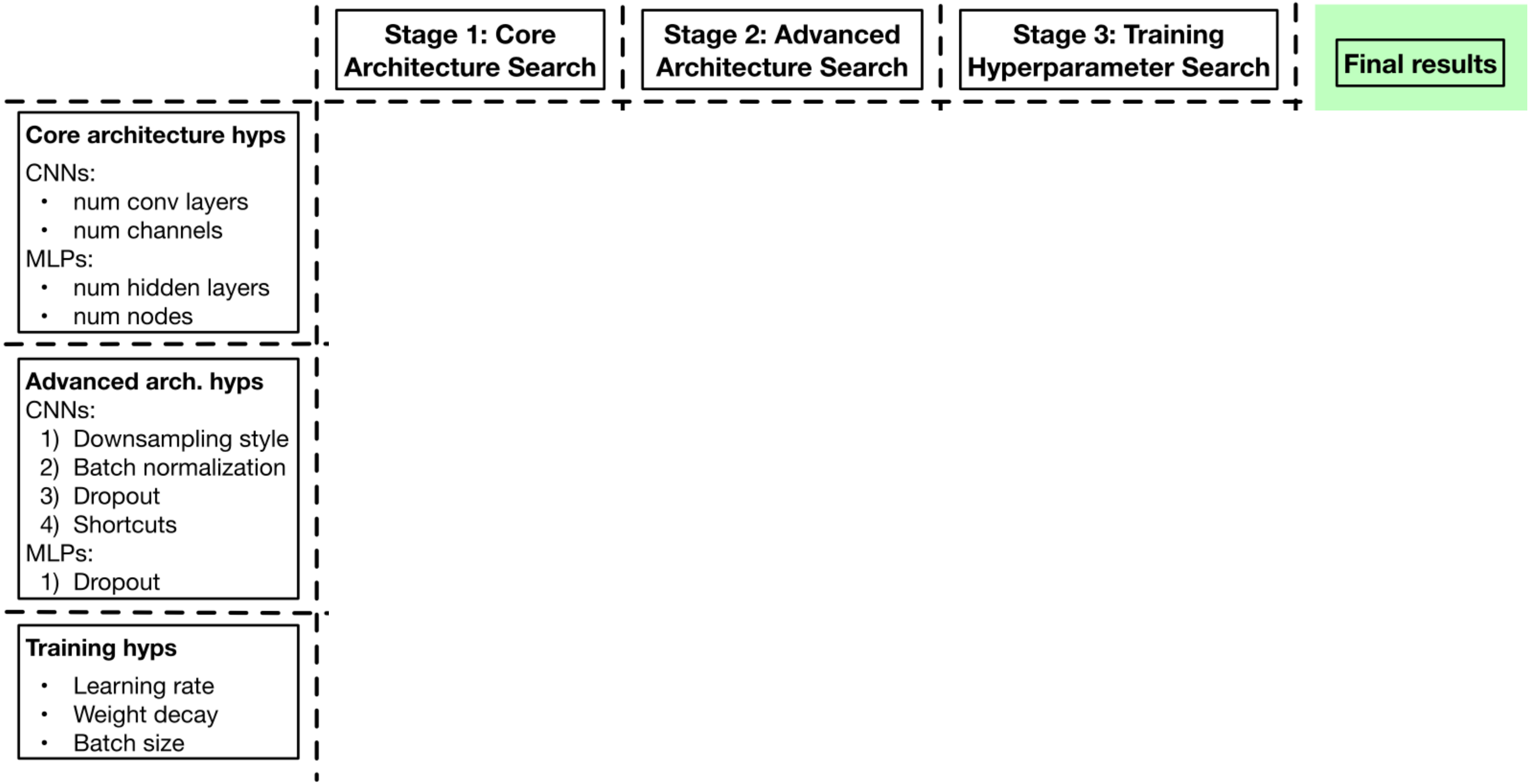
w_c is like regularization



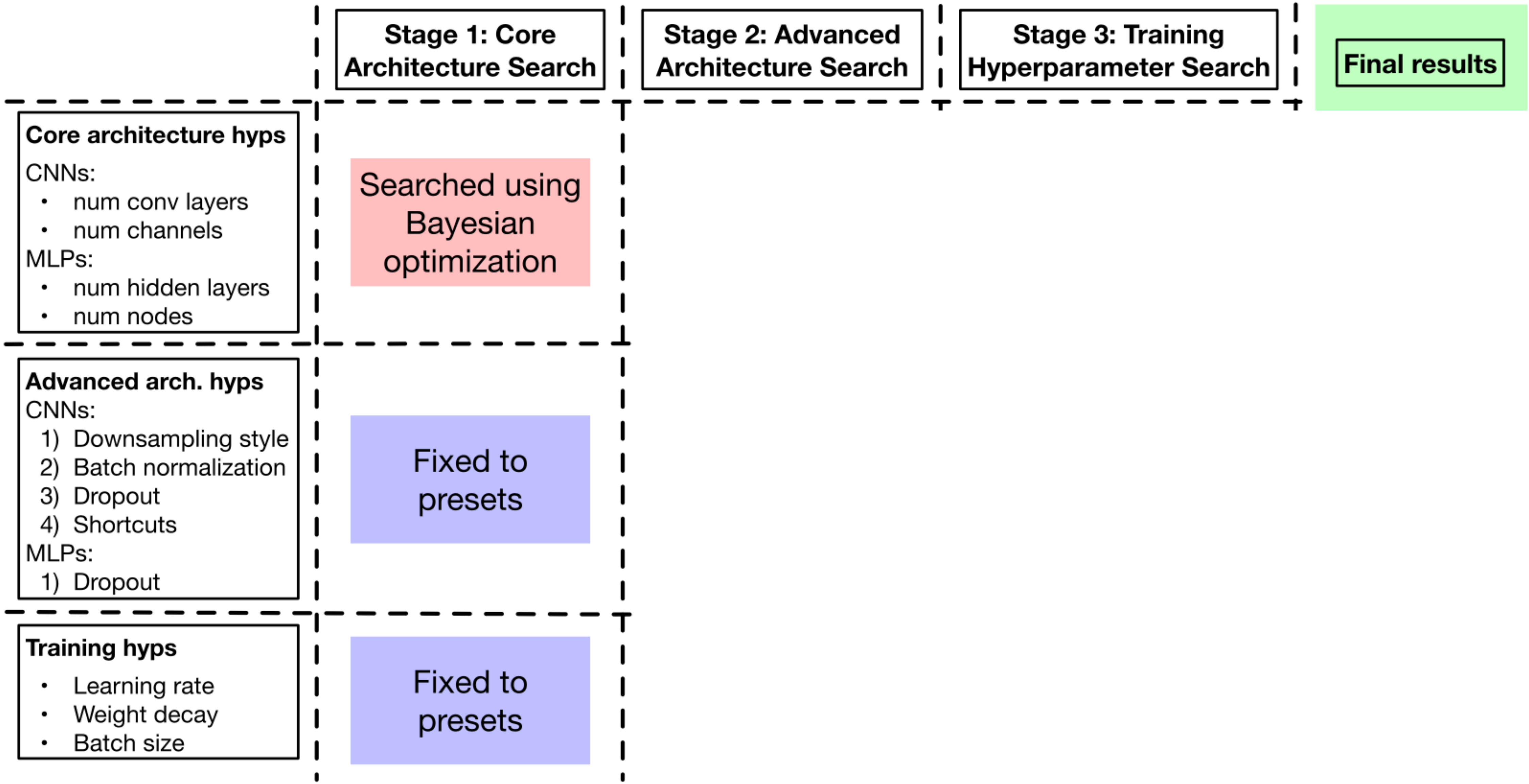
Quick to train
Sacrifice performance

Good performance
Slow to train
Slow search process

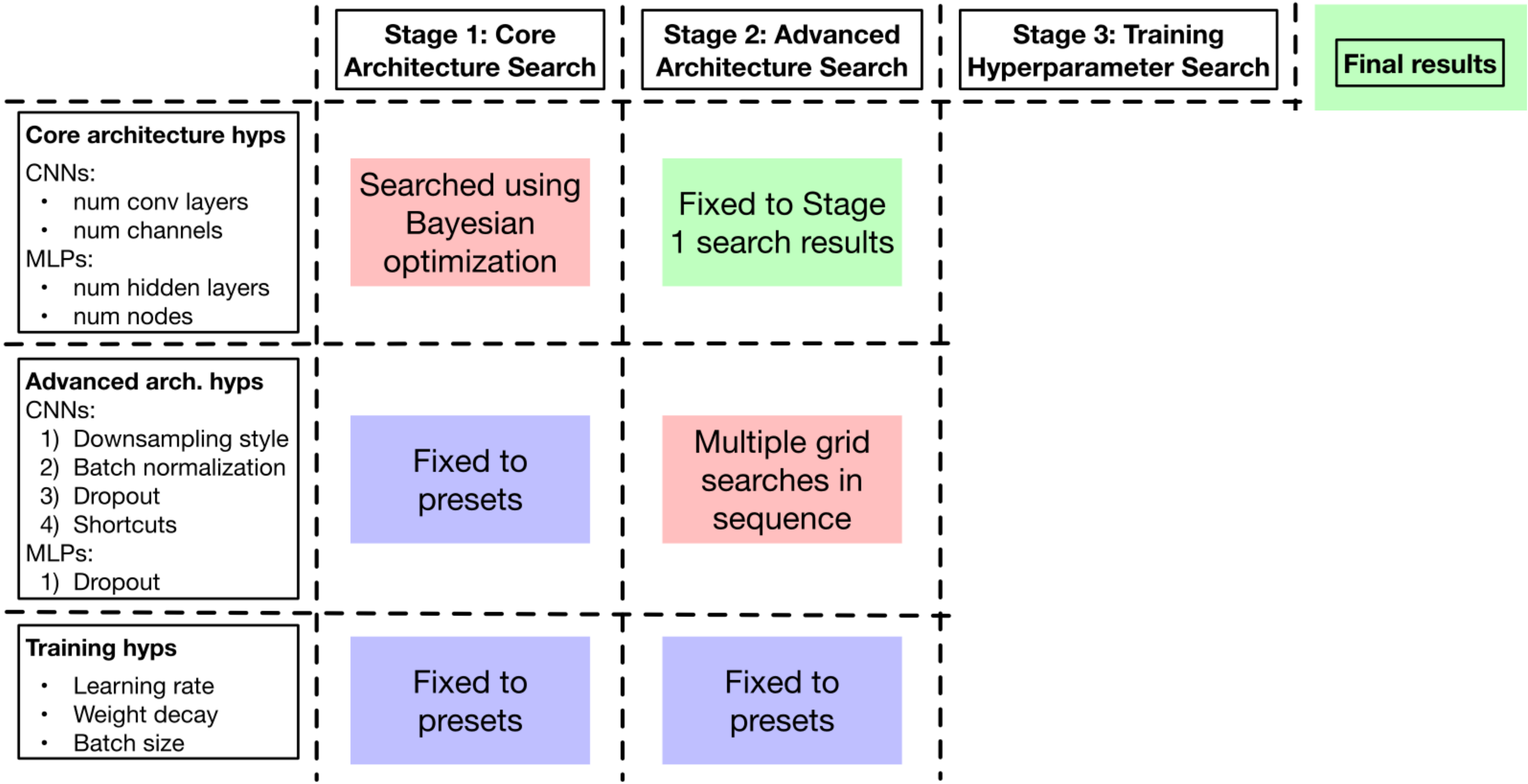
Three-stage search process



Three-stage search process



Three-stage search process



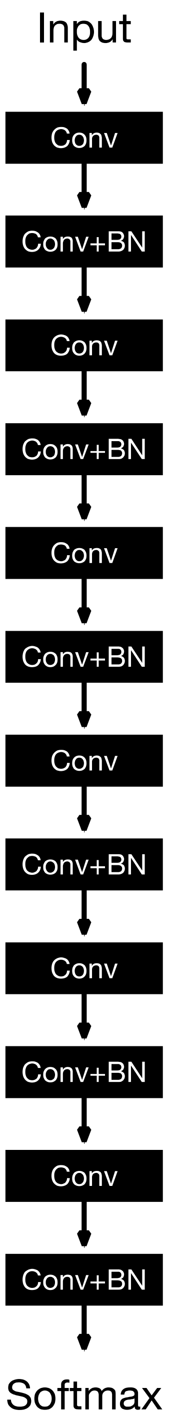
Three-stage search process

	Stage 1: Core Architecture Search	Stage 2: Advanced Architecture Search	Stage 3: Training Hyperparameter Search	Final results
Core architecture hyps CNNs: • num conv layers • num channels MLPs: • num hidden layers • num nodes	Searched using Bayesian optimization	Fixed to Stage 1 search results	Fixed to Stage 1 search results	Stage 1 search results
Advanced arch. hyps CNNs: 1) Downsampling style 2) Batch normalization 3) Dropout 4) Shortcuts MLPs: 1) Dropout	Fixed to presets	Multiple grid searches in sequence	Fixed to Stage 2 search results	Stage 2 search results
Training hyps • Learning rate • Weight decay • Batch size	Fixed to presets	Fixed to presets	Searched using Bayesian optimization	Stage 3 search results

