Deep-n-Cheap

Sourya Dey

USC HAL research group

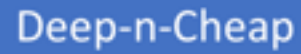
Deep Learning guest lecture

April 6th, 2020

Github

arXiv

Outline



Overview

Motivation

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

    Architecture Hyperparameters
  - Learning rate
  - Batch size

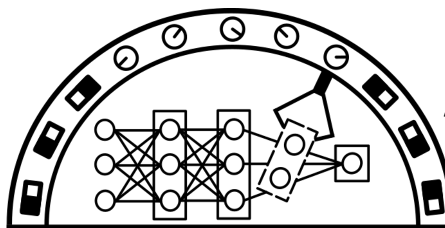
    Training Hyperparameters
  - and more…
- Our understanding of NNs is at best vague, at worst, zero!
- NNs take **a lot** of time to train. Time = Money!

AutoML (Automated Machine Learning)

- Software frameworks that play the role of the designer
- Given a problem, **search** for NN models

Jin 2019 – Auto-Keras

AWsLabs 2020 – AutoGluon



AutoML.org
Freiburg-Hannover

Mendoza 2018 – Auto-PyTorch

Our Work

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*

Approach

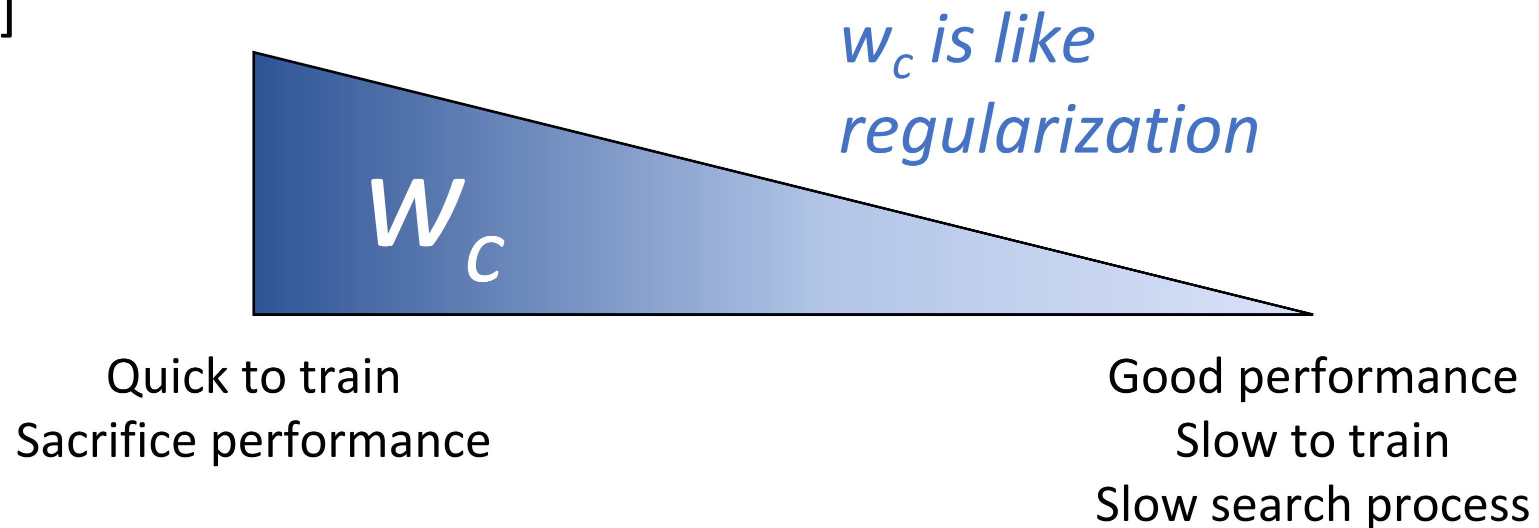
Search Objective

Optimize performance and complexity

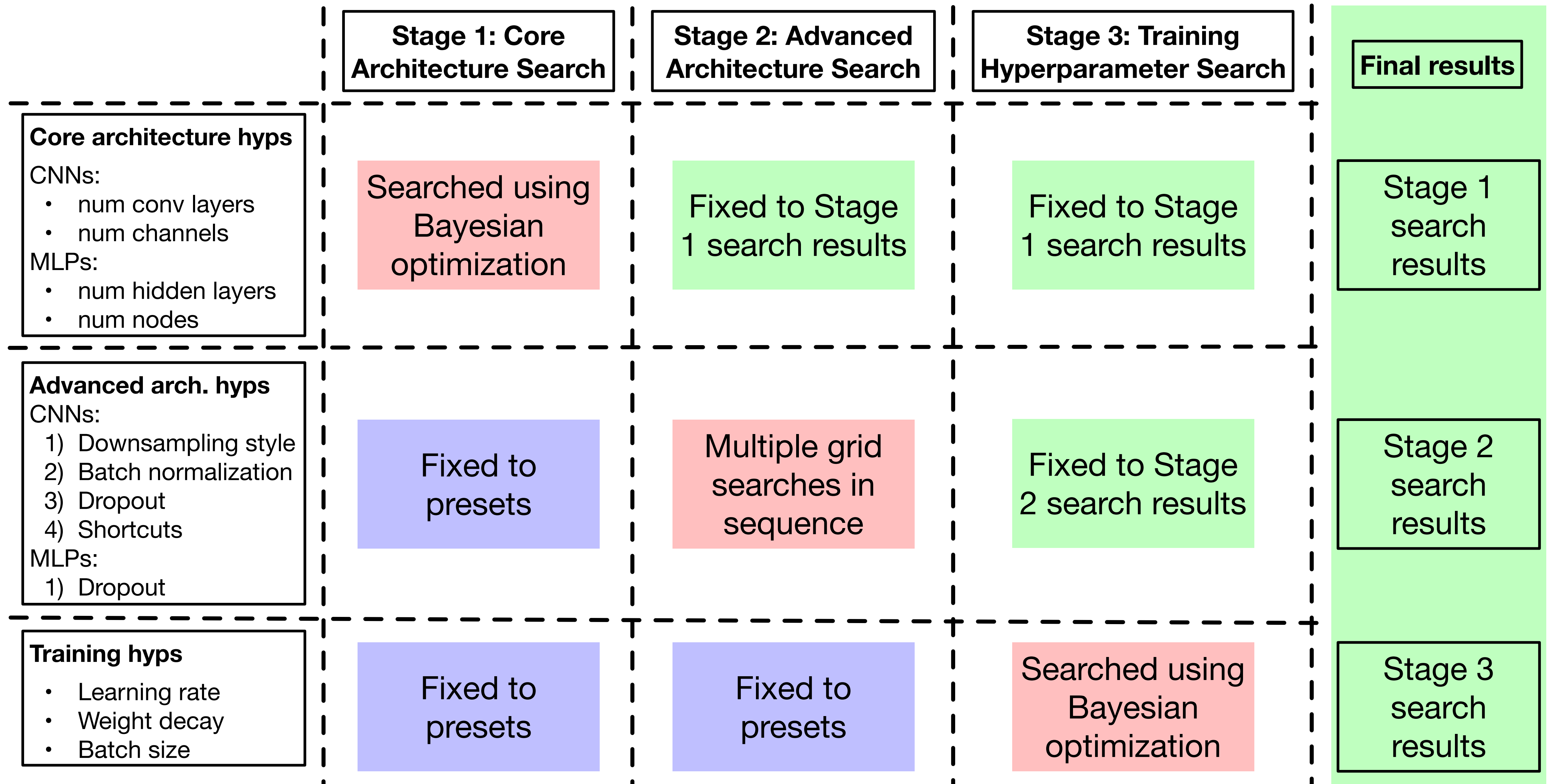
Modified loss function: $f(\text{ NN Config } \mathbf{x}\ ) = \log(\ f_p + w_c * f_c\ )$

