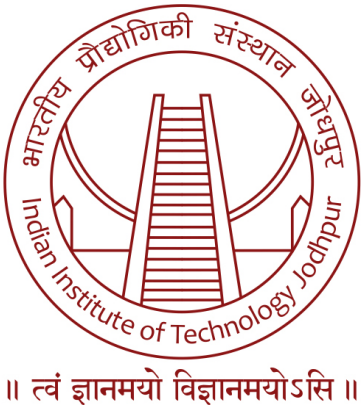
RCAI: Machine Learning on Resources Constrained Devices

A Project Report Submitted by

Jaideep Singh Heer

in partial fulfillment of the requirements for the award of the degree of

M.Tech. AI



Indian Institute of Technology Jodhpur

Department of CSE

May, 2022

Declaration

I hereby declare that the work presented in this Project Report titled RCAI: Machine Learning on Resources Constrained Devices submitted to the Indian Institute of Technology Jodhpur in partial fulfilment of the requirements for the award of the degree of M.Tech. AI, is a bonafide record of the research work carried out under the supervision of Dr. Deepak Mishra. The contents of this Project Report in full or in parts, have not been submitted to, and will not be submitted by me to, any other Institute or University in India or abroad for the award of any degree or diploma.

Signature

Jaideep Singh Heer

M20CS056

Certificate

This is to certify that the Project Report titled RCAI: Machine Learning on Resources Constrained Devices, submitted by Jaideep Singh Heer (M20CS056) to the Indian Institute of Technology Jodhpur for the award of the degree of M.Tech. AI, is a bonafide record of the research work done by him under my supervision. To the best of my knowledge, the contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Signature

Dr. Deepak Mishra

Acknowledgements

The success and final outcome of this project required a lot of guidance and assistance from my project advisor Dr. Deepak Mishraand I am extremely grateful for the support and knowledge that I received from him. I would like to express my gratitude to the IIT Jodhpur DGX2 server staff for providing me with the necessary computational resources for my project and for providing timely support. Finally, I would not forget to remember Mr. Wilfred Kisku who helped me understand the intricacies of hardware design and neural network accelerators.

Abstract

Modern machine learning models are built using large cloud computing resources and are often deployed as cloud services. This is due to the fact that most modern machine learning models have millions (if not billions) of parameters and require tera-flops of computational capacity to function.

To bring machine learning from cloud services to local devices, there is a need to devise methods that convert large models into smaller counterparts and compute these smaller counterparts in the most efficient way possible. This work focuses on two promising approaches, Neural Architecture Search [1] with FLOPs/latency weighted architecture parameters [2] and Iterative Denoising Diffusion Neural Networks [3]. The idea is to reduce the cost of computation of a state of the art iterative model to reduce the total computation cost by a factor of the total number of iterations.

This project focuses on the task of performing neural architecture search on a image super-resolution model inspired by [4]. For the same, this project adopts a DARTS [1] like approach, where a supernet is created with multiple parallel operations applied and their results added after scaling them with architecture parameters. To introduce FLOPs/latency reduction, this project takes inspiration from [2] by calculating the FLOPs/latency cost of each block in the network and propagating it by simple addition and scaling with the architecture parameters.

Contents

List of Figures

List of Tables

RCAI: Machine Learning on Resources Constrained Devices

1 Introduction and background

Machine learning with deep neural networks is expensive not only to train but also to infer. Most state of the art neural network models have close to ten billion parameters and such large models are not only very expensive to train but also require expensive hardware for inference.

A large part of a model's computational cost comes from the cost of moving data between multiple layers of memory that are present in a traditional computer system. For a billion parameter model, such data movement must happen multiple times as large matrices are brought in and out of memory in smaller chunks. One can easily see how a traditional computer system design can lead to high power consumption for large neural network inference.

This is where the motivation for Neural Architecture Search (NAS) comes into the picture. First introduced in [5], the task for a NAS algorithm is to find the best neural network architecture such that the accuracy obtained over the test dataset is maximized. Since their use initially for improving accuracy, many different NAS algorithms have been developed to control not only the accuracy but also other metrics of the searched architecture such as its depth, width and resolution [6]. Newer NAS algorithms implement searching for networks while minimizing constraints such as network depth, width and even FLOPs [2].

At the same time, there are also iterative neural network systems, like Denoising Diffusion Probabilistic Models (DDPM) [3], that apply a smaller neural network model on some input data iteratively to get the final output. This allows a trade-off between latency and memory consumption since the iterative process is usually very slow due to the neural network model being applied many times.

This work focuses on bringing NAS with FLOPs/latency metrics and DDPM models closer with the goal of creating a NAS algorithm that finds the best trade-off between neural network architecture and FLOPs/latency execution cost for an iterative DDPM model. The specific task that this work focuses on is image super-resolution using iterative refinement, using the model and DDPM technique from [4].

2 Literature survey

Most research in neural architecture search has mainly been focused at improving model accuracy and there have been many proposals for such NAS algorithms, [7] is a survey on such algorithms and techniques. It defines a NAS algorithm having three parts, a search space, a search strategy and a performance estimation strategy. Depending on how each of these are defined, different NAS algorithms can be created. In such a setup, a *vanilla* NAS algorithm can be defined as one that simply generates architectures, trains them to convergence, evaluates them and updates its generator based on the evaluation.

Most initial NAS approaches used such a vanilla NAS approach along with Evolutionary Algorithms (EA) or Bayesian Optimisation (BO) to generate new samples for network architectures [8] [6] [9] [10]. There are even approaches that use variants of the genetic algorithm to optimize hardware parameters like number of PEs, power consumption, etc. while performing the overall NAS in a vanilla fashion by evaluating each sample individually on the hardware [11] [9] [10]. However the cost of running such NAS algorithms is very high since one must evaluate each architecture individually and the generator learns to generate good architecture slowly.