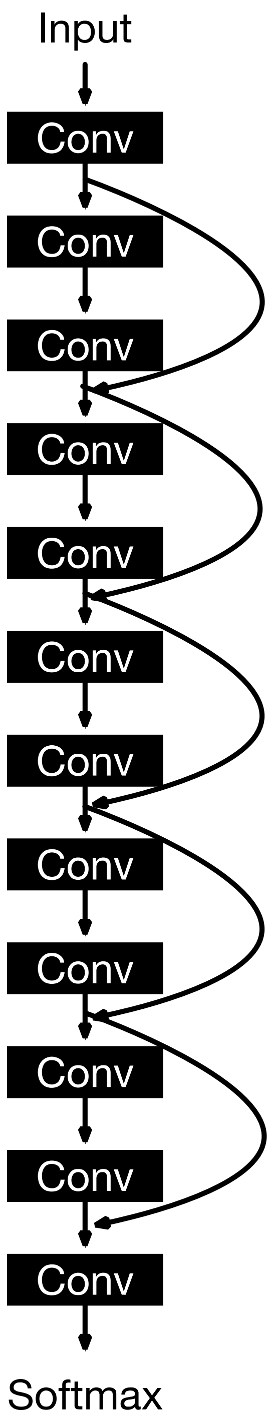
ARCHITECTURE + TRAINING

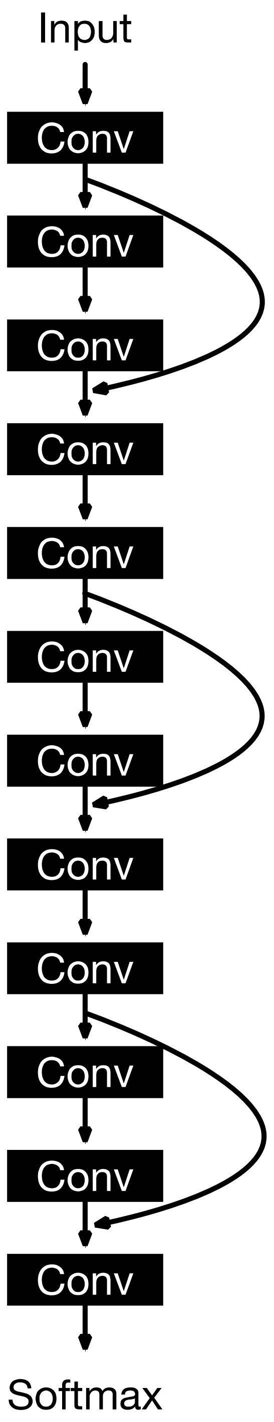
Examples of Stage 2

BN = 0.5



Full shortcuts (left)
Half shortcuts (right)





Bayesian Optimization Workflow

Model function f

- *Sample* some initial data $\mathbf{X}_{1:n1}$ and find $f(\mathbf{X}_{1:n1})$
- Form prior to approximate f. This is a *Gaussian process* with $\boldsymbol{\mu}_{n1x1}$, $\boldsymbol{\Sigma}_{n1xn1}$
- Repeat for n2 steps:
  - Sample new points $\mathbf{X'}_{1:n3}$
  - Find *expected improvement* EI($\mathbf{x}'$) for each new point and choose $\mathbf{x}_{n1+1}$ = argmax EI($\mathbf{x}'$)
  - Form *posterior* to approximate f :
    - Augment $\mathbf{X}_{1:n1}$ to $\mathbf{X}_{1:n1+1}$
    - Find $f(\mathbf{x}_{n+1})$
    - Augment $\boldsymbol{\mu}_{n1x1}$ to $\boldsymbol{\mu}_{(n1+1)x1}$, $\boldsymbol{\Sigma}_{n1xn1}$ to $\boldsymbol{\Sigma}_{(n1+1)x(n1+1)}$
- Finally, return best f and corresponding best **x**

*Total configs explored: n1 + n2*n3*

Total configs trained: n1 + n2

Gaussian process (GP)

A collection of random variables such that any subset of them forms a multi-dimensional Gaussian random vector

$$f\left(\boldsymbol{X}_{1:n}\right) \sim \mathcal{N}\left(\underset{n\times 1}{\boldsymbol{\mu}}, \underset{n\times n}{\boldsymbol{\Sigma}}\right)$$

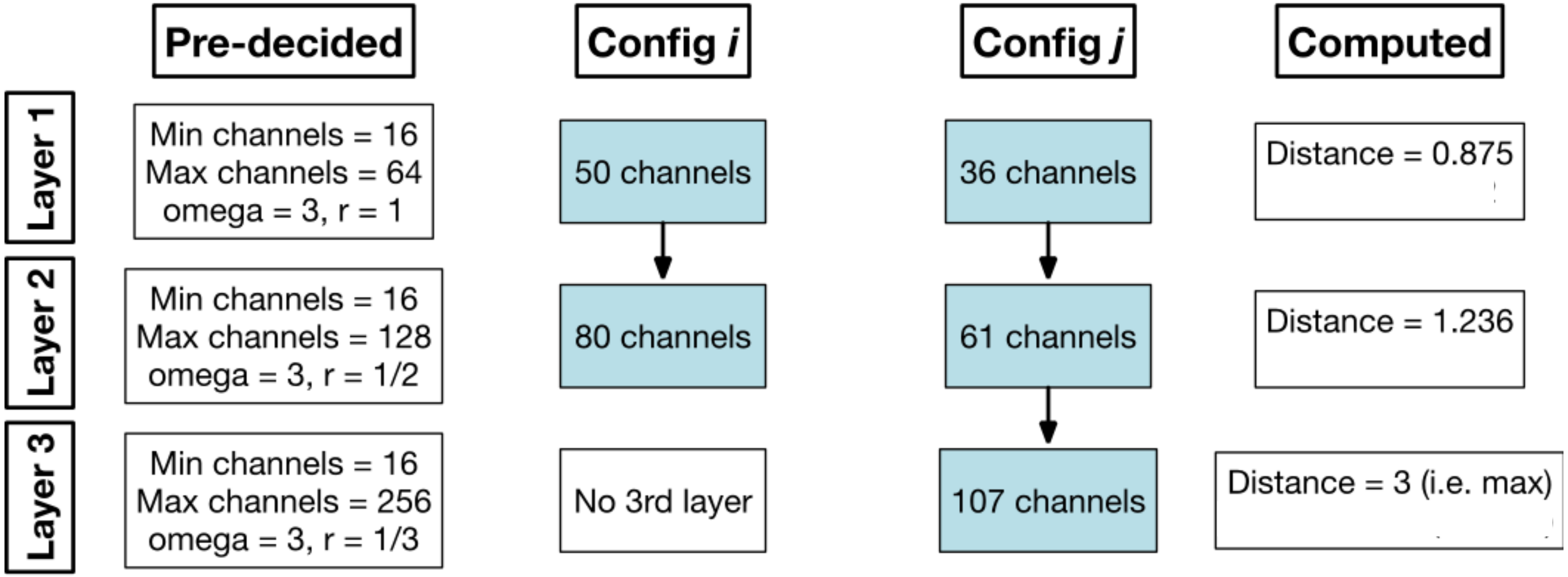
$$\boldsymbol{\mu} = \begin{bmatrix} \mu\left(\boldsymbol{x}_1\right) \\ \vdots \\ \mu\left(\boldsymbol{x}_n\right) \end{bmatrix}$$

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma\left(\boldsymbol{x}_1, \boldsymbol{x}_1\right) & \cdots & \sigma\left(\boldsymbol{x}_1, \boldsymbol{x}_n\right) \\ \vdots & \ddots & \vdots \\ \sigma\left(\boldsymbol{x}_n, \boldsymbol{x}_1\right) & \cdots & \sigma\left(\boldsymbol{x}_n, \boldsymbol{x}_n\right) \end{bmatrix}$$

Covariance kernel – Similarity between NN configs

Individual Distance

$$d\left(x_{ik}, x_{jk}\right) = \omega_k \left(\frac{|x_{ik} - x_{jk}|}{u_k - l_k}\right)^{r_k}$$



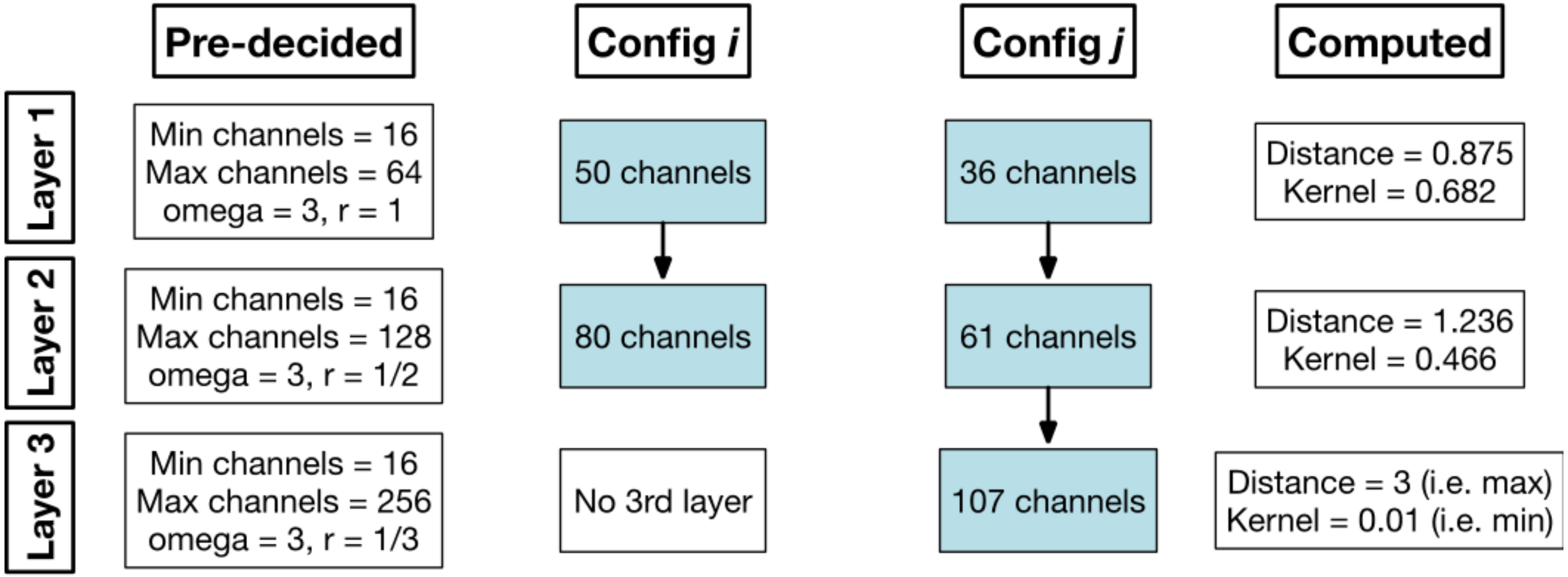
Covariance kernel – Similarity between NN configs

Individual Distance

$$d\left(x_{ik}, x_{jk}\right) = \omega_k \left(\frac{|x_{ik} - x_{jk}|}{u_k - l_k}\right)^{r_k}$$

Individual Kernel

$$\sigma\left(x_{ik}, x_{jk}\right) = \exp\left(-\frac{d^2(x_{ik}, x_{jk})}{2}\right)$$



Covariance kernel – Similarity between NN configs

Individual Distance

$$d\left(x_{ik}, x_{jk}\right) = \omega_k \left(\frac{|x_{ik} - x_{jk}|}{u_k - l_k}\right)^{r_k}$$

