Supplement: HyperSTAR: Task-Aware Hyperparameters for Deep Networks

Gaurav Mittal[⋆†] Chang Liu[⋆‡] Nikolaos Karianakis[†] Victor Fragoso[†] Mei Chen[†] Yun Fu[‡]
[†]Microsoft [‡]Northeastern University
{gaurav.mittal, nikos.karianakis, victor.fragoso, mei.chen}@microsoft.com
liu.chang6@husky.neu.edu yunfu@ece.neu.edu

Here, we begin with describing the training parameters for HyperSTAR followed by details about the hyperparameter configuration space $\mathcal{C}$ and the vector representation of each hyperparameter configuration. We then dive into the meta-data associated with the datasets to be used as feature representation in baseline experiments. Next, we provide an insight into the distribution of hyperparameter configurations over the 10 vision datasets. After this, we provide additional ablation experiments that highlight the effect of different parameters associated with obtaining the task representation for HyperSTAR. Finally, we provide model definition for HyperSTAR and the network architectures used for building the meta-dataset $\mathcal{T}$.

1. Training Parameters

Table 1 lists out the values of the different training parameters used for training HyperSTAR for both SE-ResNeXt-50 [7] and ShuffleNet-v2-x1 [10].

Table 1. Training parameters for our HyperSTAR implementation

Training Parameter	Value
Framework	Pytorch v1.2
Initial Learning Rate	0.001
Learning Rate Scheduler	StepLR
Learning Rate Epoch Step Size	150
Learning Rate Step Decay	0.1
Maximum Epochs	500
Image Batch Size	64
Hyperparameter Batch Size	16
Image Size	256×256
Random Crop Size	224×224
Image Normalize Mean	$[0.482, 0.459, 0.408]$
Image Normalize Std	$[1., 1., 1.]$
Random Scale Range	$(0.8, 1.2)$
Random Rotation Range	$(-10°, 10°)$
Random Translation Range	$X, Y \in (-0.07, 0.07)$

⋆ Authors with equal contribution.
This work was done when C. Liu was a research intern at Microsoft.

2. Hyperparameter Space $\mathcal{C}$

This section gives an overview of the different hyperparameter configurations for which each of the network architectures was trained. Table 2 shows the different hyperparameters used as part of the hyperparameter configuration space explored for SE-ResNeXt-50. Similarly, Table 3 shows for ShuffleNet-v2-x1.

- We operate over a discrete unordered set of hyperparameters with the different hyperparameters taken as a grid (Cartesian product) over all combinations. Therefore, the size of hyperparameter configuration space for SE-ResNeXt-50 and ShuffleNet-v2-x1 is 40 and 108 respectively, as can be observed from Tables 2 and 3.

- For both network architectures, the data augmentation space consists for two augmentation policies.

    1. 'Random Crop' where every time when we process an image as input to the HyperSTAR, the image is resized, normalized and randomly cropped based on parameters given in Table 1.

    2. 'Random Affine' where in addition of doing 'Random Crop', the image further undergoes an affine transformation with randomly chosen rotation, scaling and translation factors as given in Table 1.

- As the second hyperparameter dimension, we choose from a set of training strategies. Each training strategy comprises of the choice of optimizer, Stochastic Gradient Descent (SGD) [11] and Adam [9], to use for optimizing the multi-class cross entropy loss for the network architecture. The training strategy further comprises of a learning rate multiplier for the `classifier(fc)` layer of the network architecture to incorporate adaptive learning rate [8] for improved performance. For instance, in Tables 2 and 3, $(SGD, 10)$ refers to training with SGD optimizer

Table 2. Hyperparameters for SE-ResNeXt-50. $H = 2 \times 4 \times 5 = 40$

Hyperparameter	Range
Data Augmentation	{Random Crop, Random Affine (Crop+Scale+Rotate+Translate)}
Optimizer + LR Multiplier	{(SGD, 1), (SGD, 5), (SGD, 10), (Adam, 1)}
Layers upto which to finetune	{classifier, 5_2, 5_1, 4_6, 4_3}

Table 3. Hyperparameters for ShuffleNet-v2-x1. $H = 2 \times 3 \times 6 \times 3 = 108$

Hyperparameter	Range
Data Augmentation	{Random Crop, Random Affine (Crop+Scale+Rotate+Translate)}
Optimizer + LR Multiplier	{(SGD, 1), (SGD, 5), (SGD, 10)}
Layers upto which to finetune	{classifier, 2_1, 1_7, 1_5, 1_3, 1_0}
Base Learning Rate	{0.001, 0.003, 0.01}

using 10 as the learning rate multiplier over the base learning rate for the `classifier` layer.

- For the third hyperparameter dimension, we choose which layers of the network to fine tune. As given in Tables 2 and 3, each value of this hyperparameter suggests the layer name up to where to fine tune the network architecture starting from the classifier. All layers below the specified layer name will be frozen. The position of the layer names can be referenced from the network architecture definition given in Section 7. For instance, layer name 5_2 for SE-ResNeXt-50 suggests to freeze the network below layer 5_2 and train/fine-tune the network from 5_2 to the `classifier` layer.

- For SE-ResNeXt-50, we fixed the base learning rate to train for each hyperparameter configuration to be 0.001. In case of ShuffleNet-v2-x1, we explored three different base learning rates: 0.001, 0.003 and 0.01.

- Every hyperparameter configuration is a combination of the above mentioned hyperparameters. For instance, `sgd_1_0.001_4_6_random_crop` refers to training SE-ResNeXt-50 with SGD optimizer, base learning rate 0.001, learning rate multiplier of 1 and 'Random Crop' as the data augmentation policy with fine-tuning the network upto layer `4_6`.