To avoid having to train and evaluate all generated models in the search space, a popular approach is to design a *one-shot* NAS algorithm which generates only a single architecture (called a *super-network* or a *supernet*) [12] [1]. In such a *one-shot* approach the *supernet* is trained only once and after training, a *sub-network* (or *subnet*) is extracted from it by pruning some of the supernet's operations. [13] combined the vanilla NAS method with this *one-shot* method by progressively freezing the super-network's operations and training all sub-networks derived by freezing a single operation. [14] introduces more focus on inference hardware constraints by designing a one-shot model that can be pruned to extract sub-models of various sizes, thereby allowing a single one-shot model to be used for generating inference models for multiple hardware constraints. This work however, focuses on the classic *one-shot* NAS approaches, mainly DARTS [1] and P-DARTS [12].

To optimize over FLOPs/latency metrics using gradient descent, we can include them in the loss function and have our NAS algorithm automatically optimize the architecture parameters to reduce FLOPs/latency. However, these metrics are non-differentiable and have no relation to the model's output since they are generally measured using the input and output shapes of operations, which are fixed for every operation. This means that the FLOPs/latency measurements of each operation must be done separately and propagated through the model such that, when back propagated, the model's parameters can be updated. This is the approach applied by [2] and is the same approach that this project applies.

Diffusion models were first introduced in [15] and are generative model inspired by non-equilibrium thermodynamics. They employ a Markov chain of *diffusion steps* that add small amounts of noise into the input. The model's objective is then to learn to reverse the Markov process and iteratively remove

the noise to reconstruct the original sample. [16] is a great article explaining how a diffusion model works and how it is different from a GAN, VAE or a flow based model. This work uses the diffusion process and the model described in [4] as a base case to apply NAS over.

3 Problem definition and Objective

The target problem is to perform neural architecture search on a diffusion model based on SR3 [4] for the image super-resolution task on the DIV2k dataset [17] [18] [19] [20].

Neural Architecture Search (NAS) can be modeled as a bi-level optimization problem, where the inner optimization is for a single model on the given task and the outer optimization is for the network architecture. For *one-shot* NAS the bi-level optimization can be seen as optimizing two sets of parameters of the super-network, i.e. the model parameters and the architecture parameters.

Definition 3.1. Given a loss function L and a neural network $f : R^A \times R^M \rightarrow R$, where $m \in R^M$ are model parameters and $a \in R^A$ are architecture parameters, the bi-level optimization problem is defined as,

$$min_{a \in R^A} \; L(f(a, m^*(a))) \tag{1}$$

$$s.t. \quad m^*(a) \in argmin_{m \in R^M} \; L(f(a, m)) \tag{2}$$

Most differentiable NAS algorithms perform this bi-level optimization by alternatively updating the model parameters $m \in R^M$ and the architecture parameters $a \in R^A$ during model training [12] [1] [2] [21] [13] [22]. After training the architecture parameters a are used to sample a sub-model from the super-network.

A diffusion model on the other hand uses a Markov chain to perform forward diffusion during training and reverse diffusion during inference.

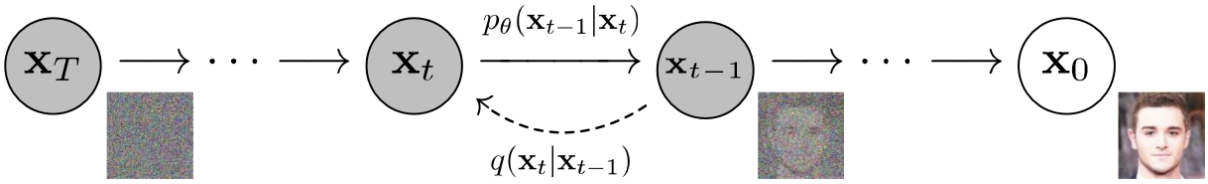


Figure 3.1: Forward and reverse diffusion process [4]

Figure 3.1 shows the overall diffusion process that this work follows. Here x_0 is the original/target noiseless image and x_t is the image obtained after t iterations of adding noise using the forward diffusion process $q(x_t|x_{t-1})$. During training we perform the diffusion process up to some randomly sampled point $t \in [0, T]$ and we train our diffusion model to predict $p_\theta(x_{t-1}|x_t)$. Here T is the total number of diffusion steps used in the diffusion model and should be set such that x_T resembles Gaussian noise, i.e. $p(x_T) = \mathcal{N}(x_T; 0, I)$. This allows our generative model to start from pure Gaussian noise during inference and re-create x_0.

Following [3], since the forward diffusion process $q(x_t|x_{t-1})$ is a fixed Markov chain that adds Gaussian

noise to the data, it can be defined as,

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1}), \quad q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I) \tag{3}$$

In equation 3 $\beta_1 \dots \beta_t$ are the variance schedule values that define the amount of noise added in every step of the forward diffusion process. These β_t values can be made trainable however similar to [4] this work uses a fixed beta variance schedule. Further [3] shows that if training is to be done by optimizing the negative log likelihood then forward diffusion process can be conducted in a single step for any time step t. This is done by introducing new terms $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_t$ by using the formula,

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)\boldsymbol{I}) \tag{4}$$

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t \boldsymbol{I}) \tag{5}$$

$$\text{where} \quad \tilde{\mu}_t(x_t, x_0) = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0 + \frac{\sqrt{\bar{\alpha}_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}x_t \quad \text{and} \quad \tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t \tag{6}$$

Following the above equations we can perform efficient training using equation 4 to generate a noisy image at any time step t and equations 5 and 6 to efficiently regenerate x_{t-1} given x_t and x_0. During model inference however, x_0 is unknown and thus reverse inference cannot be done. This is where a neural network model ϵ_θ is used to predict the noise to be removed from x_t to get x_{t-1}. [3] thus propose the following training and inference algorithms,

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$
6: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

Figure 3.2: Diffusion model training and inference [3]

Following the above by [3], [4] makes a small change by replacing $\bar{\alpha}_t$ with a new term $\gamma \sim U(\bar{\alpha}_{t-1}, \bar{\alpha}_t)$ only during training This is included in the diffusion algorithm used for this work. This work also performs gradient descent on the L1 norm instead of the L2 norm as shown in the algorithm. This makes the gradient descent step as,

$$\nabla_\theta ||\epsilon - \epsilon_\theta(\underbrace{\sqrt{\gamma}x_0 + \sqrt{1-\gamma}\epsilon}_{y_t}, t)||, \quad \gamma \sim U(\bar{\alpha}_{t-1}, \bar{\alpha}_t) \tag{7}$$

Finally, this diffusion algorithm is combined with NAS using the classic DARTS [1] approach where the neural network ϵ_θ is created as a super-network and is trained on the train data for 100 epochs. Then a sub-network is extracted and is fine-tuned for another 100 epochs on the same train data.