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)



Three-stage search process



Examples of Stage 2

Input
Conv
Conv+BN
Conv
Conv+BN
Conv
Conv+BN
Conv
Conv+BN
Conv
Conv+BN
Conv
Conv+BN
Softmax

BN = 0.5

Full shortcuts (left)
Half shortcuts (right)

Input
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Softmax

Input
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Conv
Softmax

Bayesian Optimization Workflow

- *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 $\mathbf{x}_{n1+1} = \text{argmax } EI(\mathbf{x'})$
  - 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

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

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

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

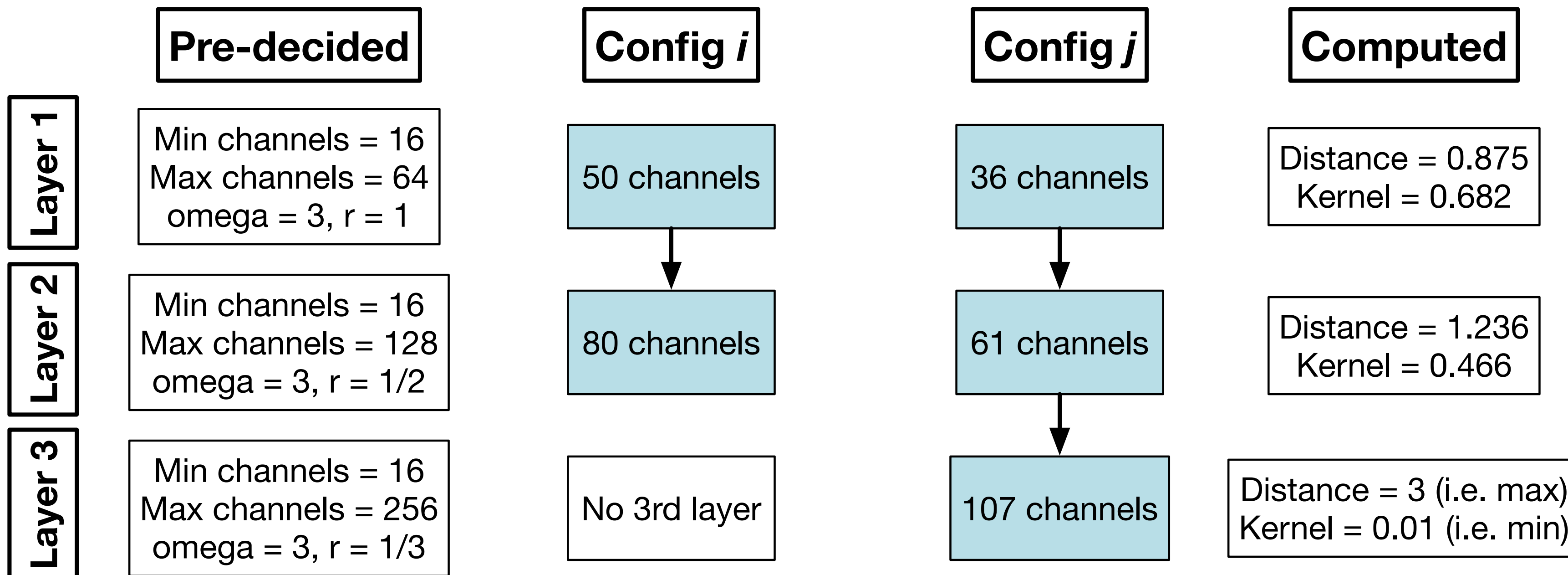
Covariance kernel – Similarity between NN configs

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

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

Results

Data loader and augmentation considerations

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

Using Pytorch data loaders
Uses generators to not burden 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]
```