- The other hyperparameters fixed for training both SE-ResNeXt-50 and ShuffleNet-v2-x1 are batch size of 45 and 100 respectively, number of epochs 100 with LR schedule of step size 25 and step decay 0.1, no weight decay and momentum of 0.9 for SGD.

3. Hyperparameter Configuration Encoding $\mathbf{c}$

We need a vector to represent each hyperparameter configuration that can be fed as input to HyperSTAR. To vectorize each hyperparameter configuration, we encode each hyperparameter as a one-hot encoding and concatenate each of these encodings together to form the final vector $\mathbf{c}$. For instance, $[0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1]$ is a vector representing the hyperparameter configuration `sgd_10_0.003_1_5_random_afffine` for ShuffleNet-v2-x1. The encoding is a 14-dimension one-hot vector where dimension $1-3$ encodes training stategy (`sgd_10`), dimension $4-9$ encodes fine-tune layer (`1_5`), dimension $10-12$ encodes base learning rate (`0.003`) and dimension $12-14$ encodes augmentation policy (`random_affine`). Similarly, we have a 13-dimensional vector encoding each hyperparameter configuration for SE-ResNeXt-50.

4. Meta-data

Although meta-data is not part of our main HyperSTAR algorithm, we describe the details of meta-data features as they are part of some of the baselines we compare with. Table 4 shows the information about the datasets (tasks) $\mathcal{D}$ used as part of the meta-dataset $\mathcal{T}$ over the dataset-configuration space. For some datasets (such as DeepFashion), we took a subset of images to keep the time of training for all the datasets and for all the hyperparameter configurations tractable. The Table also provides information about the metadata features used as part of some of the baseline experiments [4, 3] where the feature representation comprises of metadata relevant to the vision tasks we considered. Based on previous approaches [4, 1], the metadata feature vector is a 6-dimensional feature vector composed of:

1. `number of training images`
2. `number of classes/labels`
3. `average number of images per class`
4. `log(number of training images)`
5. `log(number of classes)` and
6. `log(average number of images per class`

in the given order. Since the range of values of each dimension is very different and to balance the effect of each

Table 4. The 6-dimensional metadata input vector used for each dataset. Each dimension was normalized with mean and standard deviation over all values.

Dataset	num_train_images	num_classes	avg_image_per_class	log(num_train_images)	log(num_classes)	log(image_per_class)
MIT Indoor	5350	67	79.85	8.58	4.20	4.38
Oxford-IIIT Pets	3680	37	99.46	8.21	3.61	4.60
Caltech256	22955	257	89.32	10.04	5.55	4.49
Food101	3030	101	30.00	8.02	4.62	3.40
Places365	10950	365	30.00	9.30	5.90	3.40
DeepFashion	2300	46	50.00	7.74	3.83	3.91
BookCover30	5700	30	190.00	8.65	3.40	5.25
IP102 (Pests)	4080	102	40.00	8.31	4.62	3.69
Sun397	20549	397	51.76	9.93	5.98	3.95
Textures (DTD)	3760	47	80.00	8.23	3.85	4.38

metadata feature dimension, we normalize each dimension using the mean and standard deviation over all the values of that given dimension. That brings each dimension to follow a zero-mean with unit standard deviation normal distribution.

5. Hyperparameter Distribution

As we mentioned in the main paper, the success of transfer learning for visual classification tasks [13, 2] requires spending countless hours by Machine Learning (ML) experts to find the optimal hyperparameters which can achieve the best possible performance for the new dataset (task) for a given network architecture. One of the reasons for spending this large amount of time is the fact that same set of hyperparameters are not optimal across all datasets. Given the visual nature of the target dataset, the hyperparameter configurations can differ significantly in their relative ranking in the hyperparameter configuration space based on the achieved performance (Top-1 accuracy). Figure 1 and 2 show this diversity in the ranking of the different hyperparameter configurations in the form of a box-whisker plot for SE-ResNeXt-50 and ShuffleNet-v2-x1 respectively. In each plot, higher rank denotes the corresponding configuration achieving a better Top-1 accuracy compared to other configurations with lower ranks. Rank 1 denotes the configuration with least Top-1 accuracy and Rank 40 (for SE-ResNeXt-50) or 108 (for ShuffleNet-v2-x1) denotes the configuration with the highest Top-1 accuracy for a dataset. The y-axis spans over this ranking while the x-axis spans over the different hyperparameter configurations (40 for SE-ResNeXt-50 and 108 for ShuffleNet-v2-x1). The configurations are ordered from left-to-right in the order of decreasing mean rank over the 10 datasets. We can observe from both plots that a large number of hyperparameter configurations shows a significant variation in ranking across the 10 real-world image classification datasets. For instance, for many configuration for ShuffleNet-v2-x1, the ranking can be as high as 108 (best performing configuration) for one dataset and can be as low as 10 (really poor performing configuration) for another dataset at the same time. It seems that the variation in performance for any given hyperparameter configuration is much larger in case of ShuffleNet-v2-x1 compared to SE-ResNeXt-50. This could be due to the shallower network architecture for ShuffleNet-v2-x1 that leads to dramatic change in performance with different visual datasets.

The two plots confirm that there is no one hyperparameter configuration that performs consistently the best over the diverse set of visual classification datasets chosen for evaluation. Therefore, with every new dataset, there is a need to search the whole space of hyperparameter configurations to find the best performing configuration. HyperSTAR significantly cuts down this search time by recommending optimal configurations in a task-aware manner based on visually grounded task representations. By learning over the joint meta dataset-configuration space, it is able to effectively predict and rank hyperparameters in an end-to-end manner directly from raw images, and save hours of time and effort required for hyperparameter search.