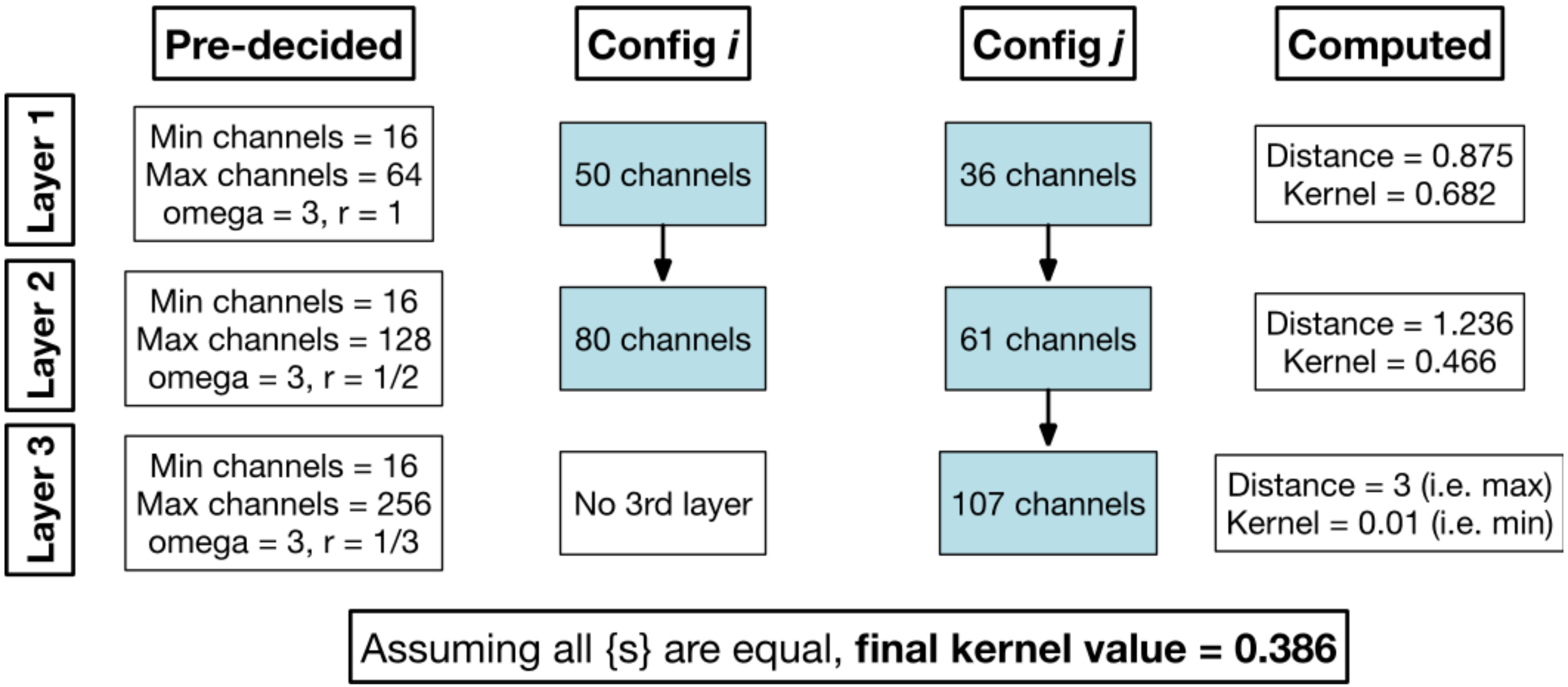
Individual Kernel

$$\sigma\left(x_{ik}, x_{jk}\right) = \exp\left(-\frac{d^2\left(x_{ik}, x_{jk}\right)}{2}\right)$$

Complete Kernel

$$\sigma\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \sum_{k=1}^{K} s_k \sigma\left(x_{ik}, x_{jk}\right)$$

Convex combination



Expected Improvement (EI)

- Let f* be the minimum of all observed values so far
- *How much can a new point **x** improve:*
  - If f(**x**) > f*, Imp(x) = 0
  - Else, Imp(x) = f*-f(**x**)
- EI(**x**) = Expectation [max(f*-f(**x**),0)]

$$EI(\boldsymbol{x}) = (f^* - \mu)P\left(\frac{f^* - \mu}{\sigma}\right) + \sigma p\left(\frac{f^* - \mu}{\sigma}\right)$$

Standard normal cdf = P, pdf = p

*Don't need to evaluate f(**x**) to find EI(**x**)*

Data loader and augmentation considerations

Using data pre-loaded from npz format
Entire dataset is in memory

```
data = np.load(‘mnist.npz’)
xtr, ytr = data[‘xtr’], data[‘ytr’]
for i in numbatches:
    inputs = xtr[i*batch_size : (i+1)*batch_size]
    labels = ytr[i*batch_size : (i+1)*batch_size]
```

Using Pytorch data loaders
Uses generators to not burden memory

```
data = torchvision.datasets.MNIST(root = data_folder, train = True, download = False, transform = transforms.ToTensor())
train_loader = torch.utils.data.DataLoader(data['train'], batch_size = batch_size, shuffle = True, num_workers = 4,
                                     pin_memory = True)
for batch in train_loader:
    inputs, labels = batch
```