```
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
```

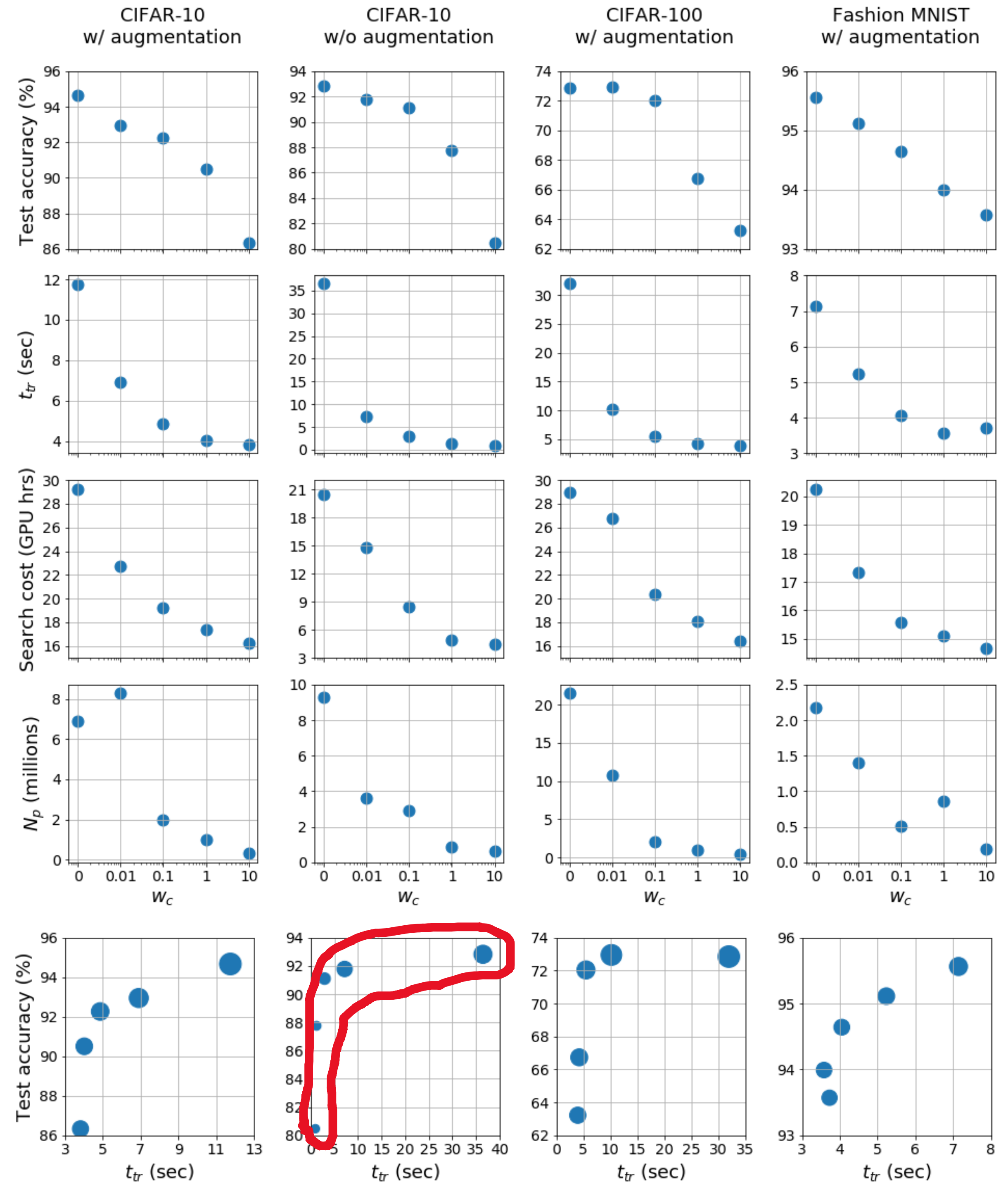
npz is faster, data loaders are more versatile

CNN Results

Complexity Penalty = Training time / epoch

We are not penalizing this, but it's correlated

Performance-complexity tradeoff



CIFAR-10 w/ aug

Conv 50
BatchNorm
Conv 52
BatchNorm
Dropout 0.3
Conv 53
BatchNorm
Conv 59, /2
BatchNorm
Dropout 0.3
Conv 95
BatchNorm
Conv 96
BatchNorm
Dropout 0.3
Conv 97
BatchNorm
Conv 120
MaxPool
BatchNorm
Dropout 0.3
Conv 193
BatchNorm
Conv 239
MaxPool
BatchNorm
Dropout 0.3
Conv 351
BatchNorm
Conv 385
BatchNorm
Dropout 0.3