6. Ablation Studies

In this section, we show a more thorough ablation study over the different parameters associated with learning the task representation $\mathbf{t}$ in HyperSTAR and applying the different regularizations. We perform the experiments on SE-ResNeXt-50 and summarize the results in Table 5. We would like to point out that each experiment in the main paper and here in the supplementary material is performed 10 times and we report the AP@10 for each experiment averaged over these 10 trials.

Batchwise Mean - Batch Size. We tried experimenting with higher batch size for computing the batchwise mean (BM) for the task representation. As seen in Table 5, we did not see any performance improvement on doubling the batchsize from 64 to 128. Therefore, we fixed the batch size to 64.

Adversarial regularization vs L1 regularization. As described in main text, in order to reduce intra-task representation variance, we impose an adversarial based loss

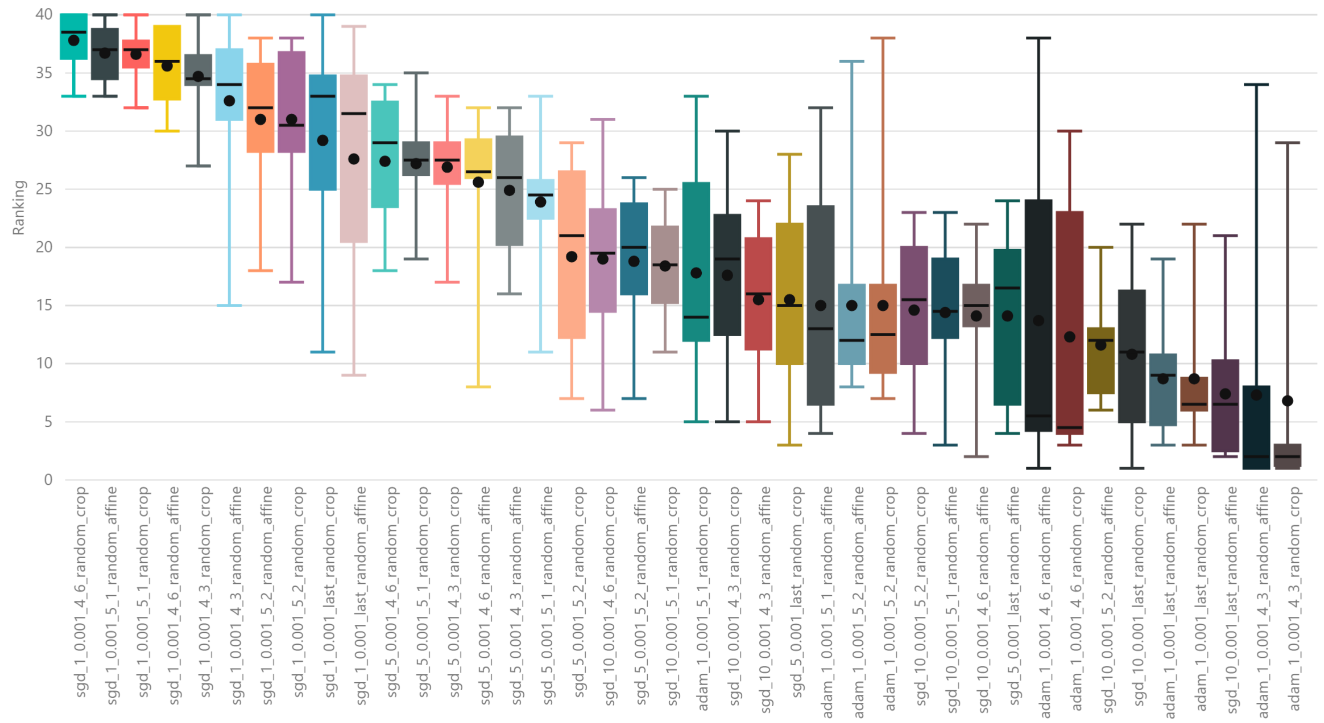

Figure 1. Ranking distribution over the hyperparameter configuration over 10 datasets for SE-ResNeXt-50 (higher rank is better)

Table 5. AP@10 comparison for SE-ResNeXt-50 for 10 public image classification datasets across different ablation experiments.

Test Dataset	BookCover30	Caltech256	DeepFashion	Food101	MIT Indoor	IP102 (Pests)	Oxford-IIIT Pets	Places365	SUN397	Textures (DTD)	Average
64 Batchwise Mean (BM)	68.16	82.34	62.39	**70.98**	72.51	84.94	**81.43**	88.05	93.74	**82.59**	78.71
128 Batchwise Mean (BM)	62.53	78.82	79.06	68.52	74.49	79.75	81.49	90.14	94.98	79.31	78.91
BM + GAN	64.02	83.83	87.63	67.27	76.45	87.49	78.42	**93.41**	92.92	77.21	80.87
BM + L1 loss	54.00	80.07	88.84	63.90	76.27	82.76	80.42	95.60	93.69	76.55	79.21
BM + GAN(L=10)	64.02	83.83	87.63	67.27	76.45	87.49	78.42	**93.41**	92.92	77.21	80.87
BM + GAN(L=100)	56.76	81.82	77.60	68.71	76.06	81.67	81.57	92.26	95.47	77.48	78.94
BM + GAN(L=global)	61.66	80.65	73.94	69.57	76.77	84.36	79.29	94.51	96.20	75.98	79.29
BM + Single Similarity	62.60	80.97	82.39	67.31	78.79	83.64	79.52	90.37	**94.47**	81.63	80.17
BM + Double Similarity	66.80	87.88	92.09	67.36	79.35	85.72	77.53	93.74	93.73	77.91	82.21
BM + Double Similarity + GAN	**68.27**	**86.72**	**91.51**	68.20	77.97	87.52	79.64	91.72	91.85	81.46	**82.49**