4 Methodology

To perform image super-resolution, this work uses the SR3 model [4] as the base model for comparison. The super-resolution task is to convert image size from 64x64 to 128x128. Following [4] this work first up-scales and interpolates the input low-resolution (LR) 64x64 image to target high-resolution 128x128 using bi-cubic up-scaling.

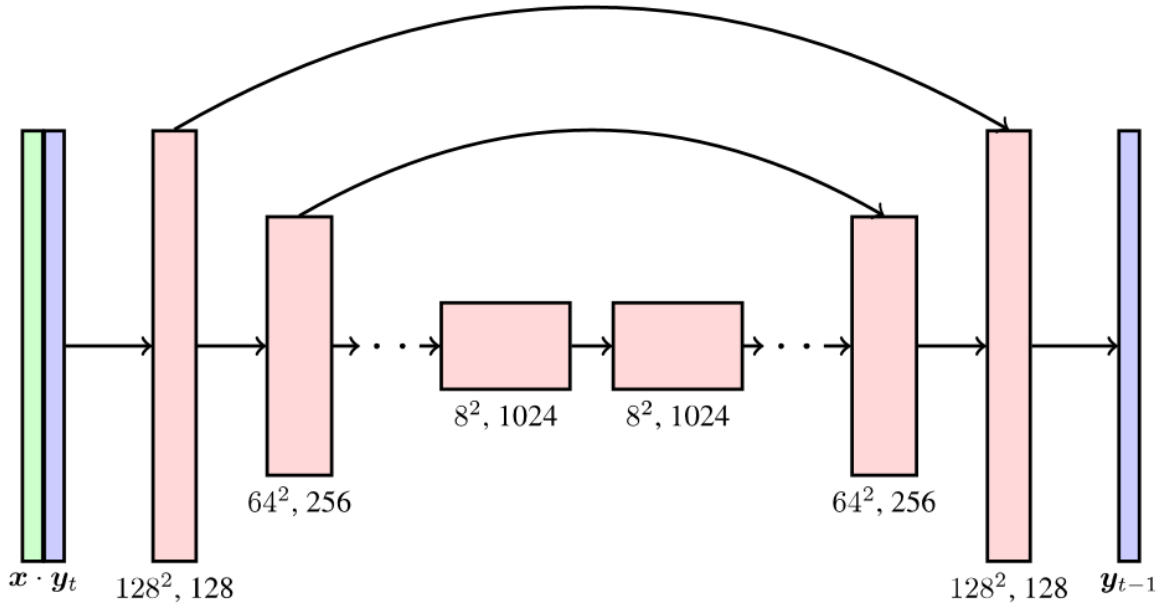


Figure 4.1: UNet model with skip connections showing image-size, channels per UNet block and input x and y_t concatenated to predict y_{t-1} [4]

During training, the HR image y is first converted into a noisy HR image $y_t = \sqrt{\gamma}x_0 + \sqrt{1-\gamma}\epsilon$ using equation 4 and 7. This noisy image y_t is then concatenated with the up-scaled LR image x along the channel dimension to create a 6x128x128 input tensor. Doing this allows the model's output to be conditioned on the input data x, which is useful during inference when y_T is pure Gaussian noise. Figure 4.1 shows the basic structure of the UNet with skip connections. Here, each inner rectangle represents a UNet block which contains n ResNet blocks. This work sets $n = 1$ ResNet blocks for all UNet blocks to keep the model size reasonable.

Along with the 6x128x128 tensor, the model also receives the γ value as an indication of the level of noise to remove from y_t. This γ value is first converted into a positional encoding using the method described in [23] and is then used to apply feature-wise transformations using the FiLM [24] technique.

To apply DARTS like NAS to such a UNet model, this work implements a *Multi-Operation* block which takes a single input and passes it to multiple operations to get their outputs. The outputs of each operation are scaled by a learnable parameter of the Multi-Operation block. These Multi-Operation parameters

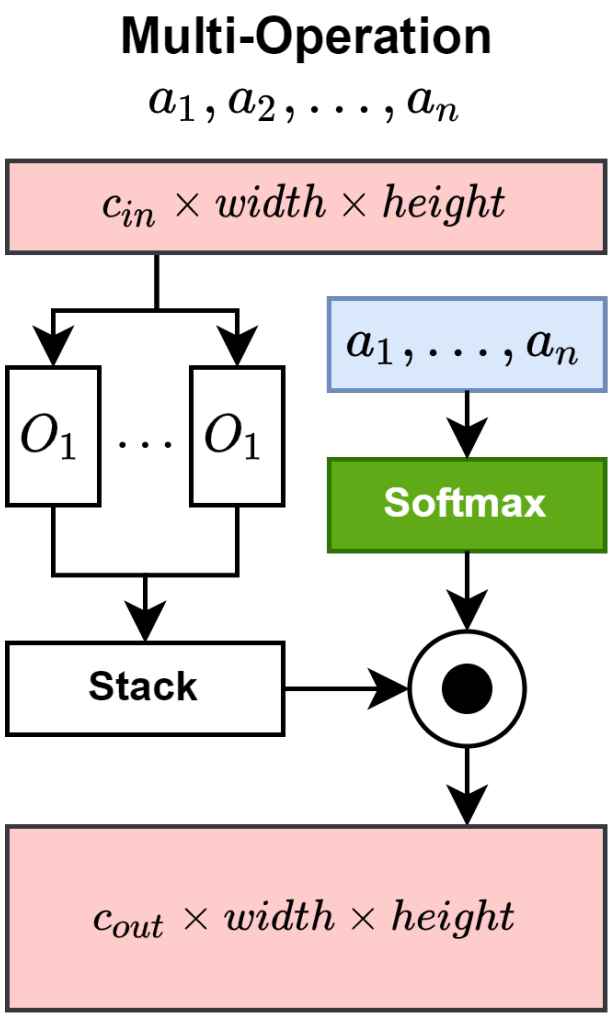


Figure 4.2: A Multi-Operation block with $a_1, ..., a_n$ as learnable architecture parameters, $O_1, ..., O_n$ as operations in selection set and a softmax function which is applied to them for scaling each operation's output with its corresponding a_i

are referred to as the super-network's **architecture parameters**. To ensure that the architecture parameters only perform operation-relative scaling and do not interfere with the operation's outputs, the architecture parameters are passed through a softmax function before they scale the operation outputs. This work considers three variants of the softmax function functions used in all Multi-Operation blocks and compares their effect on training and model performance. Figure 4.2 shows the basic data-flow of the Multi-Operation block.