npz is faster, data loaders are more versatile

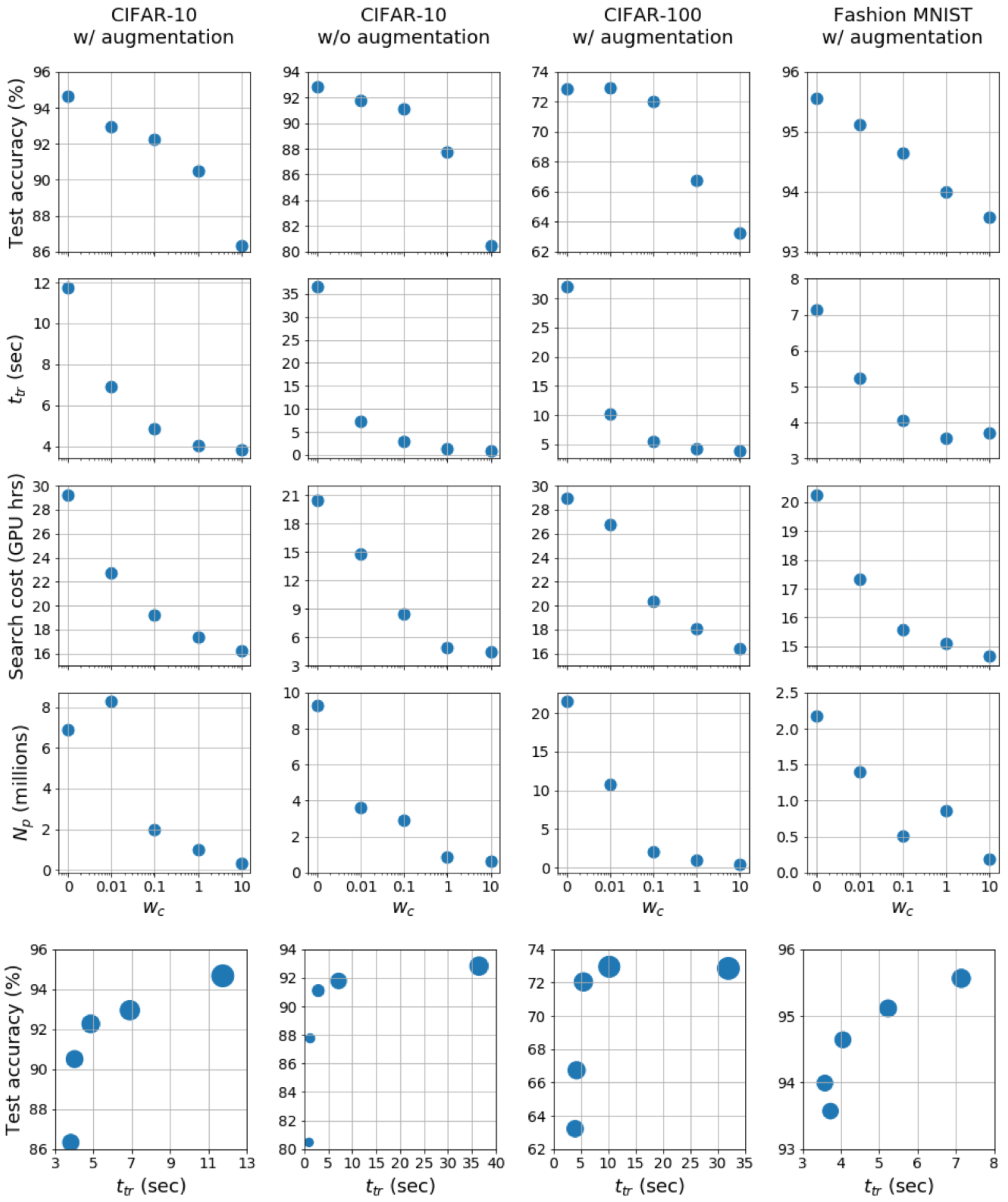
CNN Results

Complexity Penalty = Training time / epoch

AWS p3.2xlarge with 1 V100 GPU

We are not penalizing this, but it's correlated

Performance-complexity tradeoff



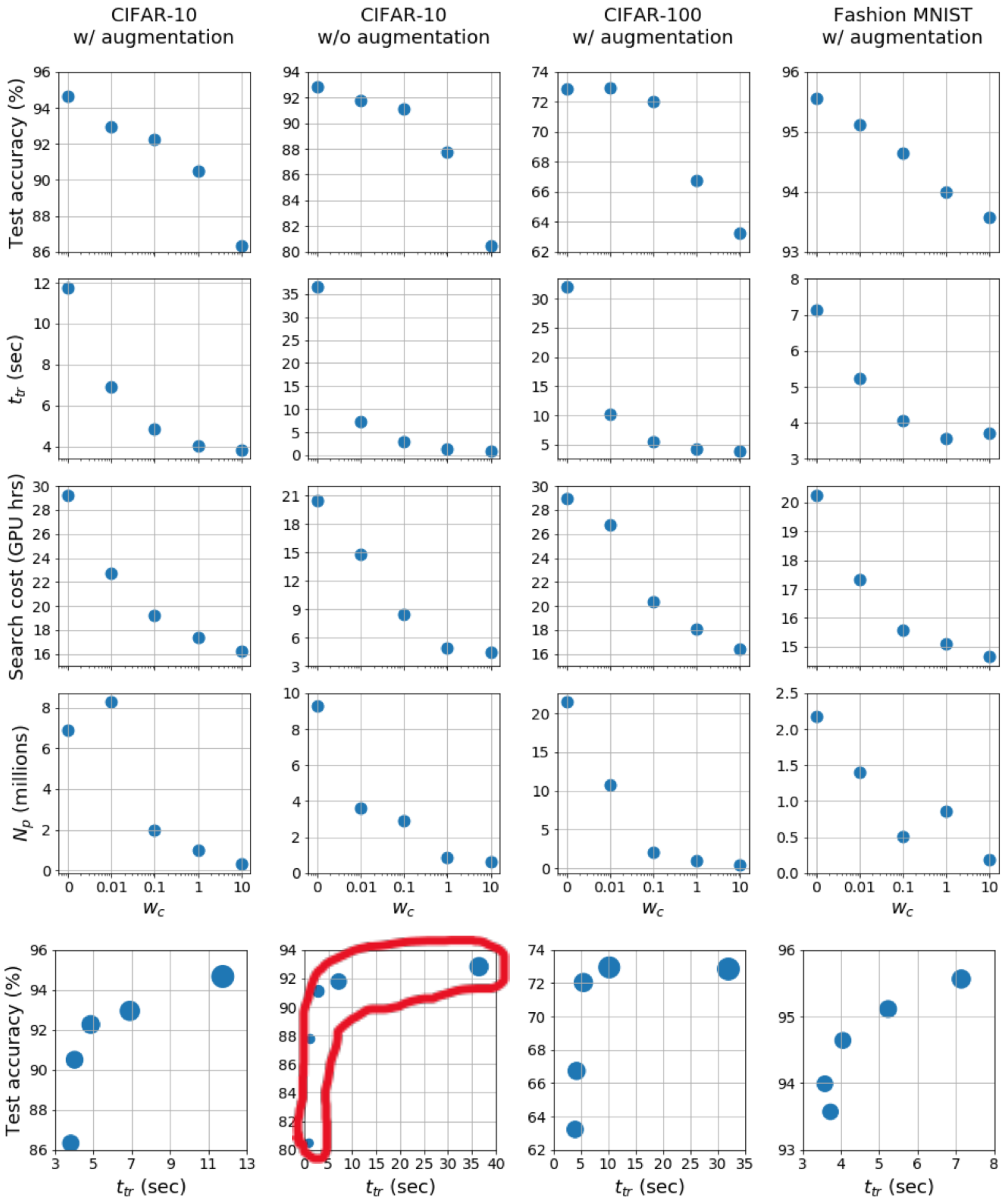
CNN Results

Complexity Penalty = Training time / epoch

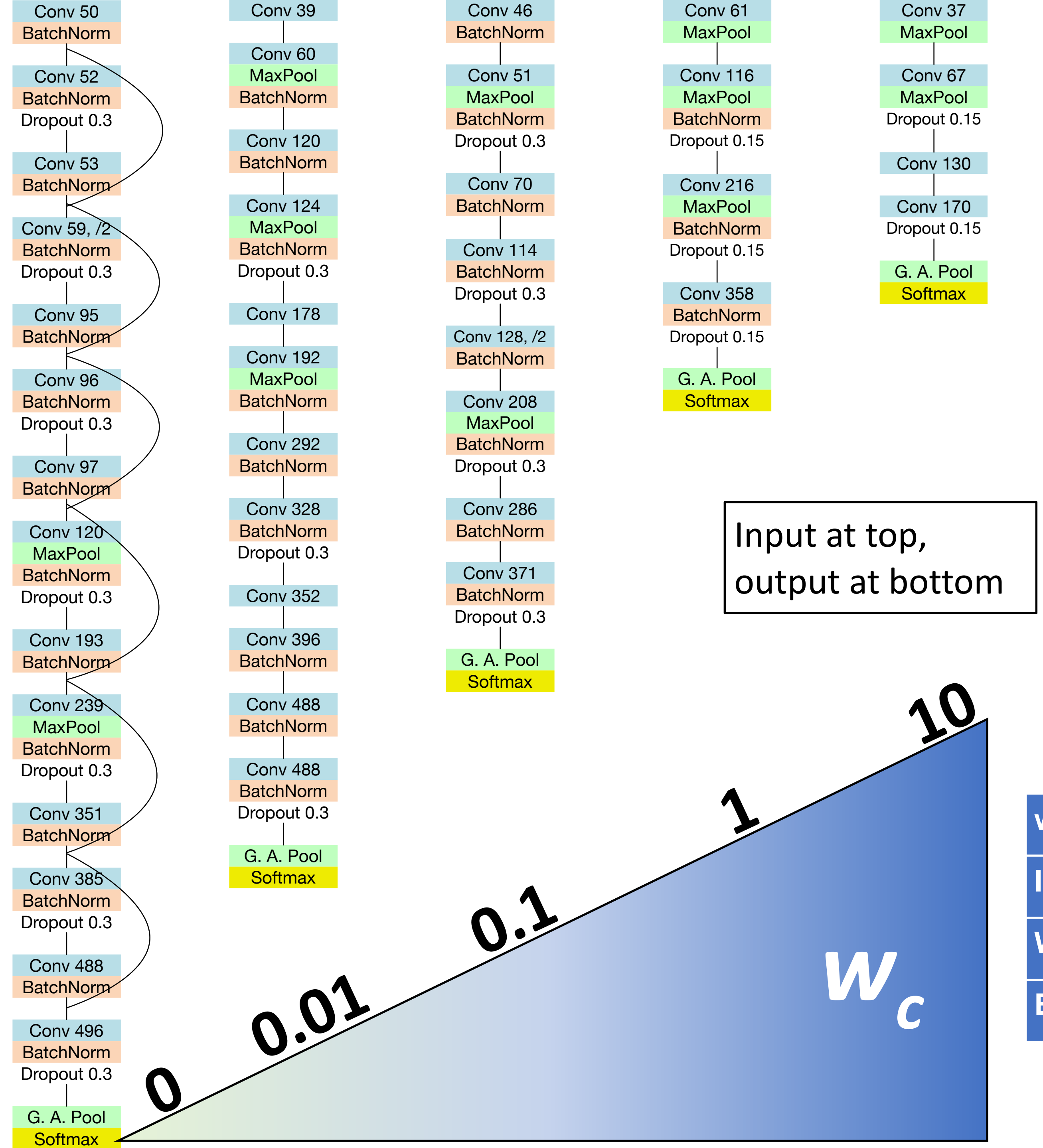
AWS p3.2xlarge with 1 V100 GPU

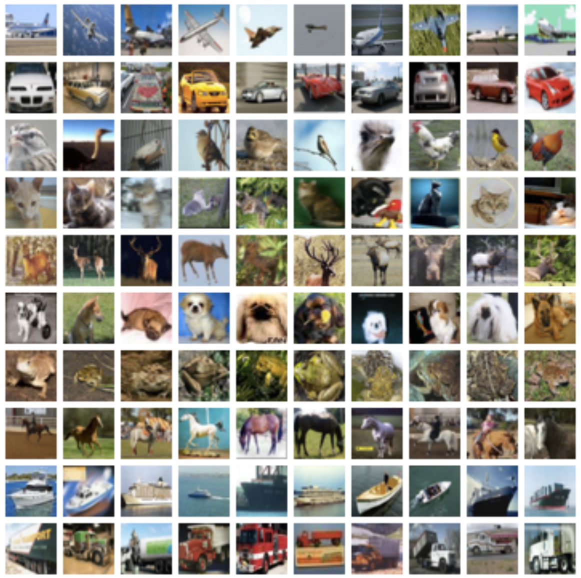
We are not penalizing this, but it's correlated

Performance-complexity tradeoff



CIFAR-10 w/ aug





w_c	0	0.01	0.1	1	10
Initial learning rate η	0.001	0.001	0.001	0.003	0.001
Weight decay λ	3.3 x 10^{-5}	8.3 x 10^{-5}	1.2 x 10^{-5}	0	0
Batch size	120	256	459	452	256

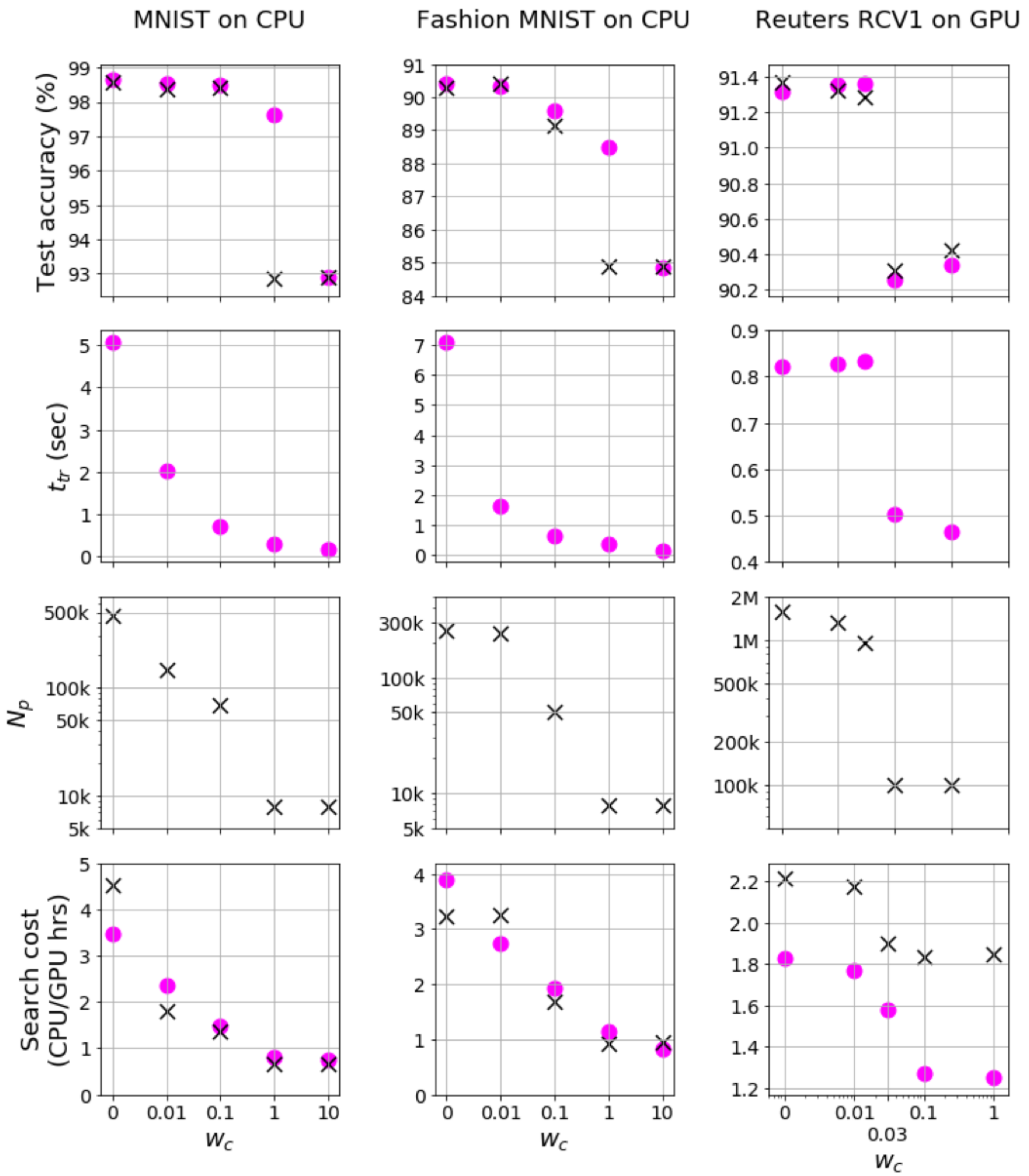
λ strictly correlated with N_p

MLP Results

Pink dots:
Complexity Penalty =
Training time / epoch

Black crosses:
Complexity Penalty =
 # Trainable Params

CPU = Macbook Pro with 8GB RAM, no CuDA
GPU = (Same) AWS p3.2xlarge with V100

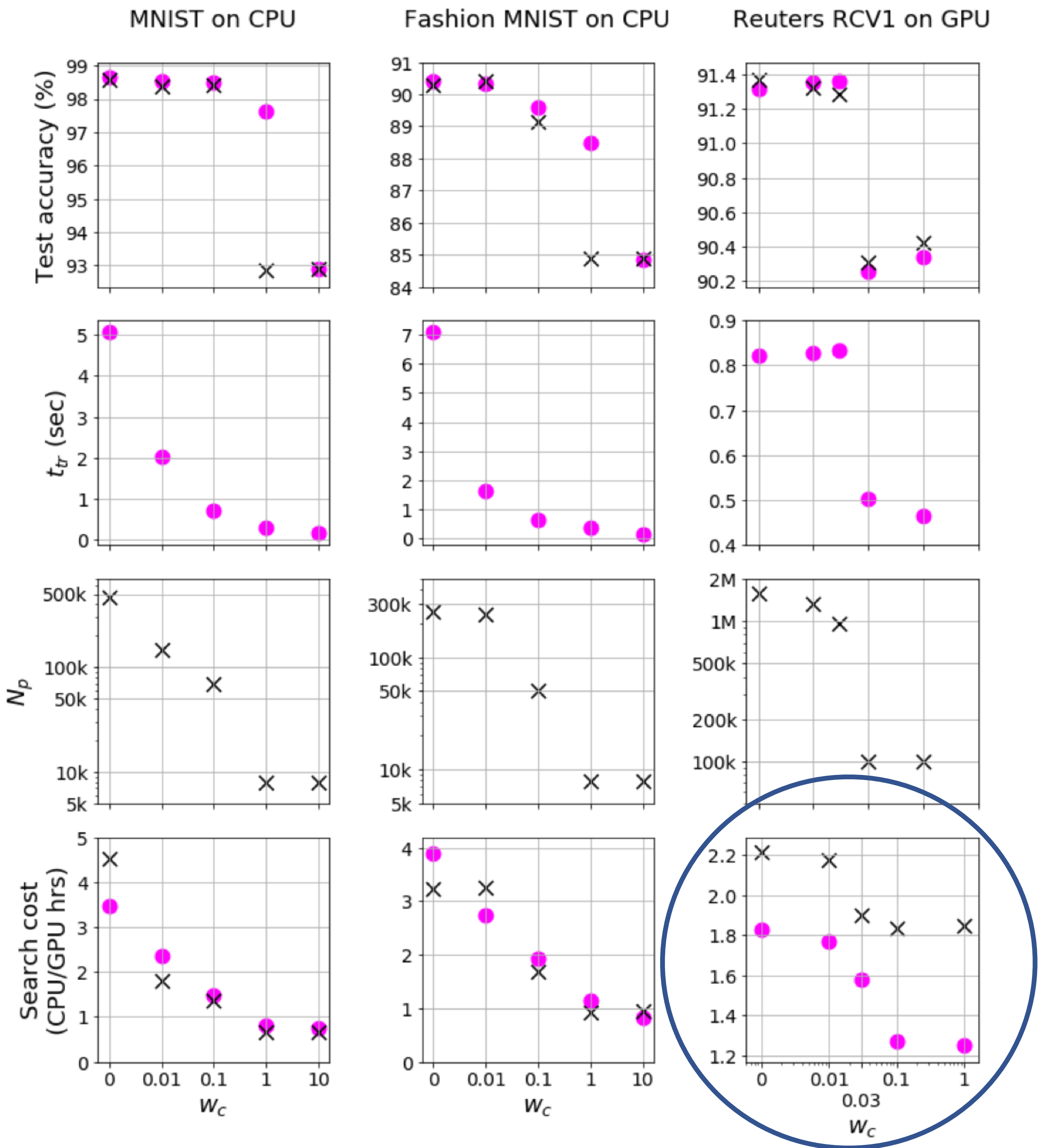


MLP Results

Pink dots:
Complexity Penalty =
Training time / epoch

Black crosses:
Complexity Penalty =
 # Trainable Params

CPU = Macbook Pro with 8GB RAM, no CuDA
GPU = (Same) AWS p3.2xlarge with V100



Running Deep-n-Cheap

How to run?