with the goal of keeping the dataset representation computed from batches $\mathbf{t}_i^l$ close to the global representation of the dataset $\mathbf{t}_i^G$. We took inspiration from domain adaptation techniques to introduce this loss [5, 12, 6]. As a common ablation for this loss, we compare the performance of HyperSTAR on using this adversarial loss versus a naïve L1 based regularization between batch-wise task representation $\mathbf{t}_i^l$ and global dataset representation $\mathbf{t}_i^G$. From the Table 5, we can observe that the average AP@10 of introducing adversarial based regularization (BM + GAN) is significantly better compared to introducing L1 regularization (BM + L1 loss). This suggests that the adversarial loss reduces the intra-task representation variance leading to improved recommendations by HyperSTAR.

Window Size L for computing global dataset representation $\mathbf{t}_i^G$. We further experiment with different window sizes L for computing the global dataset representation $\mathbf{t}_i^G$ which is used to impose the adversarial loss to reduce the intra-task representation variance. Table 5 shows three different window sizes we experiment with, $L = 10$, $L = 100$ and $L =$ global (where we compute mean over all batches of a dataset processed since the beginning of training HyperSTAR). Among the three cases, we observe that with $L = 10$, we obtain the best $AP@10$. We believe that longer horizon/window size over the processed image batches can lead to information loss due to smoothening out of the feature representation, thus impacting the ability of the discriminator to differentiate between resulting aggregated dataset representation and batchwise representation. This, in turn, affects the effectiveness of the adversarial regularization lead-

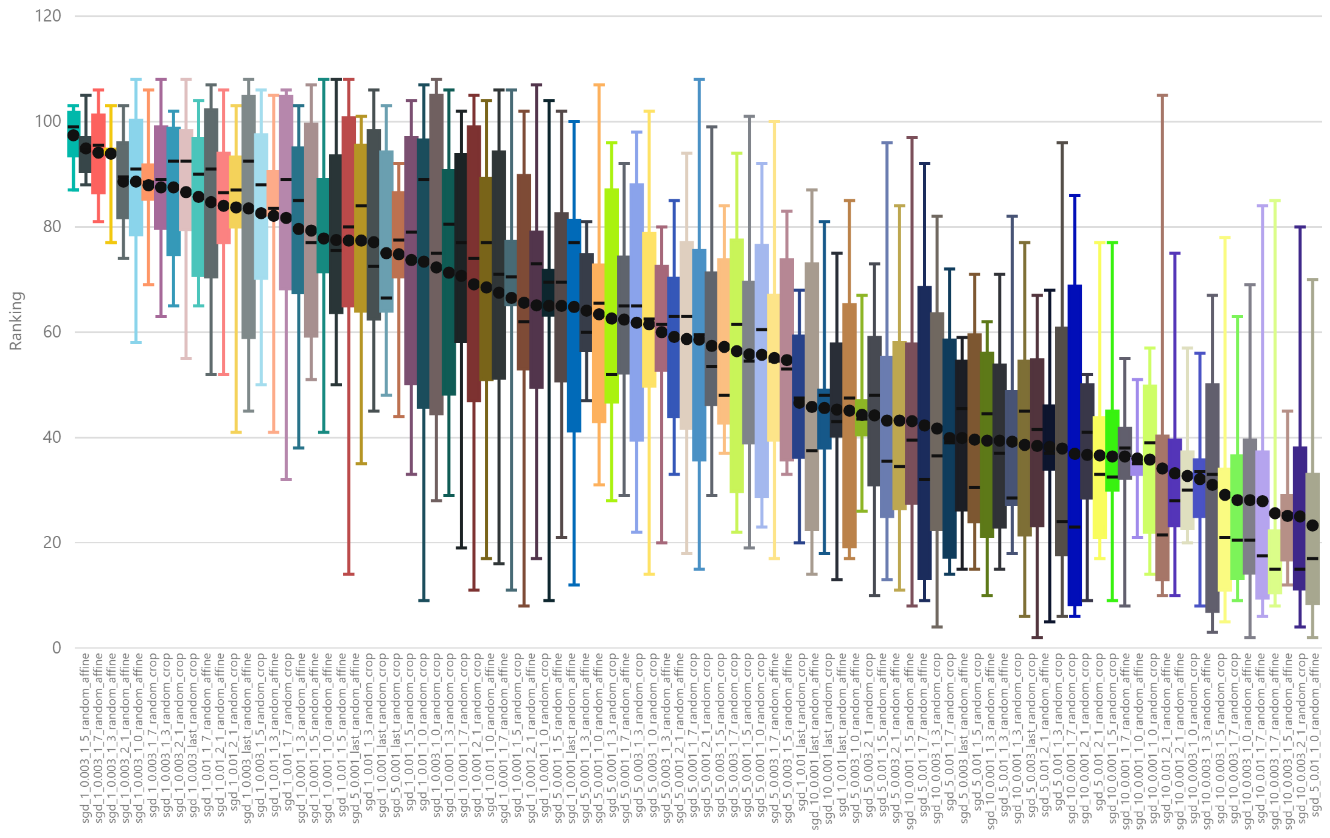

Figure 2. Ranking distribution over the hyperparameter configuration over 10 datasets for ShuffleNet-v2-x1 (higher rank is better)

Table 6. Ablation study for Batch Mean + Similarity Regularization where the precomputed similarity ground truth are generated via AP@K with different K.