Conv 488
BatchNorm
Conv 496
BatchNorm
Dropout 0.3
G. A. Pool
Softmax

Conv 39
Conv 60
MaxPool
BatchNorm
Conv 120
BatchNorm
Conv 124
MaxPool
BatchNorm
Dropout 0.3
Conv 178
Conv 192
MaxPool
BatchNorm
Conv 292
BatchNorm
Conv 328
BatchNorm
Dropout 0.3
Conv 352
Conv 396
BatchNorm
Conv 488
BatchNorm
Conv 488
BatchNorm
Dropout 0.3
G. A. Pool
Softmax

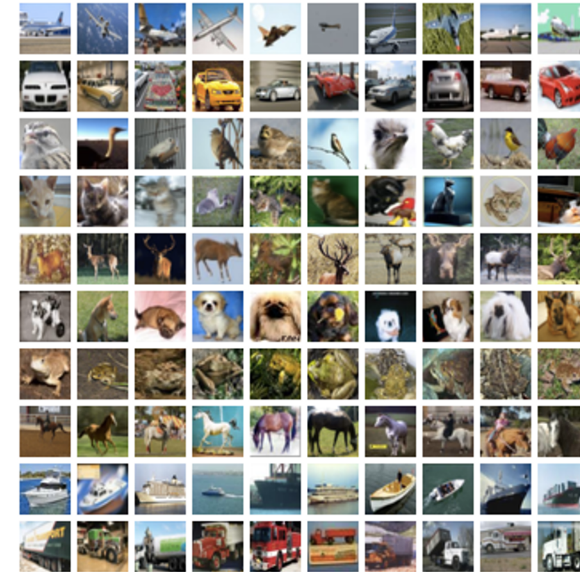
Conv 46
BatchNorm
Conv 51
MaxPool
BatchNorm
Dropout 0.3
Conv 70
BatchNorm
Conv 114
BatchNorm
Dropout 0.3
Conv 128, /2
BatchNorm
Conv 208
MaxPool
BatchNorm
Dropout 0.3
Conv 286
BatchNorm
Conv 371
BatchNorm
Dropout 0.3
G. A. Pool
Softmax

Conv 61