Figure 4.3 shows the ResNet blocks that are placed sequentially in each UNet block. Each UNet block has n ResNet blocks followed by or preceded by a scaling operation depending on which side of the UNet the block is present in. To introduce NAS into the UNet architecture this work simply places Multi-Operation blocks in both ResNet blocks and Group Norm Conv blocks with 2 and 3 operations respectively. Here the ResNet block contains a Self-Attention/Multi-Operation block however this block is configurable to be enabled only after a given depth in the UNet. In this work, the Self-Attention/Multi-

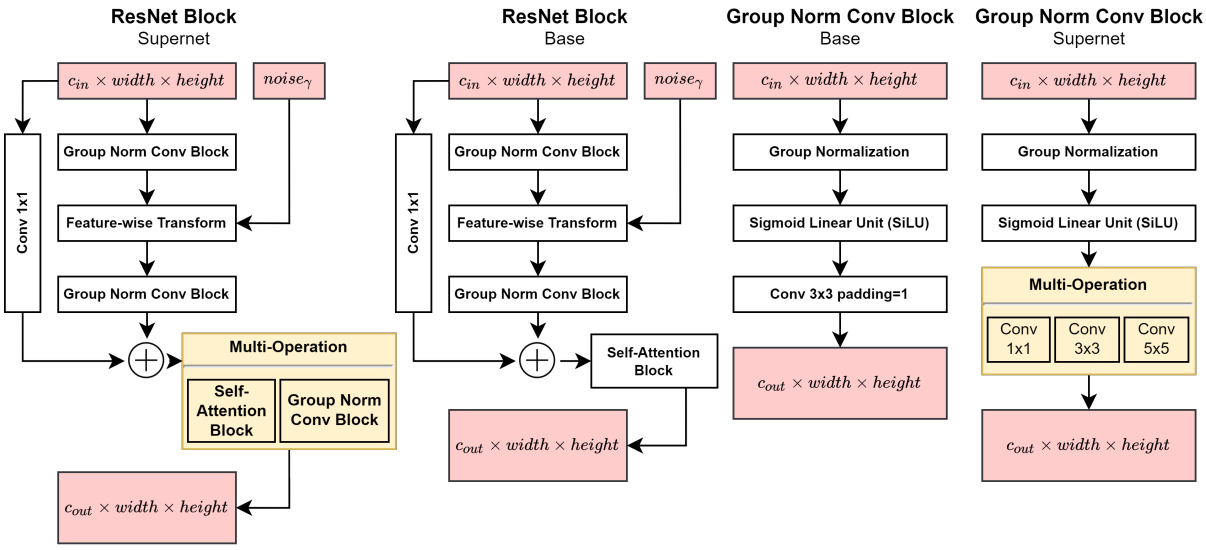


Figure 4.3: ResNet and Group Norm Convolution blocks for base network and super-network

Operation in the ResNet block starts from the 3rd UNet depth level. Note that the Group Norm Conv block placed in the ResNet block's Multi-Operation will also have a Multi-Operation block inside it so the total NAS search space size for 1 ResNet block is $3*3*(1+3) = 36$ if Self-Attention/Multi-Operation block is enabled and is $3*3 = 9$ if it is disabled.

This work implements a UNet with 5 layers with channel multipliers of $1, 2, 4, 8, 16$ for layers from top to bottom with the base number of channels of 64. The Self-Attention/Multi-Operation block in ResNet blocks is enabled from depth 3 on-wards, i.e. for the last 2 layers of the UNet. This makes the super-network's NAS search space size $9*9*9*36*36 = 944,784$ possible sub-networks.

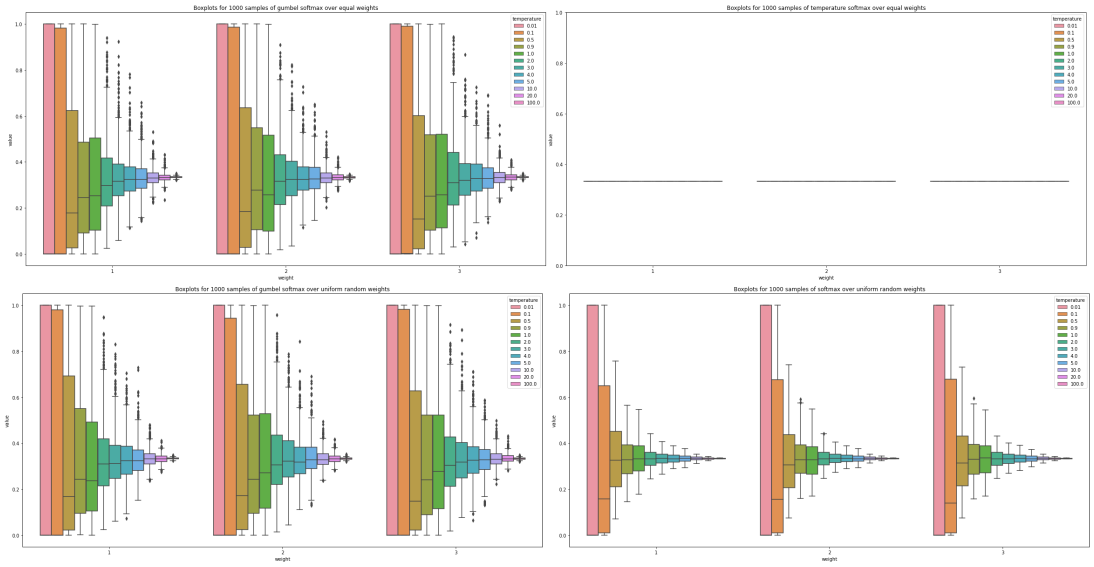


Figure 4.4: Boxplots showing Gumbel-Softmax (left) vs Softmax with temperature (right) function behaviour across 1000 samples over **equal** weights (above) and **uniformly sampled** weights (below)