- Install Python 3
- Install Pytorch

```
$ pip install sobol_seq tqdm
$ git clone https://github.com/souryadey/deep-n-cheap.git
$ cd deep-n-cheap
$ python main.py
```

For **help:**

```
$ python main.py -h
```

Datasets (including custom)

Set `dataset` to either:

- `--dataset=torchvision.datasets.<dataset>`. Currently supported values of `<dataset>` = MNIST, FashionMNIST, CIFAR10, CIFAR100
- `--dataset='<dataset>.npz'`, where `<dataset>` is a `.npz` file with 4 keys:
  - `xtr` : numpy array of shape (num_train_samples, num_features...), example (50000,3,32,32) or (60000,784). Image data should be in *channels_first* format.
  - `ytr` : numpy array of shape (num_train_samples,)
  - `xte` : numpy array of shape (num_test_samples, num_features...)
  - `yte` : numpy array of shape (num_test_samples,)
- Some datasets can be downloaded from the links in `dataset_links.txt`. Alternatively, define your own **custom datasets.**

Comparison (CNNs on CIFAR-10)

Framework	Additional settings	Search cost (GPU hrs)	Best model found from search			
			Architecture	t_{tr} (sec)	Batch size	Best val acc (%)
Proxyless NAS	Proxyless-G	96	537 conv layers	429	64	93.22
Auto-Keras	Default run	14.33	Resnet-20 v2	33	32	74.89
AutoGluon	Default run	**3**	Resnet-20 v1	37	64	88.6
	Extended run	101	Resnet-56 v1	46	64	91.22
Auto-Pytorch	'tiny cs'	6.17	30 conv layers	39	64	87.81
	'full cs'	6.13	41 conv layers	31	106	86.37
Deep-n-Cheap	$w_c = 0$	29.17	14 conv layers	10	120	**93.74**
	$w_c = 0.1$	19.23	8 conv layers	4	459	91.89
	$w_c = 10$	16.23	4 conv layers	**3**	256	83.82

Penalizes inference complexity, <u>not</u> training

Auto Keras and Gluon don't support getting final model out, so we compared on best val acc found during search instead of final test acc

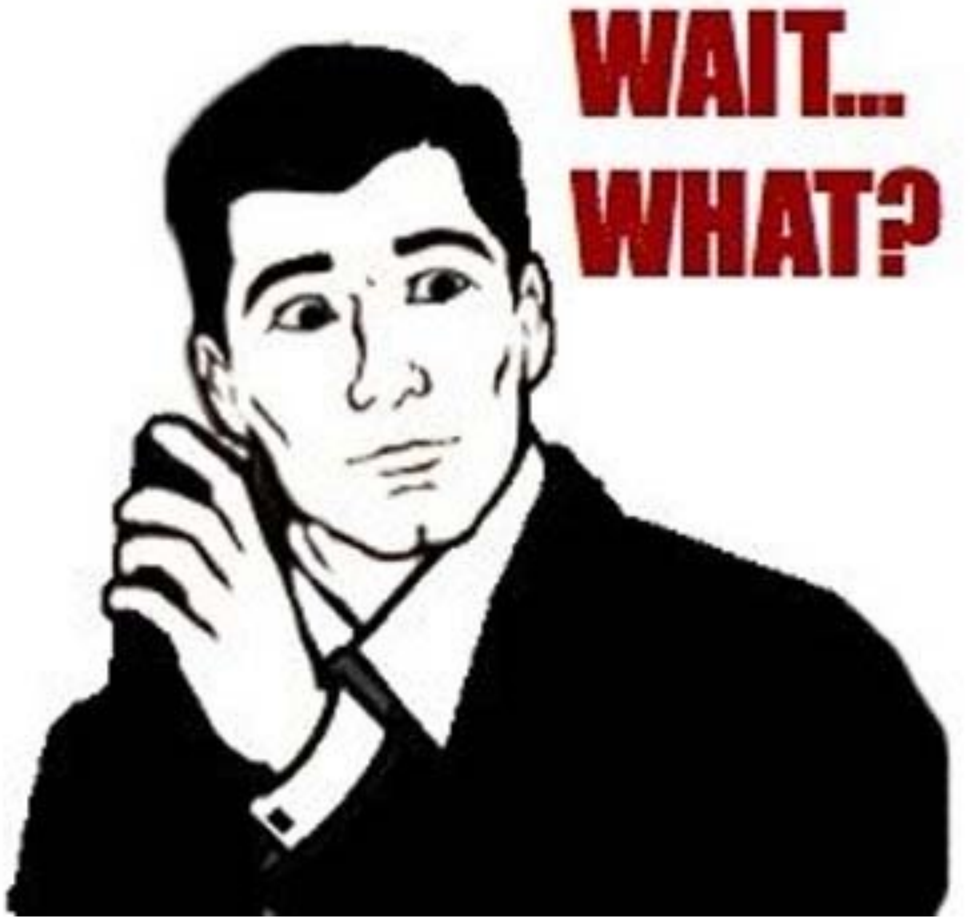
Comparison (MLPs)

Framework	Additional settings	Search cost (GPU hrs)	Best model found from search				
			MLP layers	N_p	t_{tr} (sec)	Batch size	Best val acc (%)
Fashion MNIST							
Auto-Pytorch	'tiny cs'	6.76	50	27.8M	19.2	125	**91**
	'medium cs'	5.53	20	3.5M	8.3	184	90.52
	'full cs'	6.63	12	122k	5.4	173	90.61
Deep-n-Cheap (penalize t_{tr})	$w_c = 0$	0.52	3	263k	0.4	272	90.24
	$w_c = 10$	**0.3**	1	**7.9k**	**0.1**	511	84.39
Deep-n-Cheap (penalize N_p)	$w_c = 0$	0.44	2	317k	0.5	153	90.53
	$w_c = 10$	0.4	1	**7.9k**	0.2	256	86.06
Reuters RCV1							
Auto-Pytorch	'tiny cs'	7.22	38	19.7M	39.6	125	88.91
	'medium cs'	6.47	11	11.2M	22.3	337	90.77
Deep-n-Cheap (penalize t_{tr})	$w_c = 0$	1.83	2	1.32M	0.7	503	**91.36**
	$w_c = 1$	**1.25**	1	**100k**	**0.4**	512	90.34
Deep-n-Cheap (penalize N_p)	$w_c = 0$	2.22	2	1.6M	0.6	512	**91.36**
	$w_c = 1$	1.85	1	**100k**	5.54	33	90.4

Takeaway

We may not need very deep networks!

Also see Zagoruyko 2016 – WRN

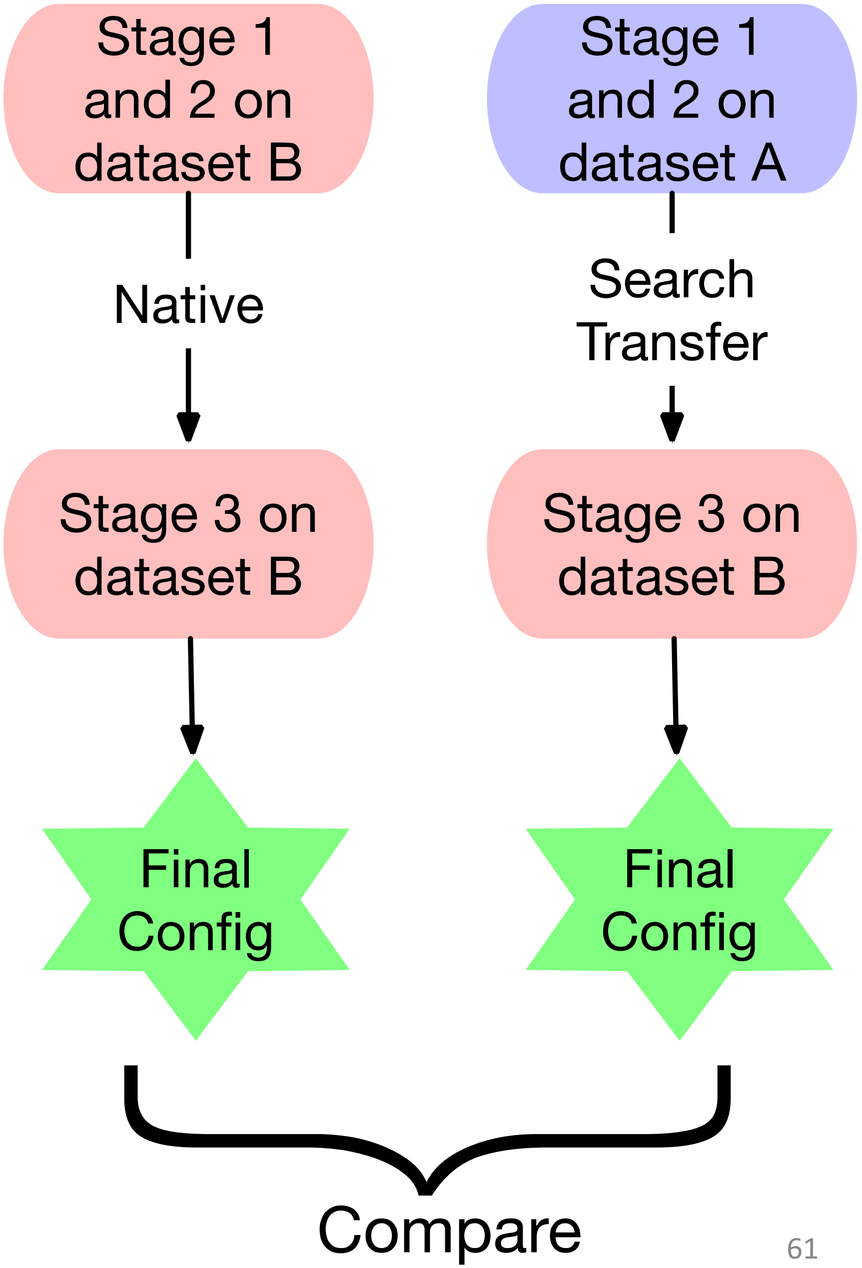


Search transfer

Can a NN architecture found after stages 1 and 2 on dataset A be applied to dataset B after running Stage 3 training hyperparameter search?

How does it compare to native search on dataset B?