MaxPool
Conv 116
MaxPool
BatchNorm
Dropout 0.15
Conv 216
MaxPool
BatchNorm
Dropout 0.15
Conv 358
BatchNorm
Dropout 0.15
G. A. Pool
Softmax

Conv 37
MaxPool
Conv 67
MaxPool
Dropout 0.15
Conv 130
Conv 170
Dropout 0.15
G. A. Pool
Softmax

Input at top,
output at bottom

0
0.01
0.1
1
10
w_c



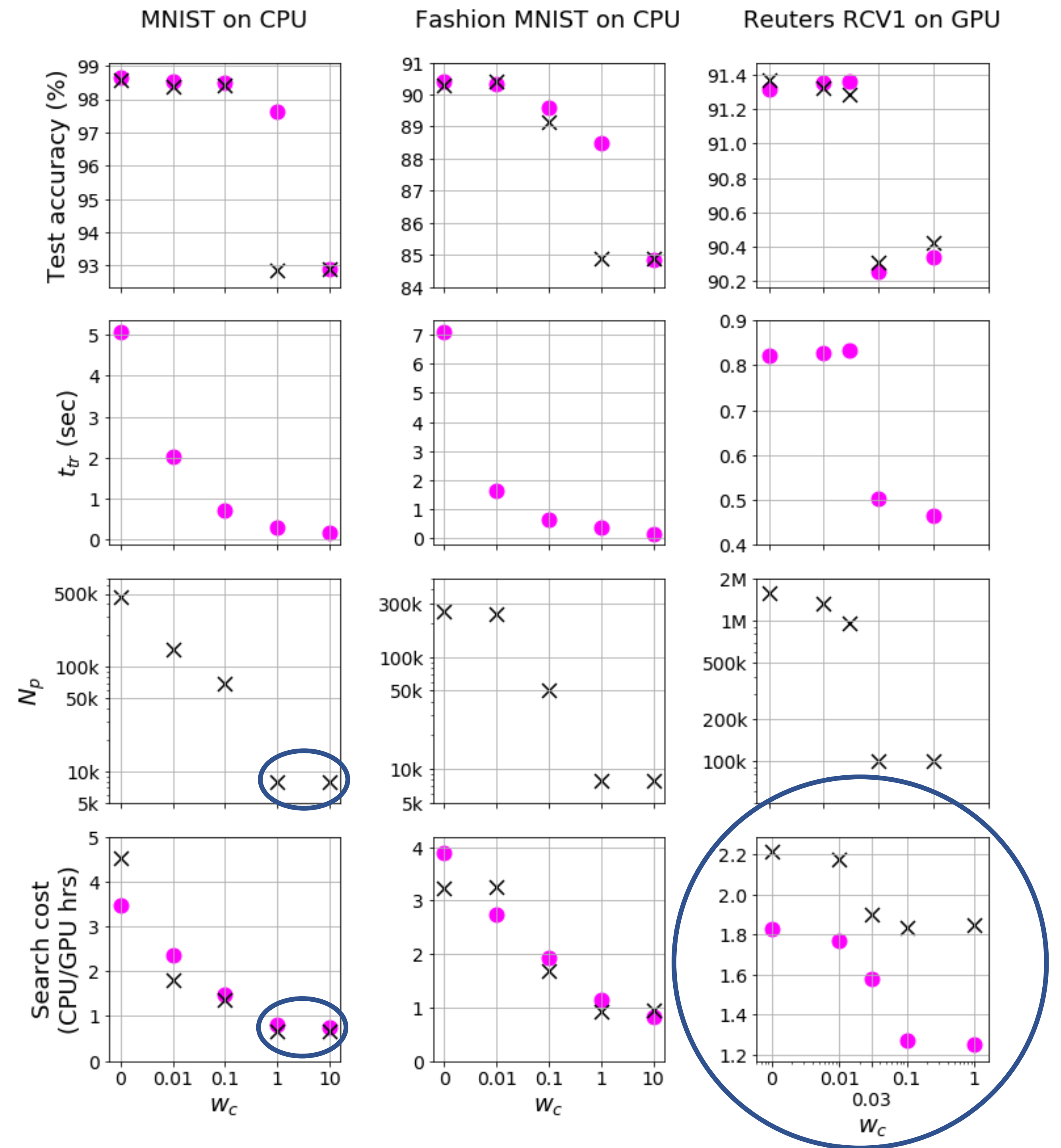
w_c	0	0.01	0.1	1	10
Initial learning rate η	0.001	0.001	0.001	0.003	0.001
Weight decay λ	3.3×10^{-5}	8.3×10^{-5}	1.2×10^{-5}	0	0
Batch size	120	256	459	452	256

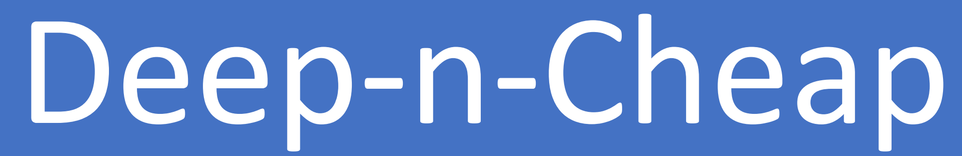
λ strictly correlated with N_p

MLP Results

Pink dots:
Complexity Penalty =
Training time / epoch