	Test Dataset	BookCover30	Caltech256	DeepFashion	Food101	MIT Indoor	IP102 (Pests)	Oxford-IIIT Pets	Places365	SUN397	Textures (DTD)	Average
Ablations	BM + Single Similarity (AP10)	75.93	20.98	33.50	81.61	45.77	51.66	40.33	63.81	47.68	51.55	51.28
	BM + Single Similarity (AP25)	69.58	20.24	34.25	83.61	44.41	50.12	38.95	68.00	45.17	48.20	50.25
	BM + Single Similarity (AP40)	67.90	22.58	35.86	83.63	37.14	45.96	41.57	65.79	43.56	51.45	49.55
	BM + Single Similarity (AP60)	**80.23**	20.51	35.28	**87.10**	38.21	**57.73**	**54.69**	71.78	44.78	46.52	53.68

ing to reduced performance of HyperSTAR. A smaller window size keeps the variance of global dataset representation in check, thus helping the adversarial loss to work well.

Single vs Multiple datasets for similarity based regularization. As described in main text, we impose a regularizer $\mathcal{L}_{\text{sim}}$ to reduce the difference between the cosine similarity of the task representations of two datasets and a precomputed AP@K based similarity ground truth score. This regularizer allows two tasks to have similar representations if they have similar hyperparameter-configuration rankings. In any given iteration of training HyperSTAR where we sample an image batch $\mathcal{B}_m^i$ for a particular dataset $\mathcal{D}_m$, we can randomly choose one or more datasets from the remaining training datasets and impose $\mathcal{L}_{\text{sim}}$ between every two pairs. We experiment and compare this similarity regularization with single pair (BM + Single Similarity) and two pairs (BM + Double Similarity). We do not go further as it incurs a time overhead of forward propagating an additional image batch through the underlying featurizer with every additional similarity regularization, making the empirical analysis intractable while experimenting over all 10 datasets and running 10 trials for each experiment. From Table 5, we can observe that BM + Double Similarity has a 2% higher AP@10 compared to BM + Single Similarity. This suggests that introducing a stronger similarity regularization allows for better representation of the task in the embedding space leading to higher recommendation performance of HyperSTAR.

Similarity Regularization with different AP@K similarity ground truth. Since we define a hyperparameter con-

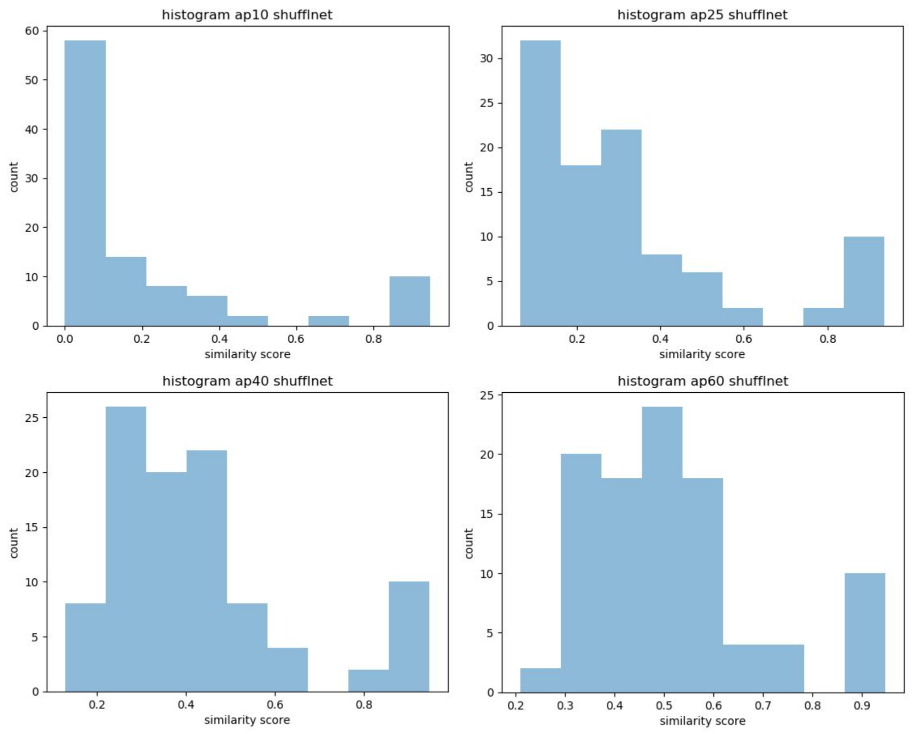

Figure 3. The histogram of pre-computed similarity ground truth via AP@K with different K for ShuffleNet-v2-x1. We found out that there are over 50 pairs of datasets whose similarity scores are zero when using AP@10. To relax the strong regularization which forces the dataset representation to be orthogonal to each other in embedding space, we computed the similarity ground truth with larger number of K for AP@k, making the number of dataset pairs over similarity score more normally distributed.

figuration space of 108 configurations for ShuffleNet-v2-x1, it offers us an opportunity to explore the impact of using similarity ground truth with different AP@K. Table 6 shows a comparison for different BM + Single Similarity settings for ShuffleNet-v2-x1 for $K = 10, 25, 40, 60$. From the table, we can observe that using $K = 60$ gives us the best $AP@10$ among all the ablation settings.