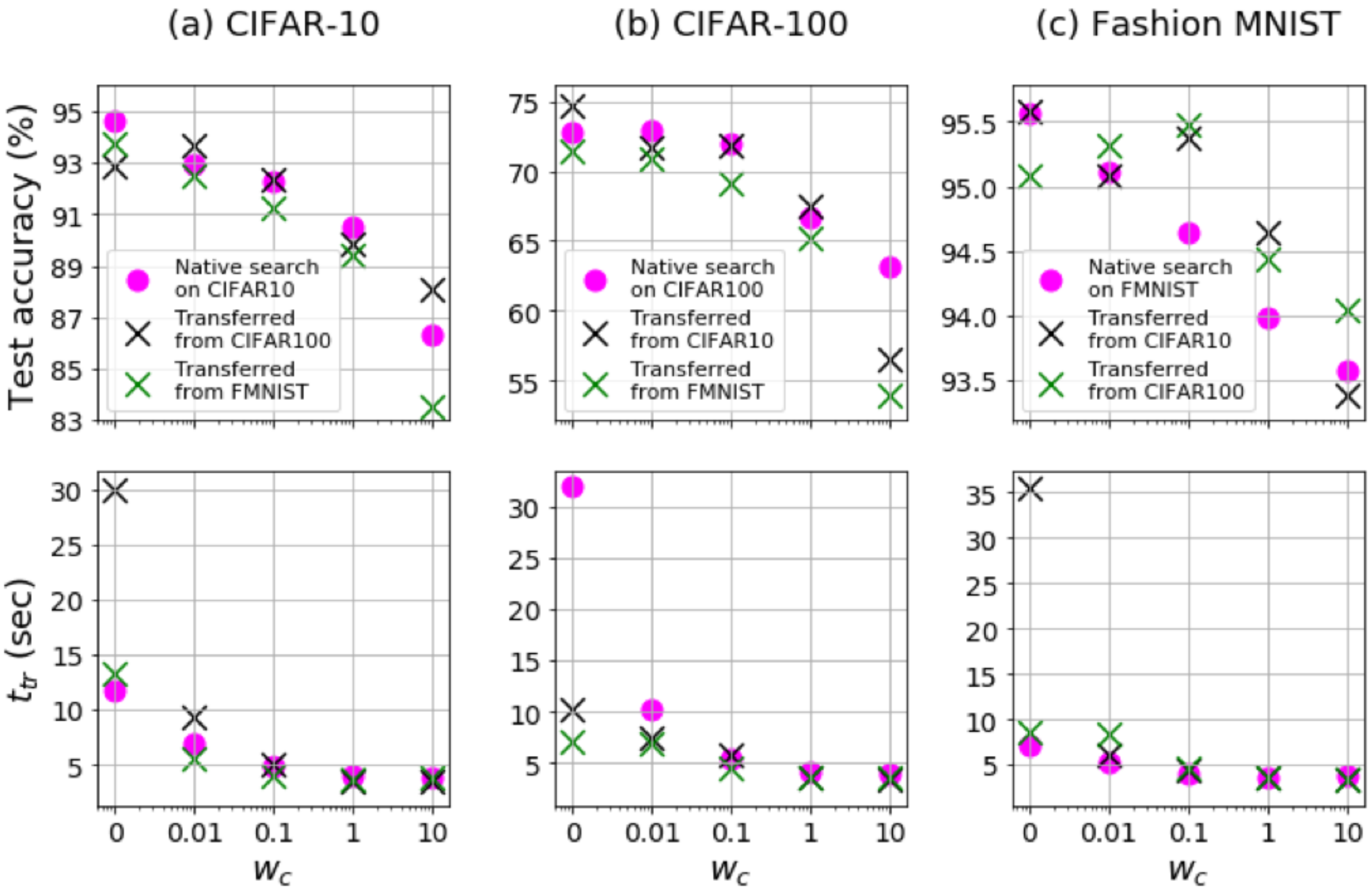
Can architectures generalize?



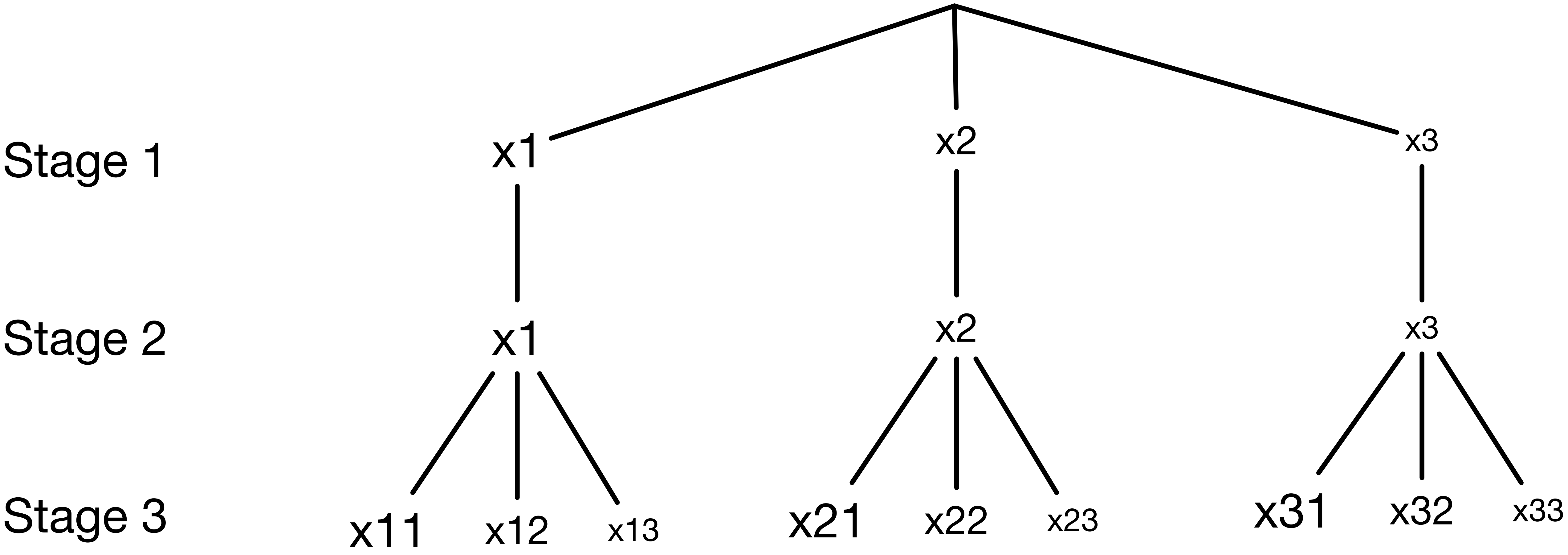
Search transfer results

Transferring from CIFAR outperforms native FMNIST (probably due to more params)

Training times mostly the same

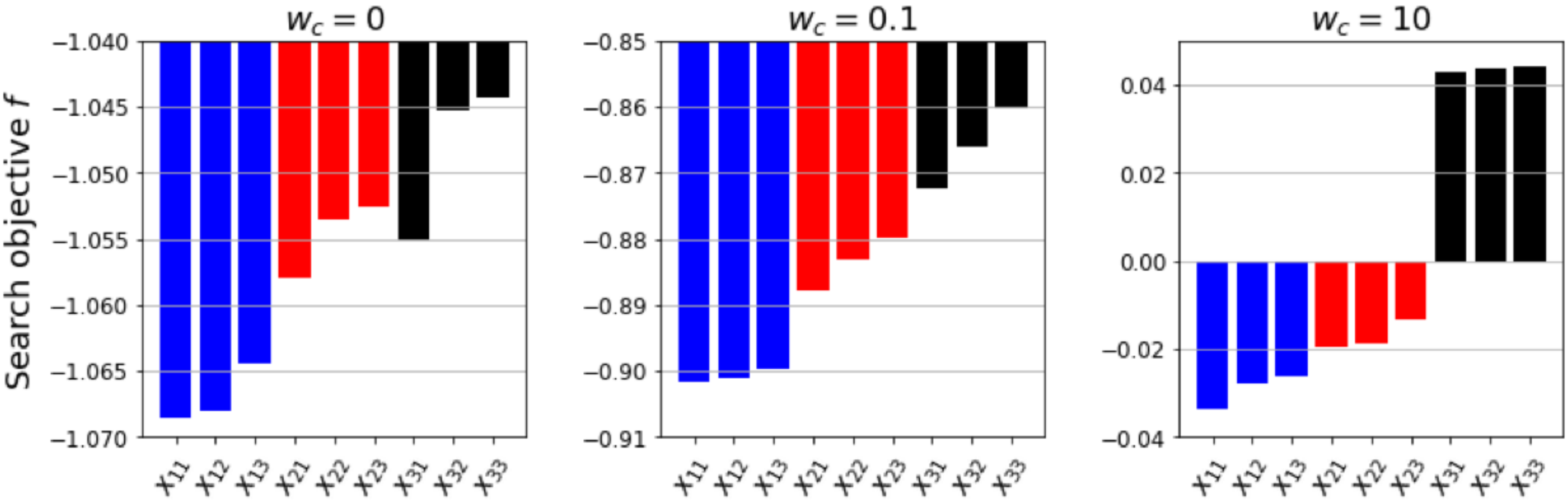


What about a non-greedy search?