Black crosses:
Complexity Penalty =
 # Trainable Params




Deep-n-Cheap

https://github.com/souryadey/deep-n-cheap/blob/master/README.md

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

Penalize inference complexity, not training

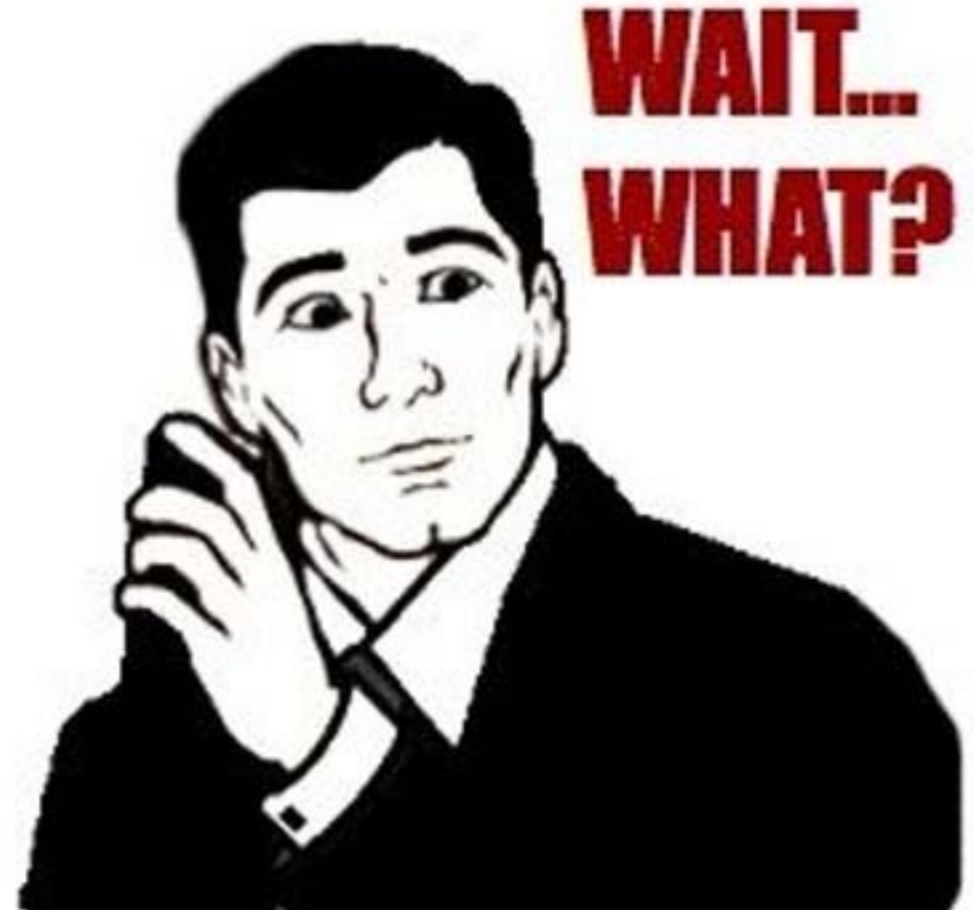
Auto Keras and Gluon don't support getting final model out, so we compared on best val acc found during search instead of final test acc

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!



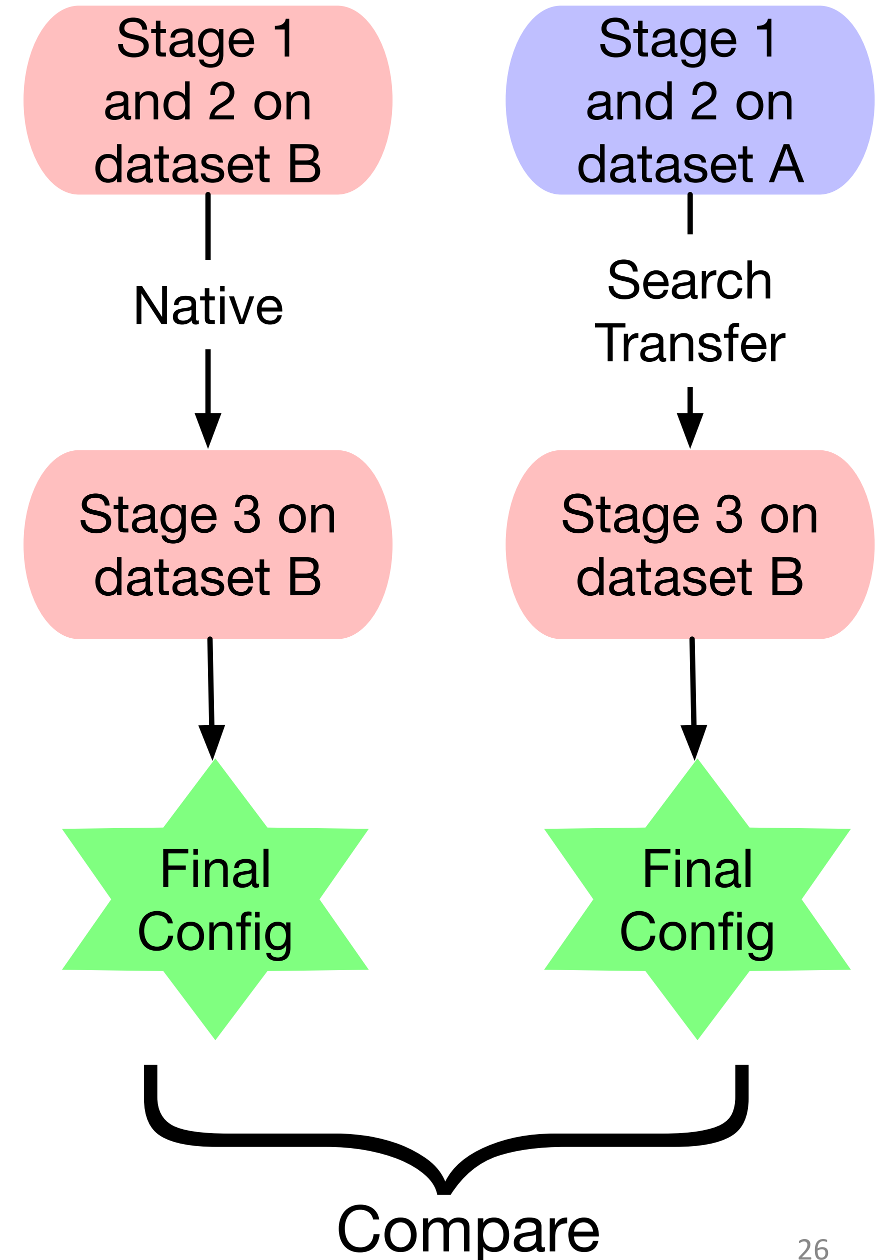
Investigations
and Insights

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?