Since the architecture parameters are passed through an softmax function before scaling, the choice for the softmax function plays an important role in the overall training process. This work explores three softmax function variants namely, **Simple softmax**, **Softmax with temperature** and **Gumbel-Softmax**. The latter two functions have a configurable temperature property that controls the behaviour of each function. Figure 4.4 shows that Gumbel-Softmax becomes more aggressively biased as the temperature decreases while Softmax with temperature properly respects the actual weight values over which it is applied. This behaviour is due to the randomness in the Gumbel-Softmax due to sampling from the Gumbel distribution.

To configure the temperature for Gumbel-Softmax and Softmax with temperature this work uses a tanh based function defined as,

$$temperature(x) = \left(1 - tanh(\frac{3 * (x - \frac{width}{2})}{\frac{width}{2}})\right) * \frac{max - min}{2} - min \tag{8}$$

Here, max and min define the range of the function and width defines the domain. This work uses $max = 20$, $min = 0.08$ and $width =$ no. of total steps.

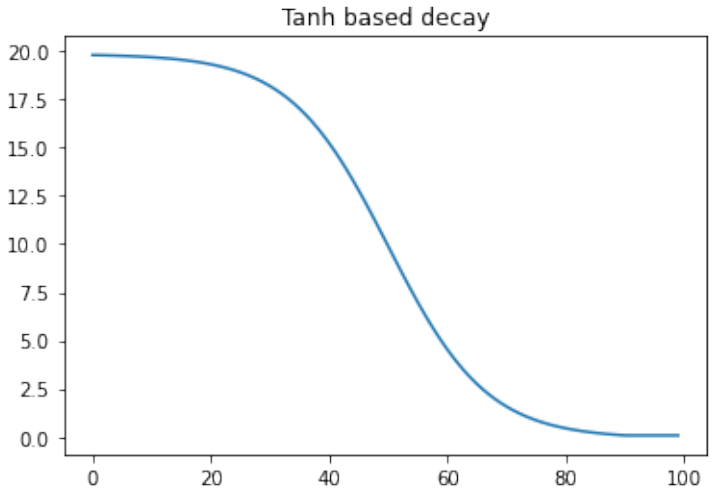


Figure 4.5: Tanh based temperature decay function with $max = 20$, $min = 0.08$ and $width = 100$

To include FLOPs/latency metrics in the loss for minimizing model size, this work uses a approach similar to [2]. Every module in the UNet is measured using the DeepSpeed profiler for PyTorch [25] and the FLOPs/latency metrics are propagated through the model in the same graph as the data. Only in this case, instead of multiplying the tensors with operation weights, the tensors are added with each operation's measured metrics and propagated forward. The only exception is the Multi-Operation block which scales these metric tensors as they are propagated through it, thus allowing back-propagation via the FLOPs/latency tensors to affect only the architecture parameters.

4.1 Training pipeline

The full training pipeline that this work employs is as follows,

- Instantiate a model, a model training diffusion engine, an architecture training diffusion engine and two inference diffusion engines for validation and testing.

  Model: The model is the UNet/supernet-UNet neural network that will be used for prediction of noise and must take as input a 6x128x128 tensor along with a scalar γ.

  Model training diffusion engine: This is the code that performs model training using the Algorithm 1 in 3.2.

  Architecture training diffusion engine This engine is the same as model training engine but it only trains the architecture parameters and uses samples from the validation dataset. This engine is also responsible for propagating the FLOPs/latency metrics through the model to add down-scaled versions of these metrics to the loss tensor. Doing so allows the architecture parameters to be trained with weighted FLOPs/latency in the loss.

  Inference diffusion engine This is the code that performs model inference using Algorithm 2 in 3.2. This engine performs all the diffusion steps starting from pure Gaussian noise and a conditioning tensor x to generate the final output $\tilde{y}_0$. It can also provide the intermediate tensors $\tilde{y}_t$ at any requested time step $0 < t < T$. This engine is used to generate the final images for validation, inference and testing.

- Train the model.

  Non-supernet If the model is not a supernet, run the model training engine for 2000 epochs and validate every 250 epochs.

  Supernet If the model is a supernet model, run the model training engine and architecture training engine one epoch at a time alternating between both for 100 epochs each (total 200 epochs). Update all Multi-Operation softmax function temperatures at every batch iteration of the model train engine. Run validation engine once at the end of 100 epochs of both train engines.

- If model is a supernet extract the sub-network by iterating over all Multi-Operation blocks and disabling all operations except the one corresponding to the argmax of the Multi-Operation's architecture parameters. This is equivalent to how sub-network extraction is done in DARTS [1]. After extracting the subnet run the model training diffusion engine for another 100 epochs and run the validation diffusion engine once this ends.

- Finally run the testing inference diffusion engine using the test dataset on the final model, i.e. trained non-supernet model or fine-tuned sub-network model.

Table 4.1: Training pipeline common hparams

Item	hparam	Value
Model training engine	lr	3e-06
Architecture training engine	lr	3e-05
Architecture training engine	temperature	Eq. 8 create-tanh-decay(max=20,min=0.08)
Architecture training engine	flops-loss-scaling	5e-17
Architecture training engine	latency-loss-scaling	5e-5
All train engines	optimizer	torch.optim.Adam
All train engines	loss	L1Loss(reduction=mean)
All engines	T	2000
All engines	$\beta_1, ..., \beta_T$	torch.linspace(start=1e-06, end=0.01, steps=2000)
Data-loader	batch-size	8
Data-loader	crop-patch-size	128
Data-loader	up-scaling	bi-cubic
Data-loader	dataset	DIV2K-unknownx2
Task	super-resolution	64x64 to 128x128
Model	start-channels	64
Model	channel-multipliers	1, 2, 4, 8, 16
Model	attention-from-depth	3
Model	resnet-blocks-per-unet-layer	1

5 Theoretical/Numerical/Experimental findings

This work focuses on running the explained NAS training pipeline on three identical models with the common hyper-parameters 4.1 using Simple softmax, Softmax with temperature and Gumbel-Softmax as the Multi-Operation softmax respectively.