To investigate further into the affect of K in computing the AP@K based similarity ground truth for $\mathcal{L}_{\text{sim}}$, we plot the histogram of the similarity ground truth AP@K values computed between every two datasets for different K in Figure 3. From the figure we can observe that the histogram for AP@10 is very sparse with most of the similarity ground truth values close to or equal to 0. This is because the likelihood of two datasets having common hyperparameter configurations among the top 10 ranks out of 108 is quite small. This is also highlighted in Figures 1 and 2 which show that the variation in the ranking of any particular hyperparameter configuration across datasets is very high. This means that it is very likely that a hyperparameter configuration among the top 10 best performing configurations for a given dataset might be quite lower in rank for some other dataset. This will cause AP@10 to have low or even zero value for many pairs of datasets. We can also observe that as we increase K, the spread of values for AP@K moves increasingly toward 1 with the spread of values being close to normal distribution for $AP@60$. We believe this normality in distribution of AP@60 values allows the similarity regularization to work consistently good across all datasets when using $AP@60$ as the similarity ground truth, thereby improving recommendation performance for HyperSTAR.

7. Network Architectures

This section provides the model definitions for HyperSTAR and also for the two network architectures trained for different hyperparameter configurations to build the meta-dataset $\mathcal{T}$. Since HyperSTAR trains end-to-end over raw images, we have a convolutional network as part of learning the task representation to featurize the images into high level embeddings. As described below, the architecture of this convolutional network is same as the network architecture over which we build the meta-dataset, train HyperSTAR to predict the performance for dataset-configuration pairs and generate a recommendation over the hyperparameter space.

7.1. SE-ResNeXt-50

```
SEResNeXt_50_Featurizer(
  (fcs): ModuleList(
    (0): Linear(in_features=2048, out_features=100, bias=True)
  )
  (features): Sequential(
    (B): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
    (A): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (0): ReLU(inplace=True)
    (1): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): Sequential(
      (1): BottleneckBlock(
        (shortcut_conv): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (shortcut_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv0): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1))
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (seblock): SEBlock(
          (pooling): AdaptiveAvgPool2d(output_size=1)
          (fc0): Linear(in_features=256, out_features=16, bias=True)
          (relu): ReLU(inplace=True)
          (fc1): Linear(in_features=16, out_features=256, bias=True)
          (sigmoid): Sigmoid()
        )
      )
      (2): BottleneckBlock(
        (conv0): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1))
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (seblock): SEBlock(
          (pooling): AdaptiveAvgPool2d(output_size=1)
          (fc0): Linear(in_features=256, out_features=16, bias=True)
          (relu): ReLU(inplace=True)
          (fc1): Linear(in_features=16, out_features=256, bias=True)
          (sigmoid): Sigmoid()
        )
      )
      (3): BottleneckBlock(
        (conv0): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1))
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (seblock): SEBlock(
          (pooling): AdaptiveAvgPool2d(output_size=1)
          (fc0): Linear(in_features=256, out_features=16, bias=True)
          (relu): ReLU(inplace=True)
          (fc1): Linear(in_features=16, out_features=256, bias=True)
          (sigmoid): Sigmoid()
        )
      )
    )
    (3): Sequential(
      (1): BottleneckBlock(
        (shortcut_conv): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (shortcut_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=32, bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1))
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (seblock): SEBlock(
          (pooling): AdaptiveAvgPool2d(output_size=1)
          (fc0): Linear(in_features=512, out_features=32, bias=True)
          (relu): ReLU(inplace=True)
          (fc1): Linear(in_features=32, out_features=512, bias=True)
          (sigmoid): Sigmoid()
        )
      )
      (2): BottleneckBlock(
        (conv0): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1))
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (seblock): SEBlock(
          (pooling): AdaptiveAvgPool2d(output_size=1)
          (fc0): Linear(in_features=512, out_features=32, bias=True)
          (relu): ReLU(inplace=True)
          (fc1): Linear(in_features=32, out_features=512, bias=True)
          (sigmoid): Sigmoid()
        )
      )
      (3): BottleneckBlock(
        (conv0): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1))
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      (seblock): SEBlock(
        (pooling): AdaptiveAvgPool2d(output_size=1)
        (fc0): Linear(in_features=512, out_features=32, bias=True)
        (relu): ReLU(inplace=True)
        (fc1): Linear(in_features=32, out_features=512, bias=True)
        (sigmoid): Sigmoid()
      )
    )
    (4): BottleneckBlock(
      (conv0): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1))
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (seblock): SEBlock(
        (pooling): AdaptiveAvgPool2d(output_size=1)
        (fc0): Linear(in_features=512, out_features=32, bias=True)
        (relu): ReLU(inplace=True)
        (fc1): Linear(in_features=32, out_features=512, bias=True)
        (sigmoid): Sigmoid()
      )
    )
  )
  (4): Sequential(
    (1): BottleneckBlock(
      (shortcut_conv): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (shortcut_bn): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv0): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=32, bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1))
      (bn2): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (seblock): SEBlock(
        (pooling): AdaptiveAvgPool2d(output_size=1)
        (fc0): Linear(in_features=1024, out_features=64, bias=True)
        (relu): ReLU(inplace=True)
        (fc1): Linear(in_features=64, out_features=1024, bias=True)
        (sigmoid): Sigmoid()
      )
    )
    (2): BottleneckBlock(
      (conv0): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1))
      (bn2): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (seblock): SEBlock(
        (pooling): AdaptiveAvgPool2d(output_size=1)
        (fc0): Linear(in_features=1024, out_features=64, bias=True)
        (relu): ReLU(inplace=True)
        (fc1): Linear(in_features=64, out_features=1024, bias=True)
        (sigmoid): Sigmoid()
      )
    )
    (3): BottleneckBlock(
      (conv0): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1))
      (bn2): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (seblock): SEBlock(
        (pooling): AdaptiveAvgPool2d(output_size=1)
        (fc0): Linear(in_features=1024, out_features=64, bias=True)
        (relu): ReLU(inplace=True)
        (fc1): Linear(in_features=64, out_features=1024, bias=True)
        (sigmoid): Sigmoid()
      )
    )
    (4): BottleneckBlock(
      (conv0): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1))
      (bn2): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (seblock): SEBlock(
        (pooling): AdaptiveAvgPool2d(output_size=1)
        (fc0): Linear(in_features=1024, out_features=64, bias=True)
        (relu): ReLU(inplace=True)
        (fc1): Linear(in_features=64, out_features=1024, bias=True)
        (sigmoid): Sigmoid()
      )
    )
    (5): BottleneckBlock(
      (conv0): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1))
      (bn2): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (seblock): SEBlock(
        (pooling): AdaptiveAvgPool2d(output_size=1)
        (fc0): Linear(in_features=1024, out_features=64, bias=True)
        (relu): ReLU(inplace=True)
        (fc1): Linear(in_features=64, out_features=1024, bias=True)
        (sigmoid): Sigmoid()
```

```
          )
        )
        (6): BottleneckBlock(
          (conv0): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn0): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
          (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (conv2): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1))
          (bn2): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (seblock): SEBlock(
            (pooling): AdaptiveAvgPool2d(output_size=1)
            (fc0): Linear(in_features=1024, out_features=64, bias=True)
            (relu): ReLU(inplace=True)
            (fc1): Linear(in_features=64, out_features=1024, bias=True)
            (sigmoid): Sigmoid()
          )
        )
      )
      (5): Sequential(
        (1): BottleneckBlock(
          (shortcut_conv): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (shortcut_bn): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (conv0): Conv2d(1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn0): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (conv1): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=32, bias=False)
          (bn1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (conv2): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(1, 1))
          (bn2): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (seblock): SEBlock(
            (pooling): AdaptiveAvgPool2d(output_size=1)
            (fc0): Linear(in_features=2048, out_features=128, bias=True)
            (relu): ReLU(inplace=True)
            (fc1): Linear(in_features=128, out_features=2048, bias=True)
            (sigmoid): Sigmoid()
          )
        )
        (2): BottleneckBlock(
          (conv0): Conv2d(2048, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn0): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (conv1): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
          (bn1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (conv2): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(1, 1))
          (bn2): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (seblock): SEBlock(
            (pooling): AdaptiveAvgPool2d(output_size=1)
            (fc0): Linear(in_features=2048, out_features=128, bias=True)
            (relu): ReLU(inplace=True)
            (fc1): Linear(in_features=128, out_features=2048, bias=True)
            (sigmoid): Sigmoid()
          )
        )
        (3): BottleneckBlock(
          (conv0): Conv2d(2048, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn0): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (conv1): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
          (bn1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (conv2): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(1, 1))
          (bn2): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
          (relu): ReLU(inplace=True)
          (seblock): SEBlock(
            (pooling): AdaptiveAvgPool2d(output_size=1)
            (fc0): Linear(in_features=2048, out_features=128, bias=True)
            (relu): ReLU(inplace=True)
            (fc1): Linear(in_features=128, out_features=2048, bias=True)
            (sigmoid): Sigmoid()
          )
        )
      )
      (6): AdaptiveAvgPool2d(output_size=1)
  )
)
SEResNeXt_50_Classifier(
  (mlp): Linear(in_features=2048, out_features=<num_classes>, bias=True)
)
```