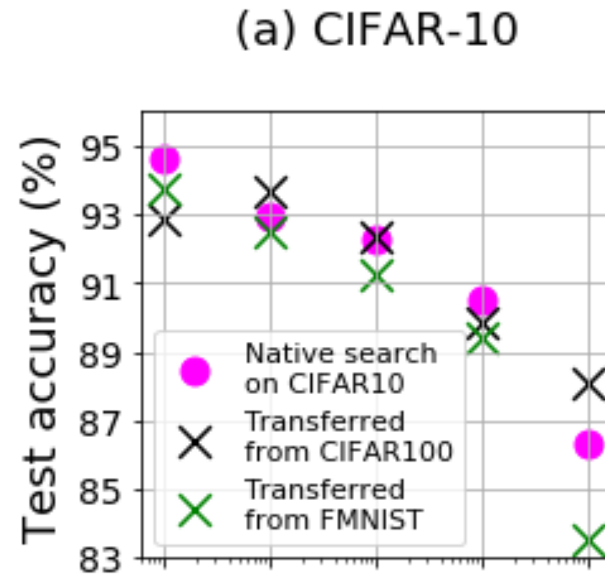
Can architectures generalize?

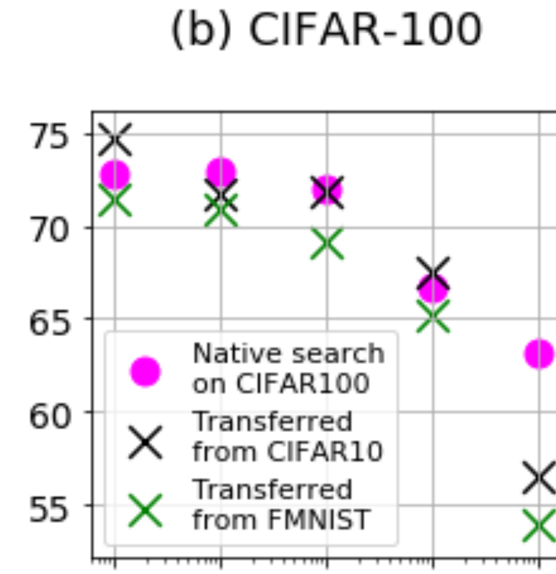


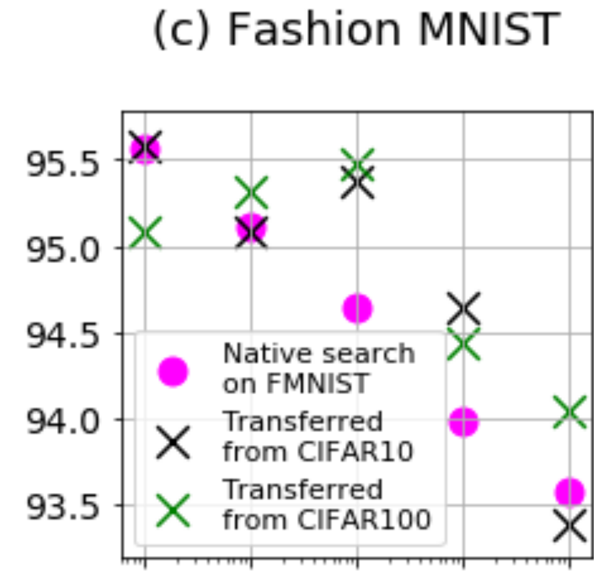
Search transfer results

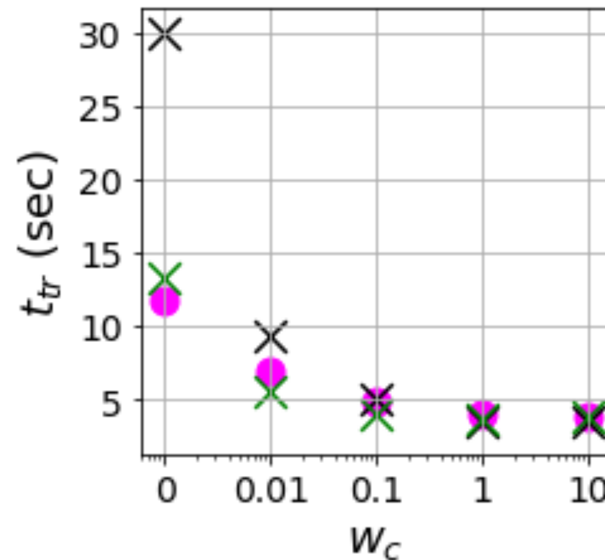
Transferring from CIFAR outperforms native FMNIST (probably due to more params)

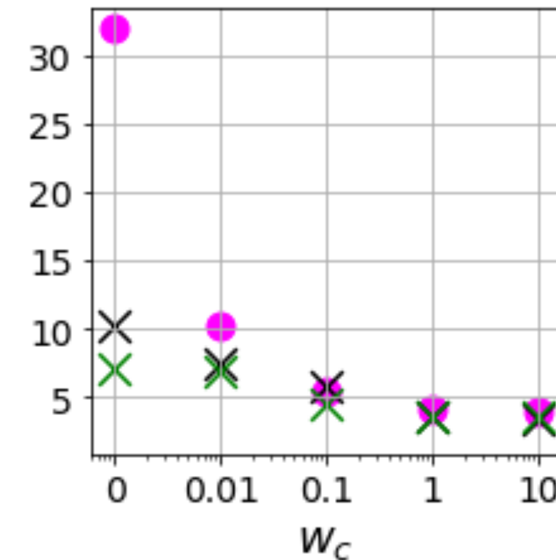
Training times mostly the same

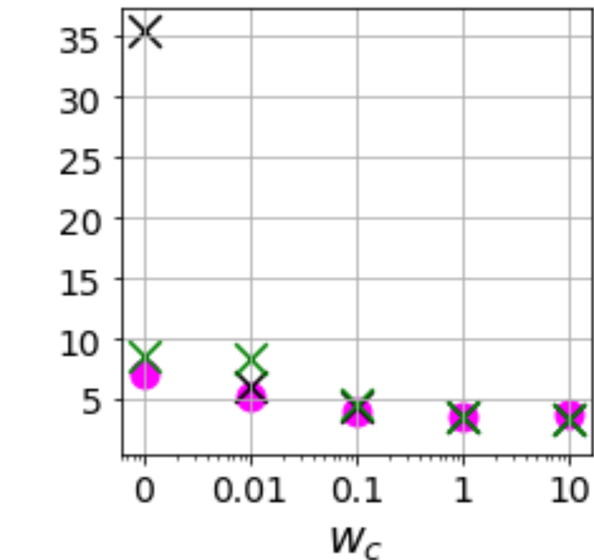




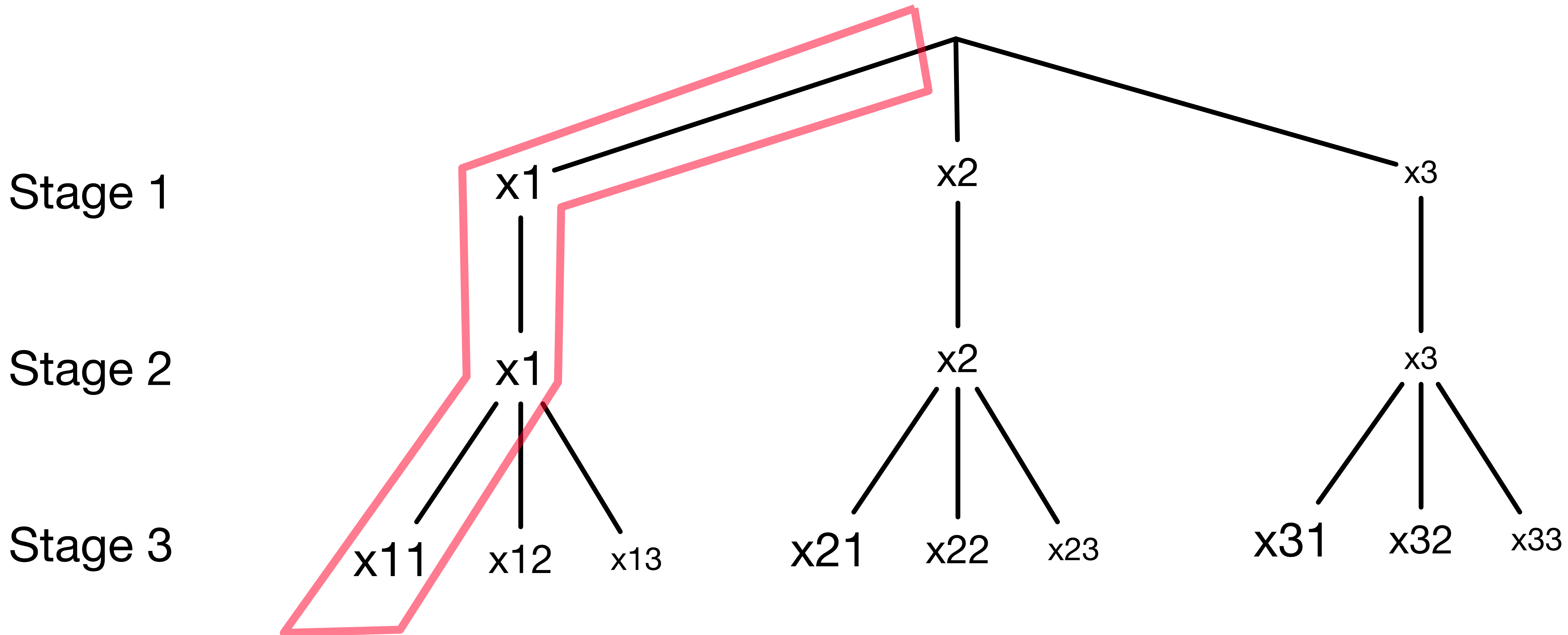




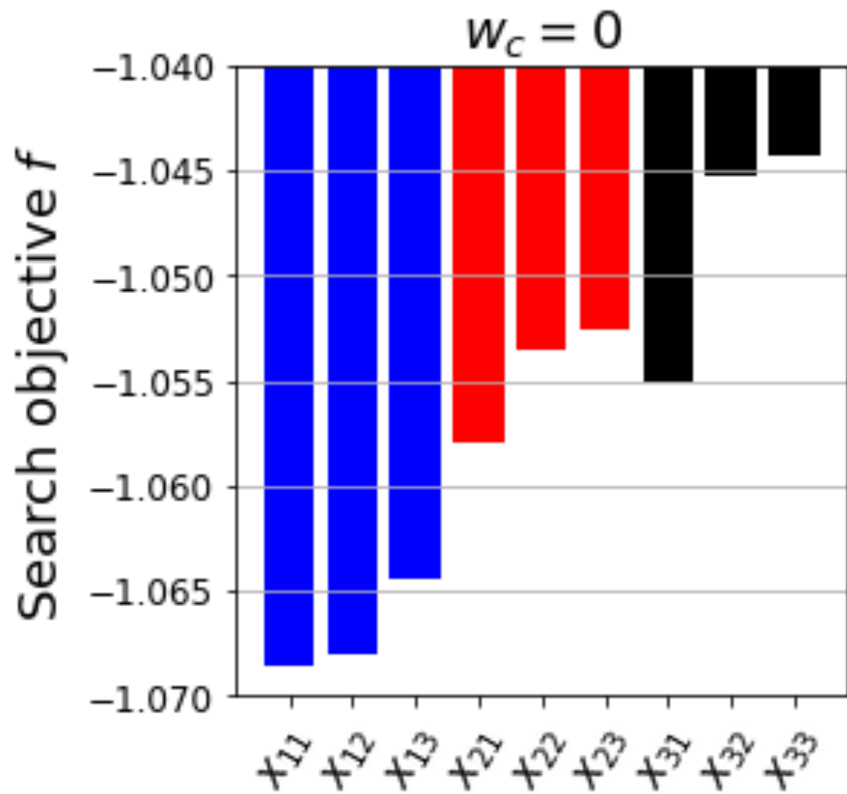


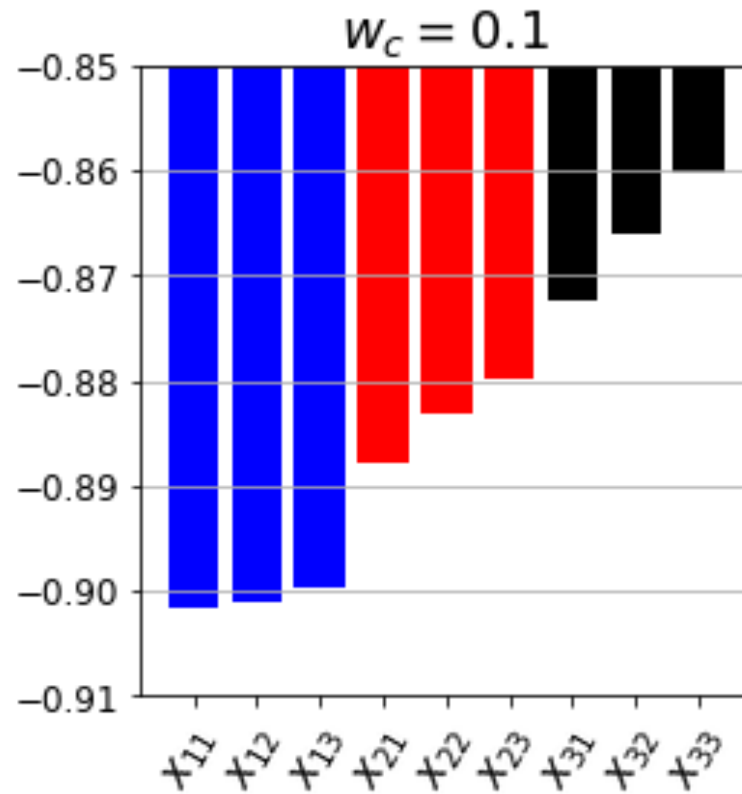


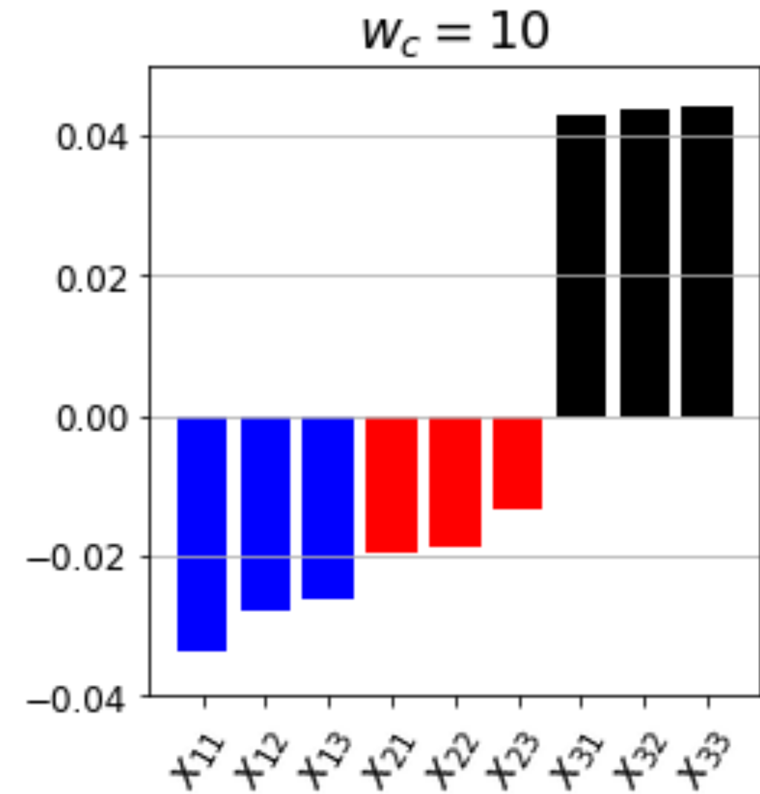
What about a non-greedy search?