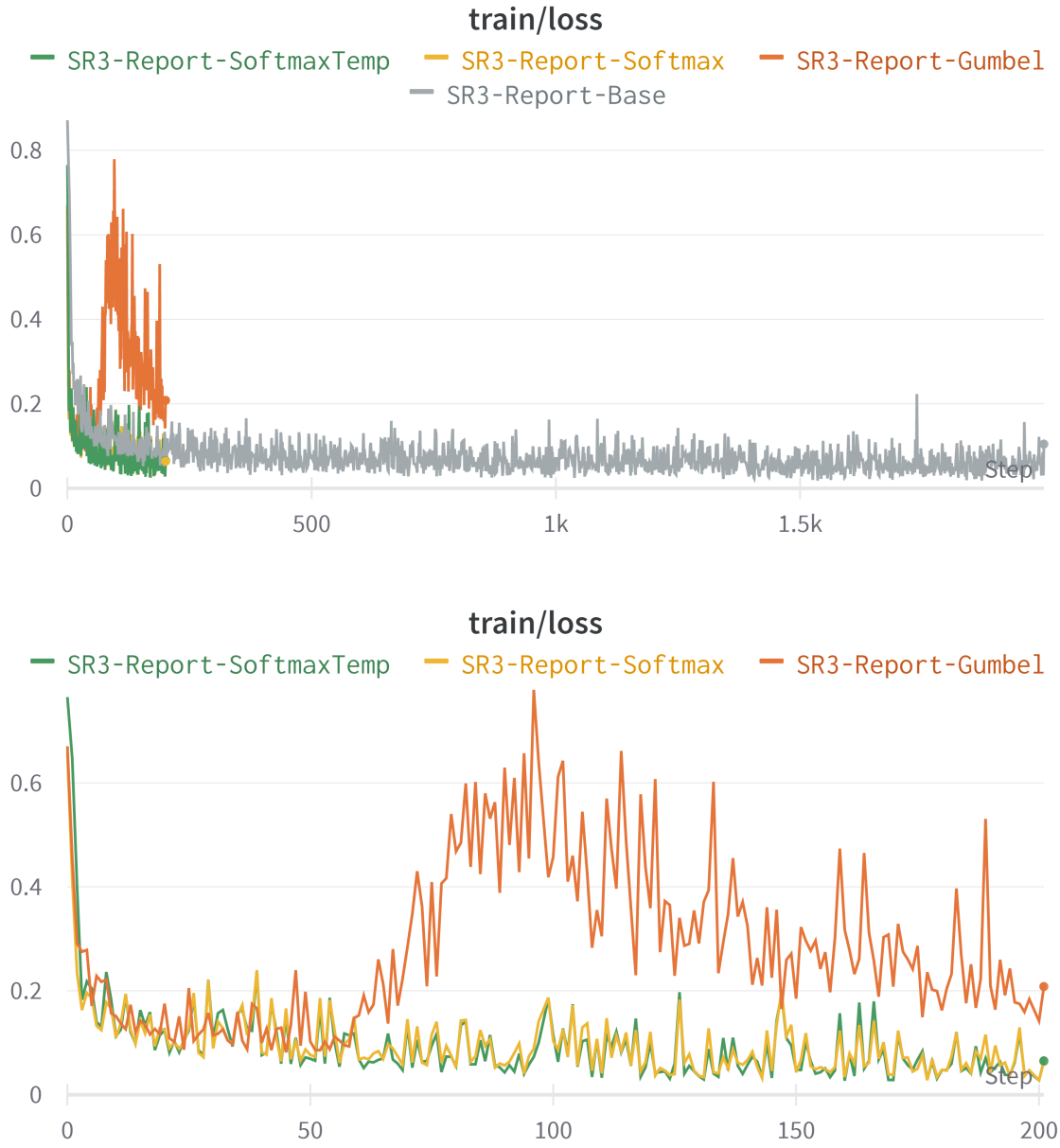


Figure 5.1: Model training engine loss for all models (above) and NAS models only (below)

Figure 5.1 shows that training with Gumbel-Softmax causes the model training loss to increase as the softmax temperature decreases and the increase in loss is very significant compared to all other methods. It also shows that the train loss is almost similar in case of Softmax and Softmax with temperature. In-fact, Softmax with temperature does not cause any difference in the model training loss compared to

only Softmax or the base model.