7.2. ShuffleNet-v2-x1

```
ShuffleNet_v2_x1_Featurizer(
  (fcs): ModuleList(
    (0): Linear(in_features=1024, out_features=100, bias=True)
  )
  (features): Sequential(
    (D): Conv2d(3, 24, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (C): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (B): ReLU(inplace=True)
    (A): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (0): Sequential(
      (0): DownSamplingBasicBlock(
        (conv0): Conv2d(24, 58, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv1): Conv2d(58, 58, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=58, bias=False)
        (bn1): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(58, 58, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn2): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(24, 24, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=24, bias=False)
        (bn3): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv4): Conv2d(24, 58, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn4): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (shuffle): ChannelShuffle()
      )
      (1): BasicBlock(
      (conv0): Conv2d(58, 58, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv1): Conv2d(58, 58, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=58, bias=False)
      (bn1): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(58, 58, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn2): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shuffle): ChannelShuffle()
    )
    (2): BasicBlock(
      (conv0): Conv2d(58, 58, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv1): Conv2d(58, 58, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=58, bias=False)
      (bn1): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(58, 58, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn2): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shuffle): ChannelShuffle()
    )
    (3): BasicBlock(
      (conv0): Conv2d(58, 58, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv1): Conv2d(58, 58, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=58, bias=False)
      (bn1): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(58, 58, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn2): BatchNorm2d(58, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shuffle): ChannelShuffle()
    )
  )
  (1): Sequential(
    (0): DownSamplingBasicBlock(
      (conv0): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv1): Conv2d(116, 116, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=116, bias=False)
      (bn1): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn2): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(116, 116, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=116, bias=False)
      (bn3): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv4): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn4): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shuffle): ChannelShuffle()
    )
    (1): BasicBlock(
      (conv0): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv1): Conv2d(116, 116, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=116, bias=False)
      (bn1): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn2): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shuffle): ChannelShuffle()
    )
    (2): BasicBlock(
      (conv0): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv1): Conv2d(116, 116, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=116, bias=False)
      (bn1): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn2): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shuffle): ChannelShuffle()
    )
    (3): BasicBlock(
      (conv0): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv1): Conv2d(116, 116, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=116, bias=False)
      (bn1): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn2): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shuffle): ChannelShuffle()
    )
    (4): BasicBlock(
      (conv0): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv1): Conv2d(116, 116, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=116, bias=False)
      (bn1): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn2): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shuffle): ChannelShuffle()
    )
    (5): BasicBlock(
      (conv0): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv1): Conv2d(116, 116, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=116, bias=False)
      (bn1): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn2): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shuffle): ChannelShuffle()
    )
    (6): BasicBlock(
      (conv0): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn0): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv1): Conv2d(116, 116, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=116, bias=False)
      (bn1): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn2): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shuffle): ChannelShuffle()
    )
    (7): BasicBlock(
        (conv0): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv1): Conv2d(116, 116, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=116, bias=False)
        (bn1): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(116, 116, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn2): BatchNorm2d(116, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (shuffle): ChannelShuffle()
      )
    )
    (2): Sequential(
      (0): DownSamplingBasicBlock(
        (conv0): Conv2d(232, 232, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv1): Conv2d(232, 232, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=232, bias=False)
        (bn1): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(232, 232, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn2): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(232, 232, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=232, bias=False)
        (bn3): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv4): Conv2d(232, 232, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn4): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (shuffle): ChannelShuffle()
      )
      (1): BasicBlock(
        (conv0): Conv2d(232, 232, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv1): Conv2d(232, 232, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=232, bias=False)
        (bn1): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(232, 232, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn2): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (shuffle): ChannelShuffle()
      )
      (2): BasicBlock(
        (conv0): Conv2d(232, 232, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv1): Conv2d(232, 232, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=232, bias=False)
        (bn1): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(232, 232, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn2): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (shuffle): ChannelShuffle()
      )
      (3): BasicBlock(
        (conv0): Conv2d(232, 232, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn0): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv1): Conv2d(232, 232, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=232, bias=False)
        (bn1): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(232, 232, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn2): BatchNorm2d(232, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (shuffle): ChannelShuffle()
      )
    )
    (3): Conv2d(464, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (4): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): AdaptiveAvgPool2d(output_size=1)
  )
)
ShuffleNet_v2_x1_Classifier(
  (mlp): Linear(in_features=1024, out_features=<num_classes>, bias=True)
)
```