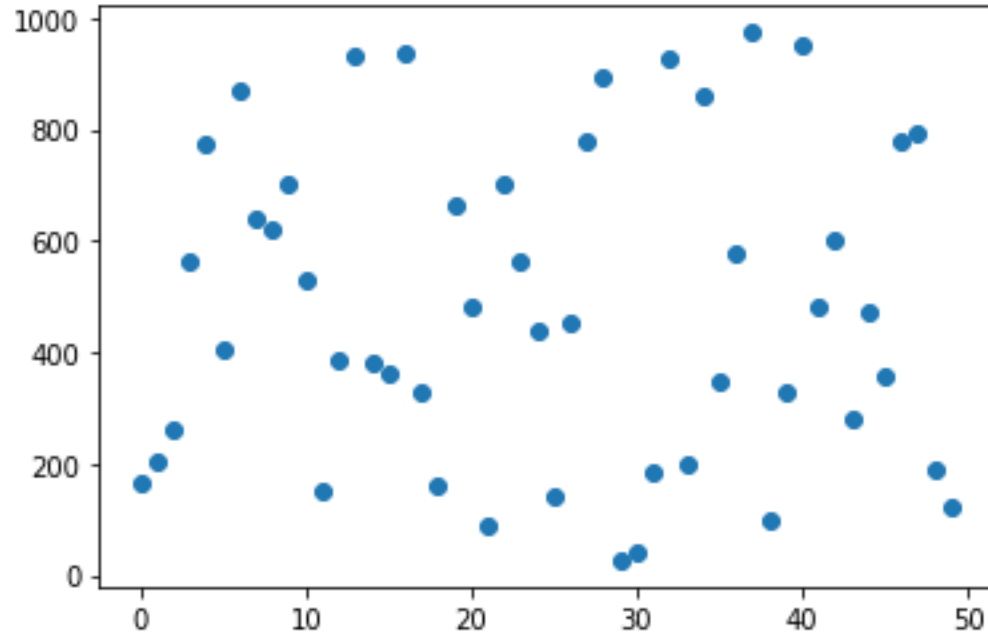
Justifying our greed



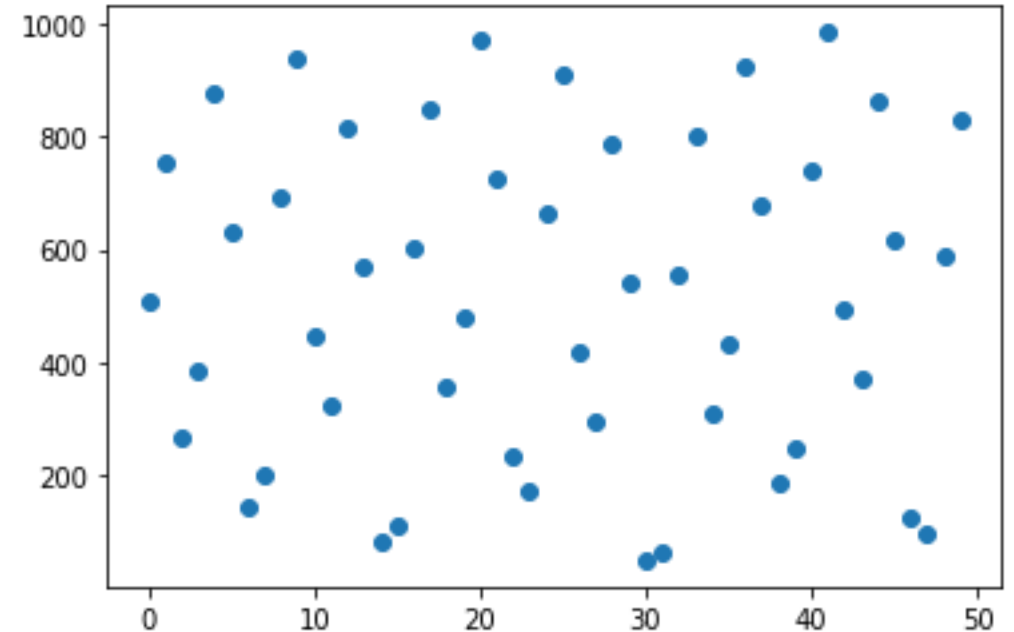




Choosing initial points in Bayesian optimization



Random sampling

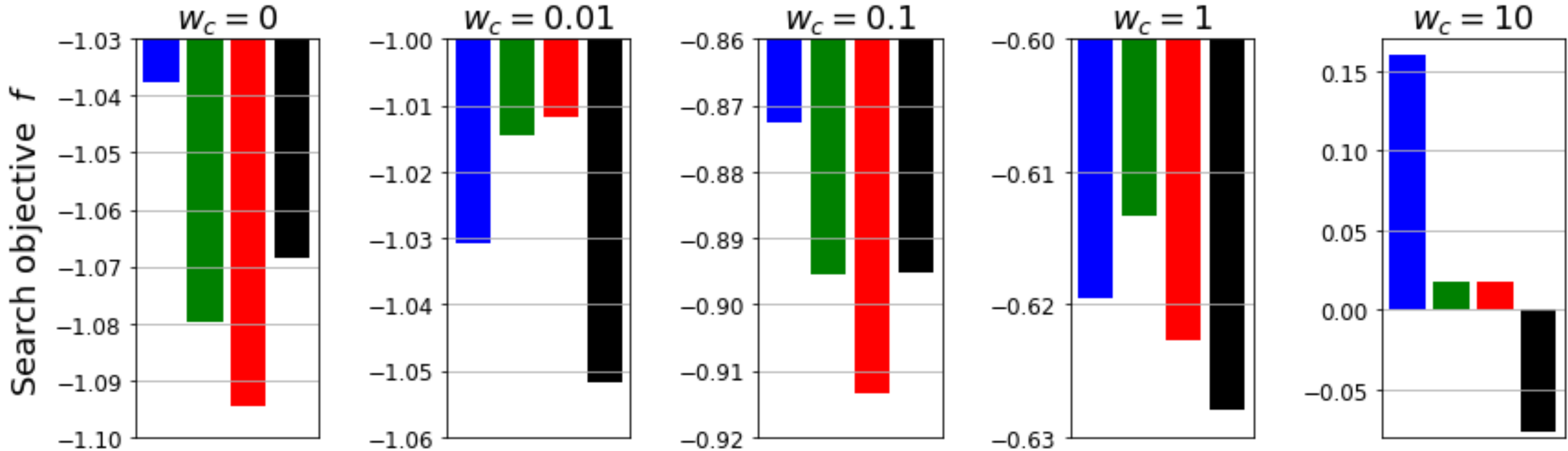


Sobol sampling

Like grid search

Better for more dimensions

BO vs random and grid search (30 points each)



Purely random search: 30 prior

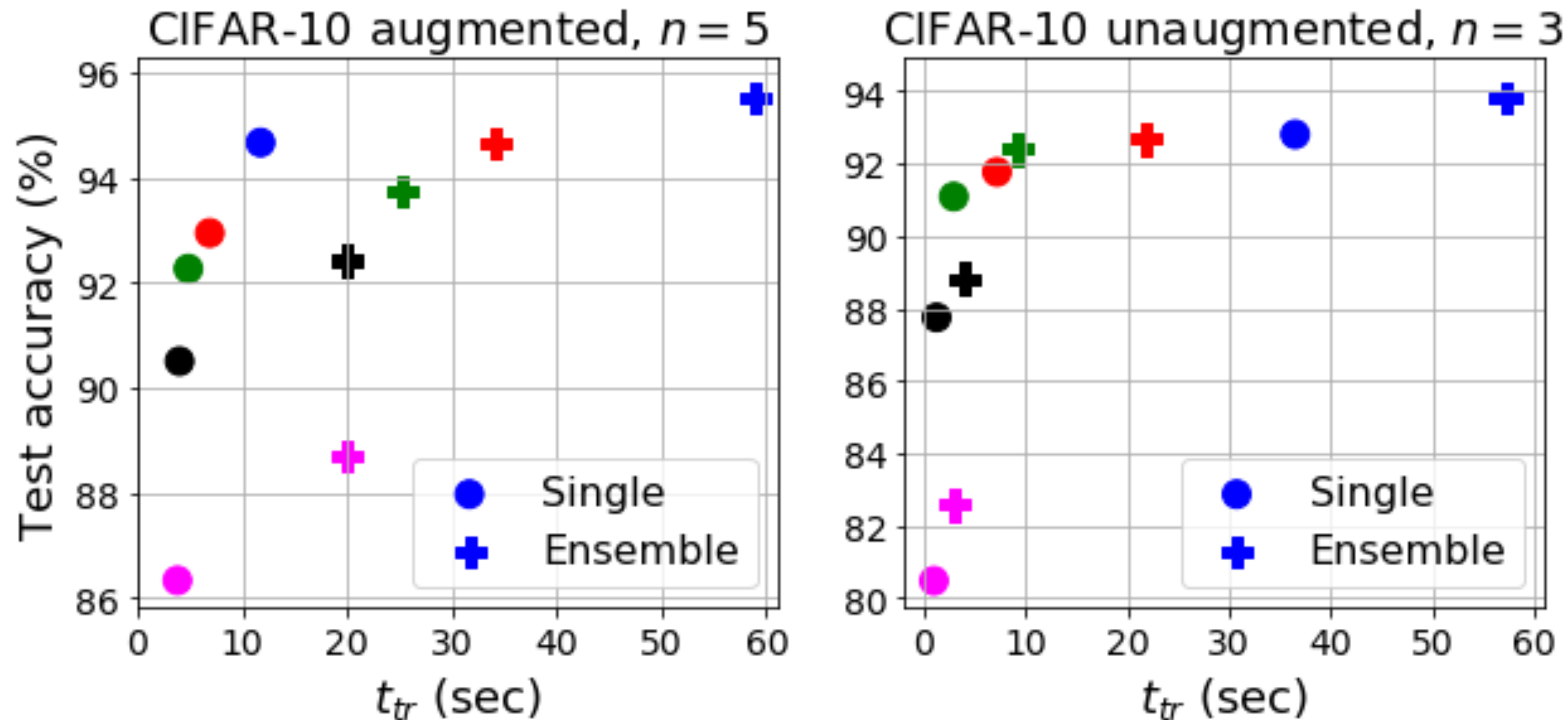
Purely grid search (Sobol): 30 prior

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

Thank you!!

Future work:

- Extension to RNNs
- Extension to more hyperparameters, e.g. kernel sizes for large images
- Tensorflow support