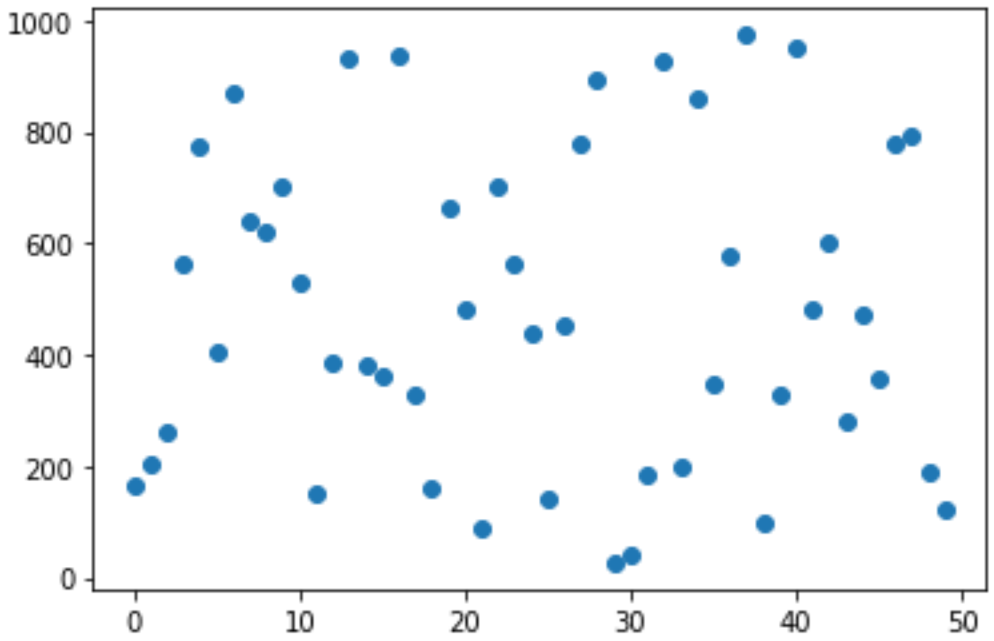


Greedy

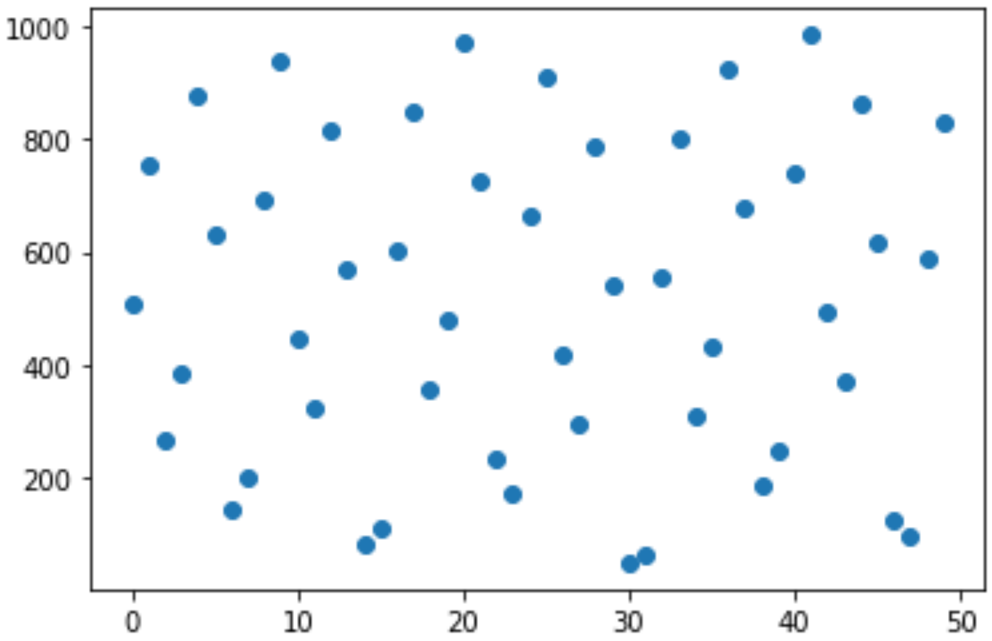
Justifying our greed



Choosing initial points in Bayesian optimization



Random sampling

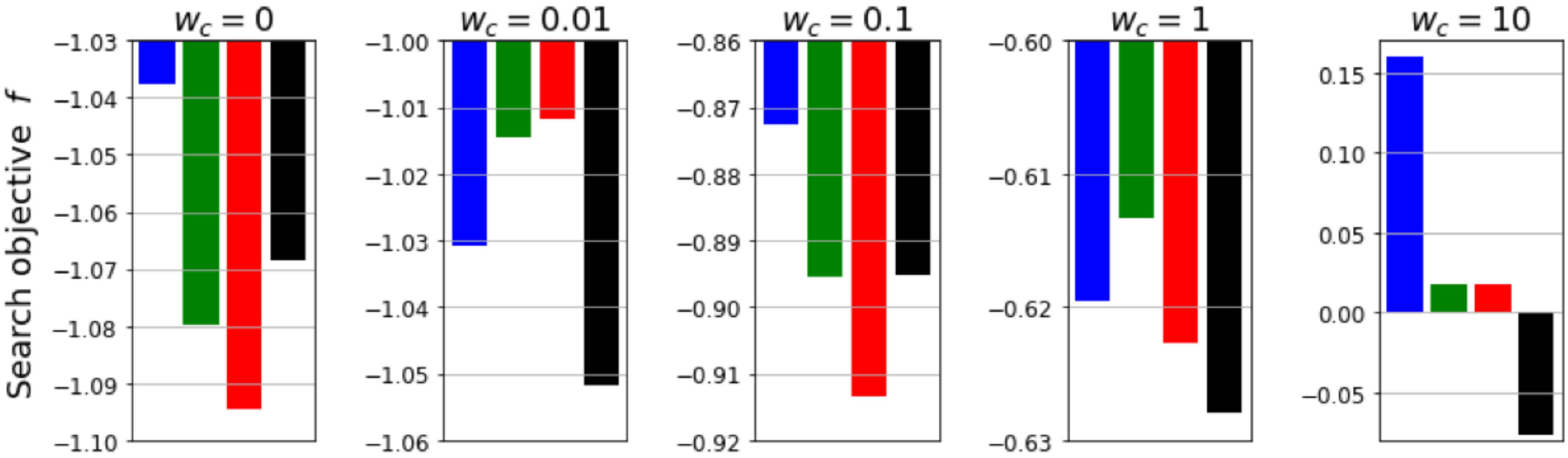


Sobol sampling

Like grid search

Better for more dimensions

BO vs random and grid search (30 points each)



Purely random search: 30 prior

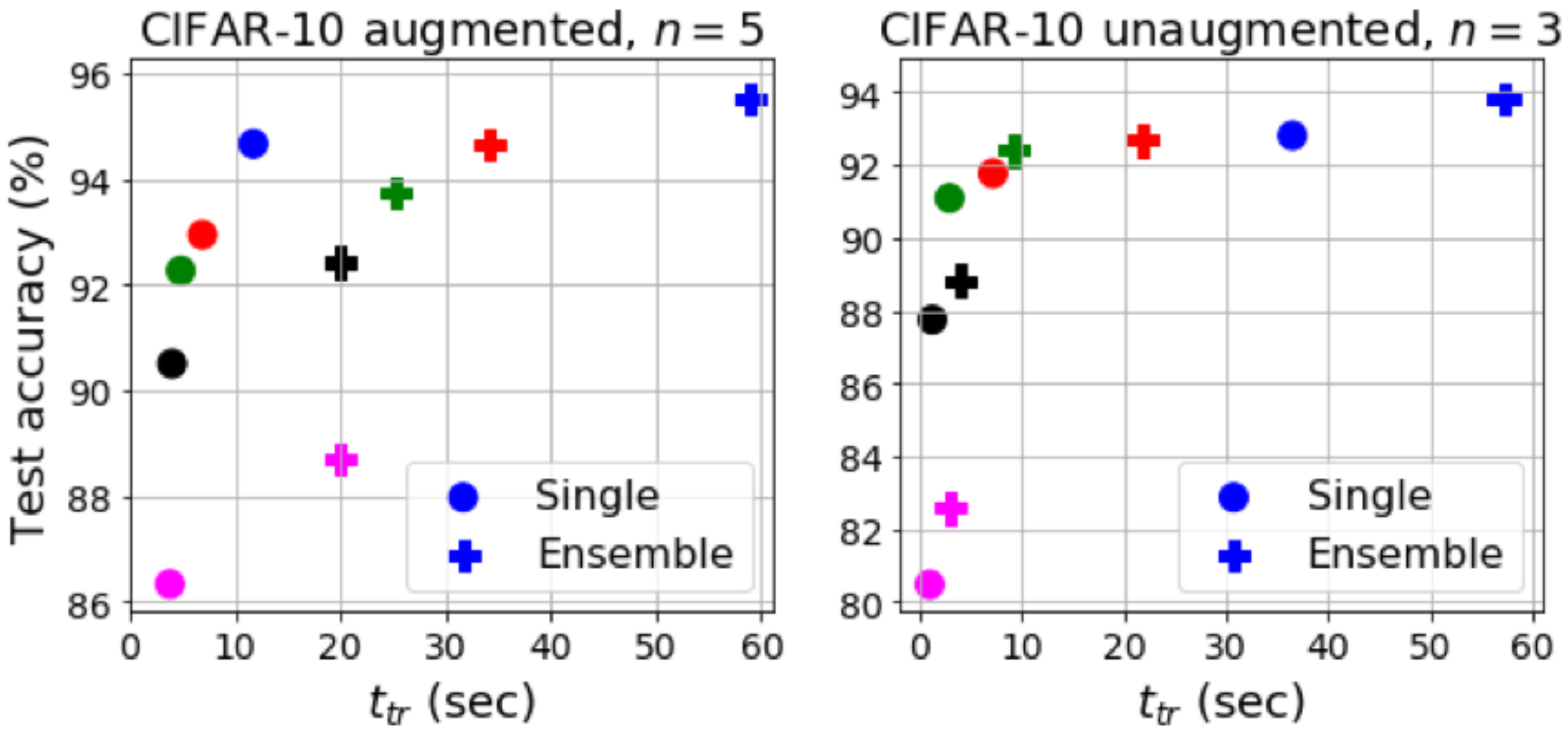
Purely grid search (Sobol): 30 prior

Balanced BO: 15 prior + 15 steps

Extreme BO: 1 prior + 29 steps

Ensembling

Multiple models vote on final test samples



Slight increases in performance at the cost of large increases in complexity

DnC releases

Latest release

v1.0

bb30e55

Verified

Compare

First release

Edit

souryadey released this 21 days ago · 7 commits to master since this release

Version used for obtaining results for the paper -- S. Dey, S. C. Kanala, K. M. Chugg and P. A. Beerel, "Deep-n-Cheap: An Automated Search Framework for Low Complexity Deep Learning", submitted to ECML-PKDD 2020.

Assets 2

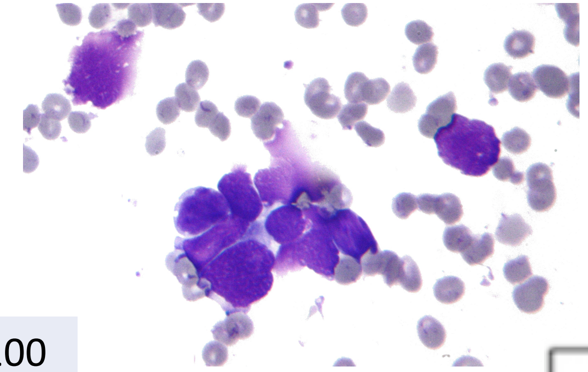
- **Source code** (zip)
- **Source code** (tar.gz)

Extension to segmentation and RNNs coming soon

Dataset Engineering

https://github.com/souryadey/morse-dataset

Data, data, everywhere,
Not quality enough to use



Real world data has challenges:

- ➢ Too few samples
- ➢ Incorrect labeling
- ➢ Missing entries

13.2	0.05		1200
10.9		A	
	0.78	B+	1400
11.4			1100

Synthetic data is generated using computer algorithms

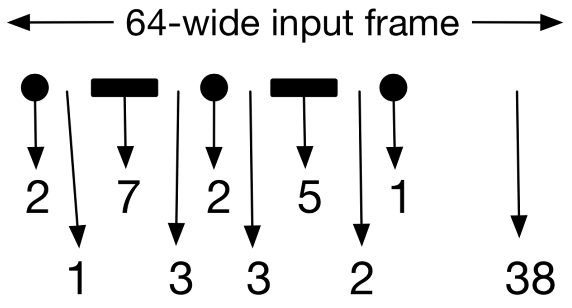
- ➢ Very large quantities can be generated
- ➢ Mimic real-world data as desired
- ➢ Classification difficulty tweaking

Morse Code Datasets

Step 1:

Frame length: 64

Dot: 1-3
Dash: 4-9
Intermediate space: 1-3
Leading spaces: None
Trailing spaces: Remaining at end



Codeword Length = 26. Remaining spaces = 38

Morse Code is a system of communication where letters, numbers and symbols are encoded using dots and dashes

Example:

+ · – · – ·

Morse Code Datasets

Morse Code is a system of communication where letters, numbers and symbols are encoded using dots and dashes

Example:

\+ · – · – ·

Step 1:

Frame length: 64

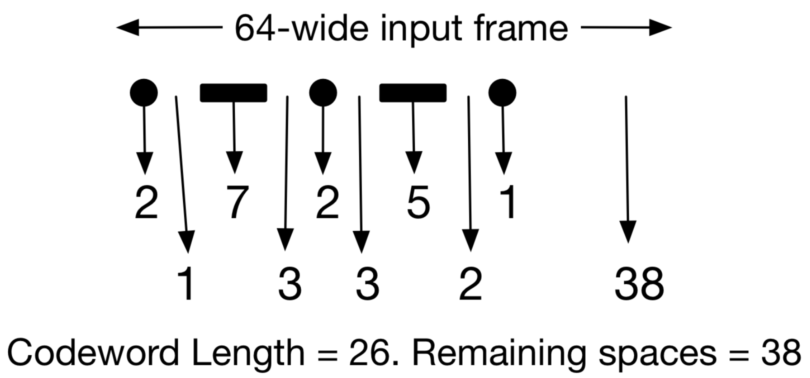
Dot: 1-3
Dash: 4-9
Intermediate space: 1-3
Leading spaces: None
Trailing spaces: Remaining at end

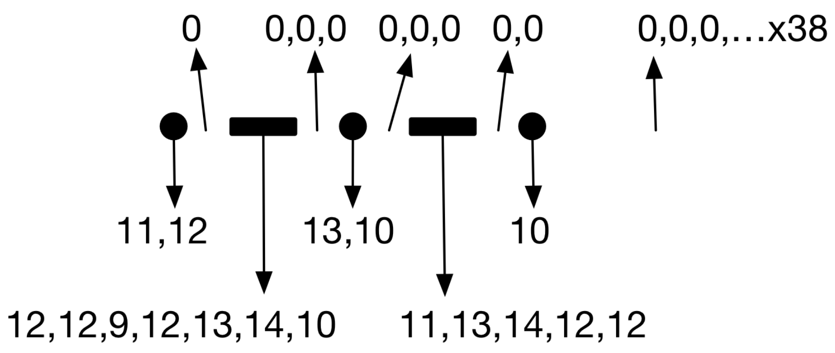
↓

Step 2:

Expected value range = [0,16]

Dot, dash = *Normal*(12,4/3)
Space = 0





Morse Code Datasets

Morse Code is a system of communication where letters, numbers and symbols are encoded using dots and dashes

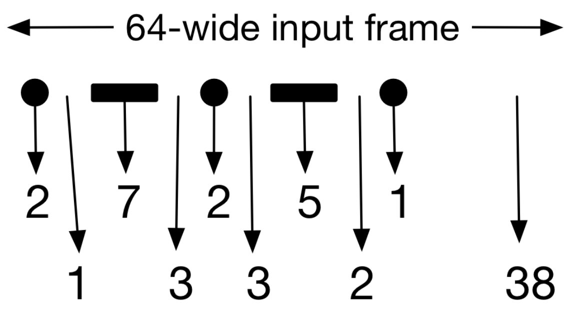
Example:

+ · – · – ·

Step 1:

Frame length: 64

Dot: 1-3
Dash: 4-9
Intermediate space: 1-3
Leading spaces: None
Trailing spaces: Remaining at end

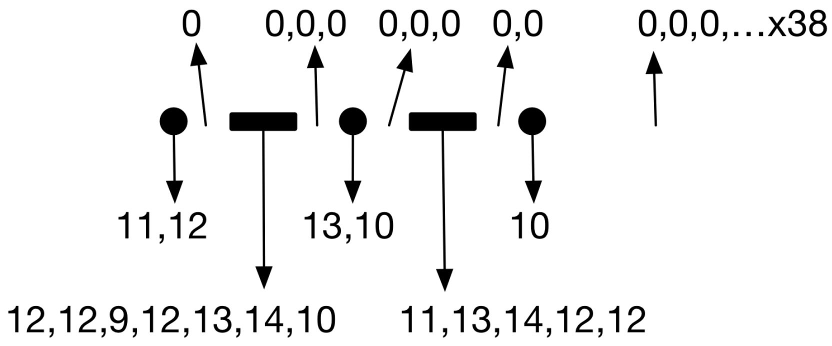


Codeword Length = 26. Remaining spaces = 38

Step 2:

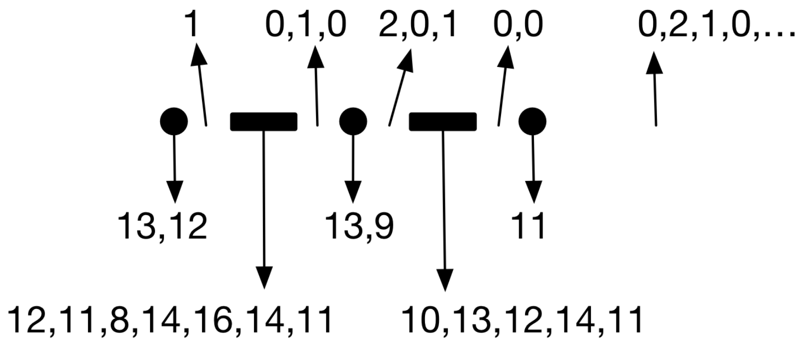
Expected value range = [0,16]

Dot, dash = *Normal*(12,4/3)
Space = 0



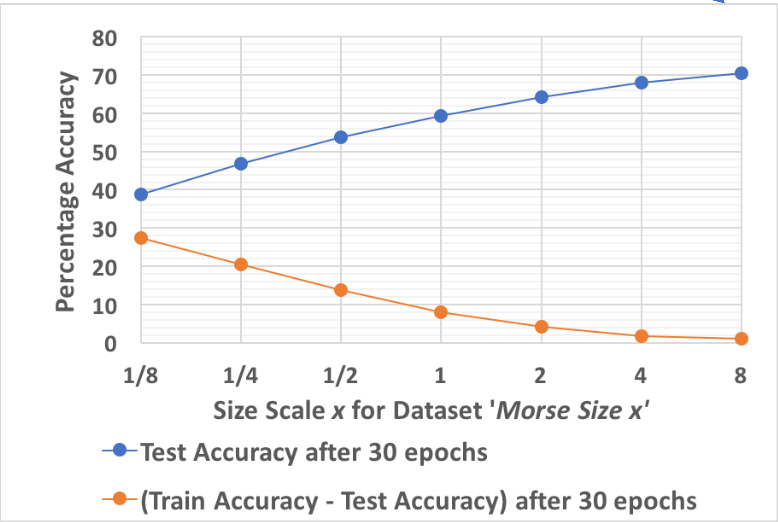
Step 3:

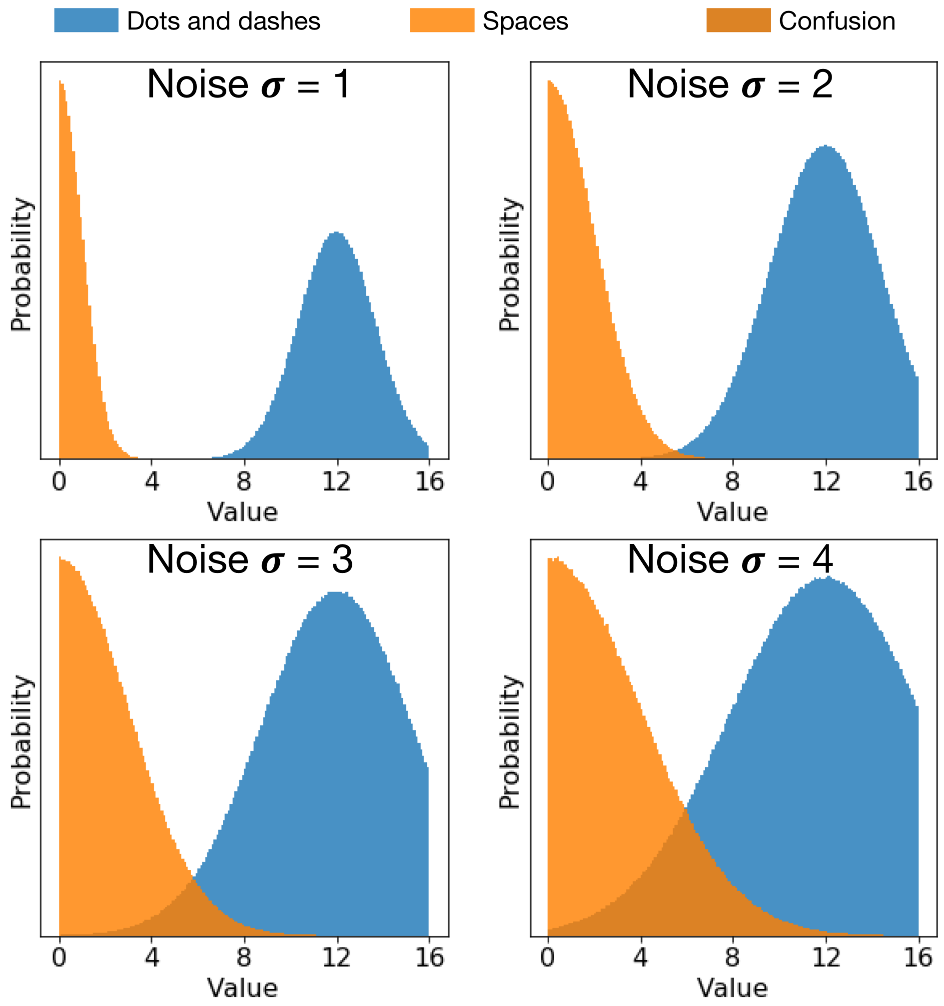
Additive Noise = *Normal*(0,$\boldsymbol{\sigma}$)
(For this case, $\boldsymbol{\sigma}=1$)



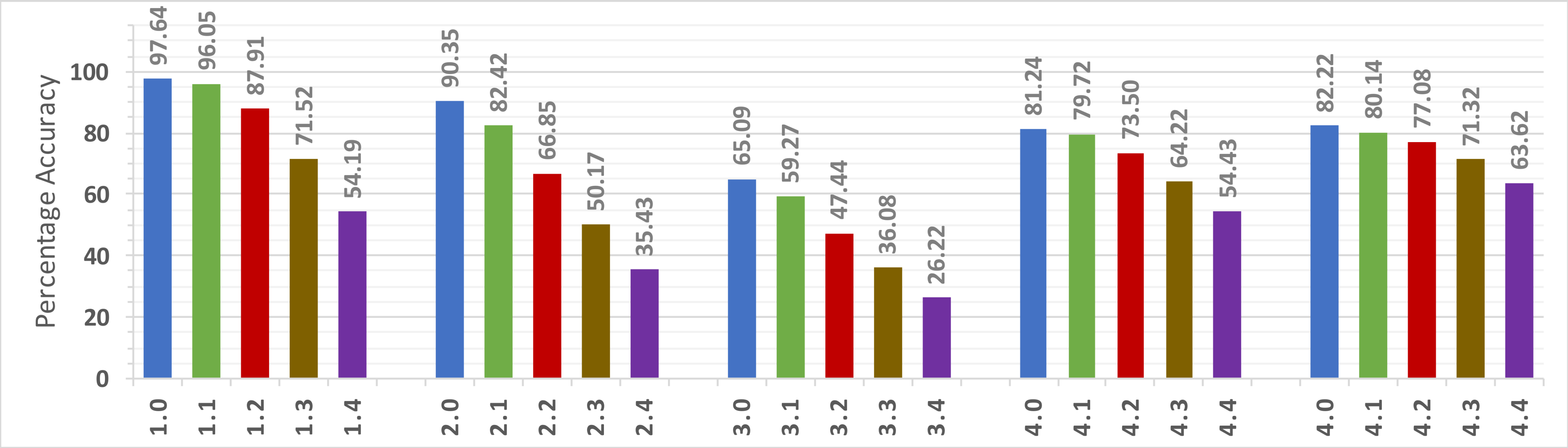
Variations and Difficulty Scaling

- ➢ More noise
- ➢ Leading and trailing spaces
- ➢ Confusing dashes with dots and spaces
- ➢ Dilating frame to size 256
- ➢ Increasing #samples in dataset





Neural network performance (3-layer MLP)



Baseline + Noise

Noise + Leading Spaces

Noise + Leading Spaces + Dash Length=3-9

Dilated by 4 : Noise + Leading Spaces + Dash Length=3-9

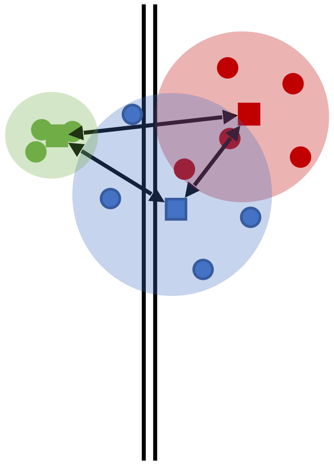
Using L2 reg = 10^-5 Dilated by 4 : Noise + Leading Spaces + Dash Length=3-9

Tunable dataset difficulty leads to a variety of benchmarks

Metrics to characterize dataset difficulty

Probability of the *m*th class occurring

Gaussian Q-function



#classes

$$V_{\text{lower}} = \sum_{m=1}^{N_L} P(m) Q \left(\sqrt{\frac{d_{\min}(m)^2}{4{\sigma_m}^2}} \right)$$

$$V_{\text{upper}} = \sum_{m=1}^{N_L} P(m) \sum_{\substack{j=1 \\ j \neq m}}^{N_L} Q \left(\sqrt{\frac{d(m,j)^2}{4{\sigma_m}^2}} \right)$$

$$V_{\text{dist}} = \frac{\sum_{m=1}^{N_L} \frac{\sigma_m}{d_{\min}(m)}}{N_L}$$

$$\boxed{V_{\text{thresh}}} = \sum_{m=1}^{N_L} \sum_{\substack{j=1 \\ j \neq m}}^{N_L} \mathbb{I} \left(\frac{\|c_m - c_j\|_1}{N_0} < 0.05 \right)$$

Minimum distance between centroids of *m*th class and any other class

Average variance across all features in *m*th class

Distance between centroids of *m*th and *j*th classes

#features

High V values => More difficult

Goodness of the Metrics

Metric	r
V_{lower}	-0.59
V_{upper}	-0.64
V_{dist}	-0.63
V_{thresh}	-0.64

Pearson's correlation coefficient between metric and test set classification accuracy of Morse code datasets of varying difficulty (negative because metrics indicate difficulty)

Metrics can be used to understand the inherent difficulty of the classification problem on a dataset before applying any learning algorithm

Publications

- **S. Dey**, S. C. Kanala, K. M. Chugg and P. A. Beerel, "Deep-n-Cheap: An Automated Search Framework for Low Complexity Deep Learning", submitted to *ECML-PKDD* 2020. Pre-print: arXiv:2004.00974.
- **S. Dey**, K. Huang, P. A. Beerel and K. M. Chugg, "Pre-Defined Sparse Neural Networks with Hardware Acceleration," in *IEEE JETCAS* 2019.
- **S. Dey**, K. M. Chugg and P. A. Beerel, "Morse Code Datasets for Machine Learning," in *ICCCNT* 2018. **Won Best Paper Award**.
- **S. Dey**, D. Chen, Z. Li, S. Kundu, K. Huang, K. M. Chugg and P. A. Beerel, "A Highly Parallel FPGA Implementation of Sparse Neural Network Training," in *ReConFig 2018*.
- **S. Dey**, K. Huang, P. A. Beerel and K. M. Chugg, "Characterizing sparse connectivity patterns in neural networks," in *ITA* 2018.
- **S. Dey**, P. A. Beerel and K. M. Chugg, "Interleaver design for deep neural networks," in *ACSSC* 2017.
- **S. Dey**, Y. Shao, K. M. Chugg and P. A. Beerel, "Accelerating training of deep neural networks via sparse edge processing," in *ICANN* 2017.

People I am thankful to…

Peter Beerel
Professor

Keith Chugg
Professor

Leana Golubchik
Professor

Kuan-Wen Huang
PhD Student

Yinan Shao
Former MS Student

Diandian Chen
Former MS Student

Souvik Kundu
PhD Student

Saikrishna C. Kanala
MS Student

… and many others!

Thank you!

https://souryadey.github.io/