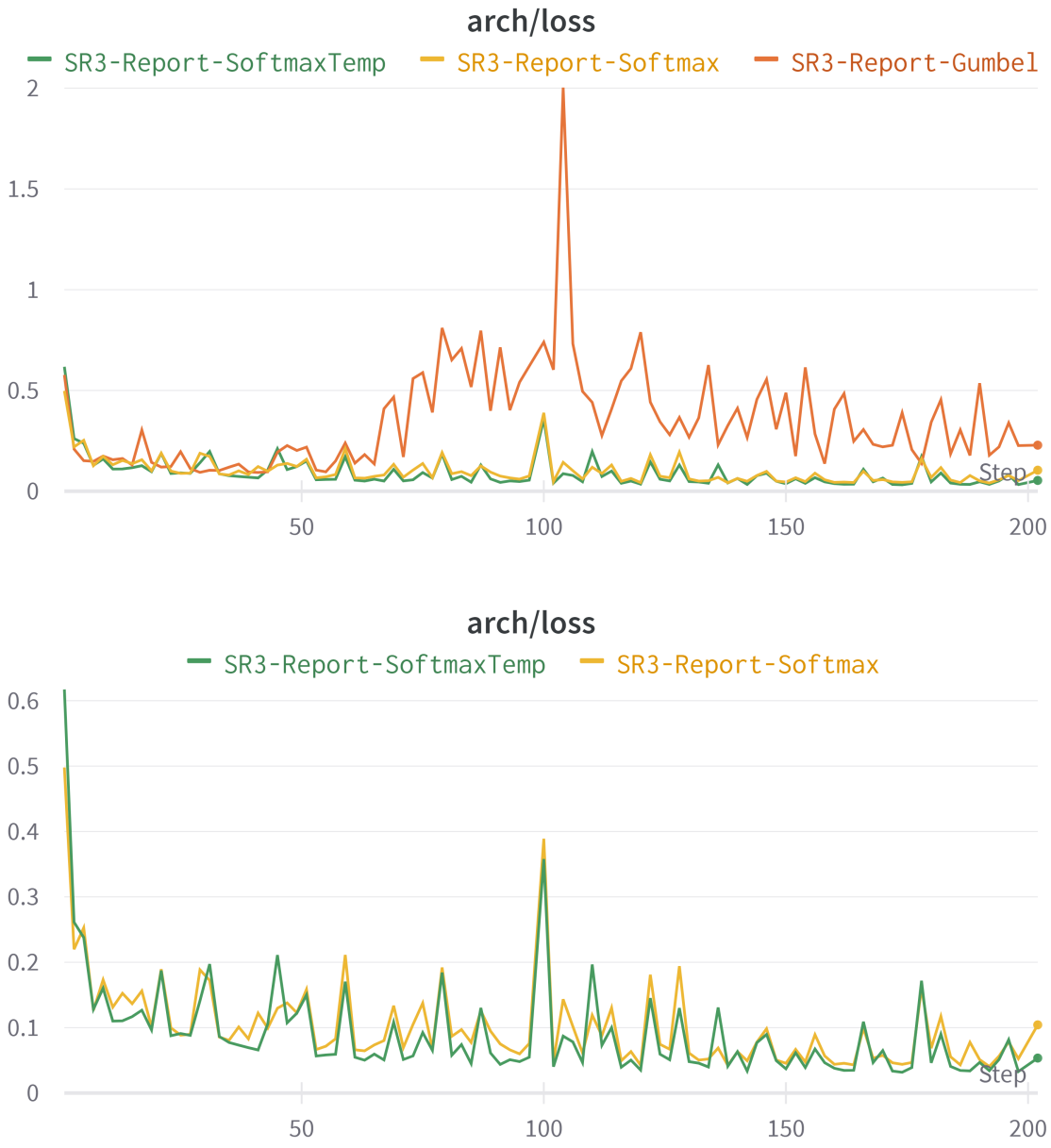


Figure 5.2: Architechure training loss for all NAS models (above) and only Softmax type models (below). This loss includes the scaled FLOPs and latency costs.

Figure 5.2 again shows a similar trend in the architecture parameter training where Gumbel-Softmax causes the architecture training loss to increase as the softmax temperature decreases and then hover an order of magnitude above Simple softmax and Softmax with temperature. Softmax and Softmax with temperature again show similar performance.

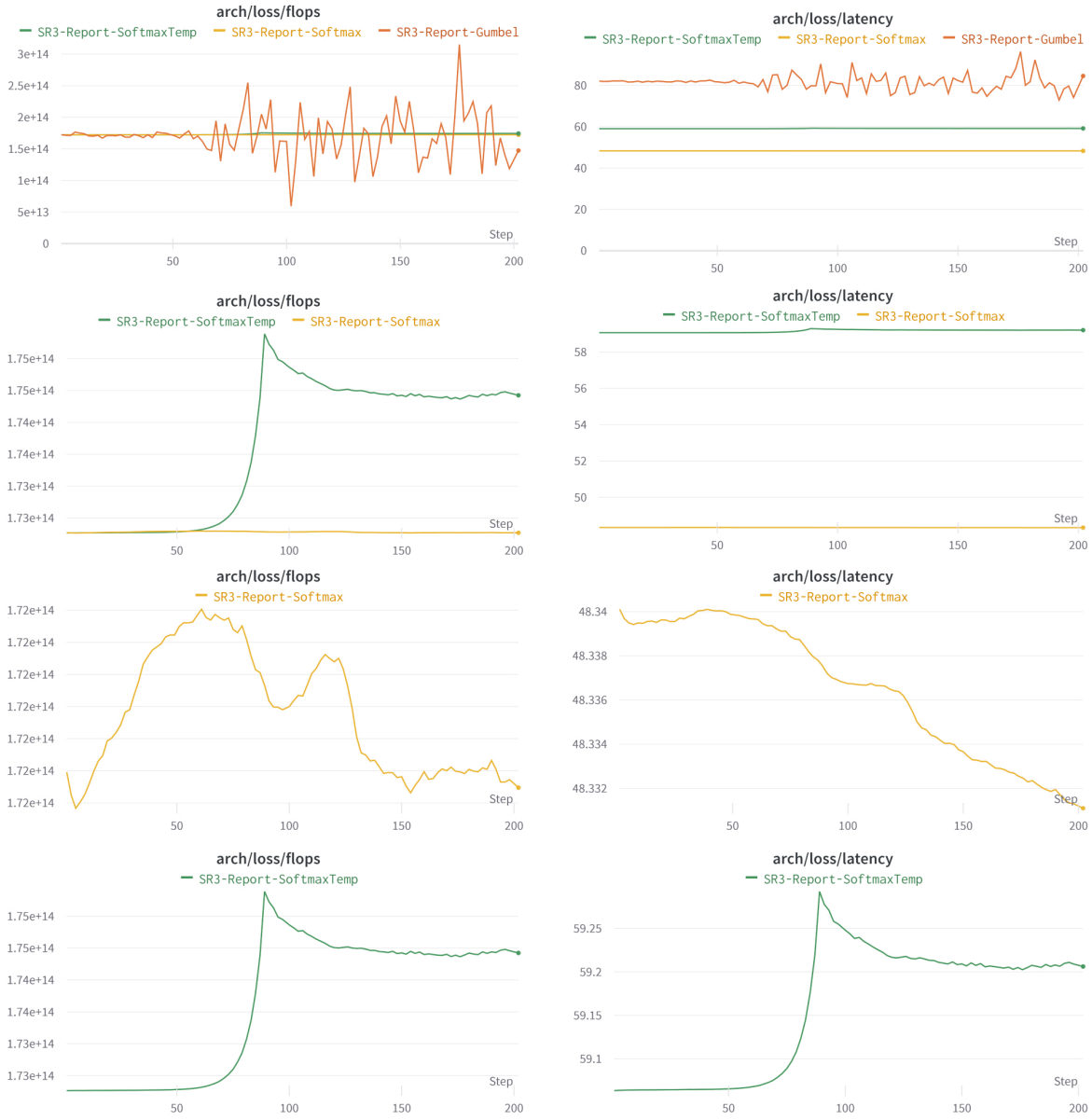


Figure 5.3: FLOPs and latency metrics of the supernet after being propagated through the model and scaled by all architecture parameters to be added to the loss. These values are not the FLOPs/latency of the sub-network.