7.3. HyperSTAR: SE-ResNeXt-50

```
HyperSTAR_SE_ResNeXt_50(
  (SEResNext_50_Featurizer): (
      <described above>
   )
  (Transformer): (
      (mlp): Linear(in_features=2048, out_features=2048, bias=True)
   )
  (config_features): Sequential(
    (0): Linear(in_features=13, out_features=256, bias=True)
    (1): ReLU(inplace=True)
  )
  (visual_features): Sequential(
    (0): Linear(in_features=2048, out_features=256, bias=True)
    (1): ReLU(inplace=True)
  )
  (performance_prediction_network): Sequential(
    (0): Linear(in_features=512, out_features=256, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=256, out_features=256, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=256, out_features=256, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=256, out_features=1, bias=True)
  )
)
```

7.4. HyperSTAR: ShuffleNet-v2-x1

```
HyperSTAR_ShuffleNetv2(
  (ShuffleNet_v2_x1_Featurizer): (
      <described above>
   )
  (Transformer): (
      (mlp): Linear(in_features=1024, out_features=1024, bias=True)
```

```
    )
  (config_features): Sequential(
    (0): Linear(in_features=14, out_features=256, bias=True)
    (1): ReLU(inplace=True)
  )
  (visual_features): Sequential(
    (0): Linear(in_features=1024, out_features=256, bias=True)
    (1): ReLU(inplace=True)
  )
  (performance_prediction_network): Sequential(
    (0): Linear(in_features=512, out_features=256, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=256, out_features=256, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=256, out_features=256, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=256, out_features=1, bias=True)
  )
)
```

References

[1] Rémi Bardenet, Mátyás Brendel, Balázs Kégl, and Michele Sebag. Collaborative hyperparameter tuning. In *Proc. of the International Conference on Machine Learning*, pages 199–207, 2013. 2

[2] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *Proc. of the International Conference on Machine Learning*, pages 647–655, 2014. 3

[3] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Proc. of the Advances in Neural Information Processing Systems*, pages 2962–2970, 2015. 2

[4] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *Proc. of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. 2

[5] Yaroslav Ganin and Victor Lempitsky. Unsupervised domain adaptation by backpropagation. *arXiv preprint arXiv:1409.7495*, 2014. 4

[6] Judy Hoffman, Eric Tzeng, Taesung Park, Jun-Yan Zhu, Phillip Isola, Kate Saenko, Alexei Efros, and Trevor Darrell. Cycada: Cycle-consistent adversarial domain adaptation. In *Proc. of the 35th International Conference on Machine Learning*, 2018. 4

[7] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018. 1

[8] Nikolaos Karianakis, Zicheng Liu, Yinpeng Chen, and Stefano Soatto. Reinforced temporal attention and split-rate transfer for depth-based person re-identification. In *Proc. of the The European Conference on Computer Vision*, September 2018. 1

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. 1

[10] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proc. of the European Conference on Computer Vision*, pages 116–131, 2018. 1

[11] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951. 1

[12] Eric Tzeng, Judy Hoffman, Kate Saenko, and Trevor Darrell. Adversarial discriminative domain adaptation. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7167–7176, 2017. 4

[13] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Proc. of the Advances in Neural Information Processing Systems*, pages 3320–3328, 2014. 3