Figure 5.3 shows different scales of the FLOPs and latency values obtained after they are propagated through the supernet. These values are not the FLOPs/latency of the sub-network but are obtained by propagating these values through and scaling them using the architecture parameters during the propagation. This allows training of architecture parameters when these FLOPs/latency values are back-propagated. These values are scaled using the hparams in table 4.1 and are then added to the L1 architecture trainer's loss before back-propagation.

In figure 5.3 we see that Gumbel-Softmax becomes more random at selecting operations as the temperature drops and never manages to get the latency metric near other two softmax functionns. We also

see that Simple softmax maintains the lowest FLOPs and latency loss values throughout architecture training and Softmax with temperature makes the FLOPs/latency optimisation quickly stagnate as the temperature drops. This shows that Simple softmax performs well in optimising both the FLOPs and the latency loss as soon as the total L1 loss reaches the same scale as the scaled FLOPs/latency.

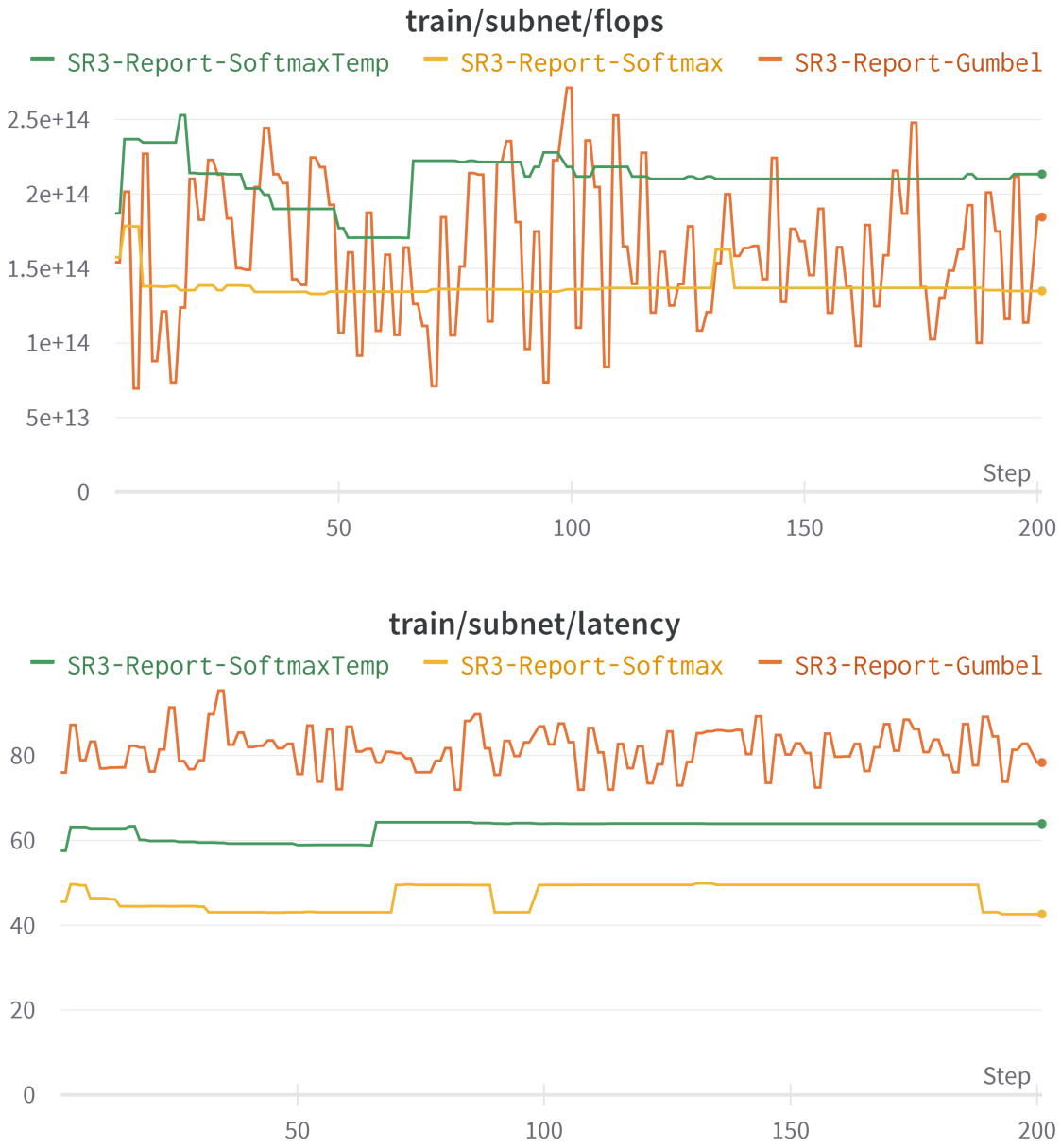


Figure 5.4: FLOPs (above) and latency (below) metrics of the sub-network during architecture training.

Figure 5.4 shows the live FLOPs/latency of the best sub-network during architecture training. We see a similar pattern with Gumbel-Softmax showing random selection of operations while both Simple softmax and Softmax with temperature show much more stability in keeping the sub-network FLOPs/latency

metrics low. Again Simple softmax performs the best while maintaining lowest latency and a consistently low FLOPs cost throughout architecture training.

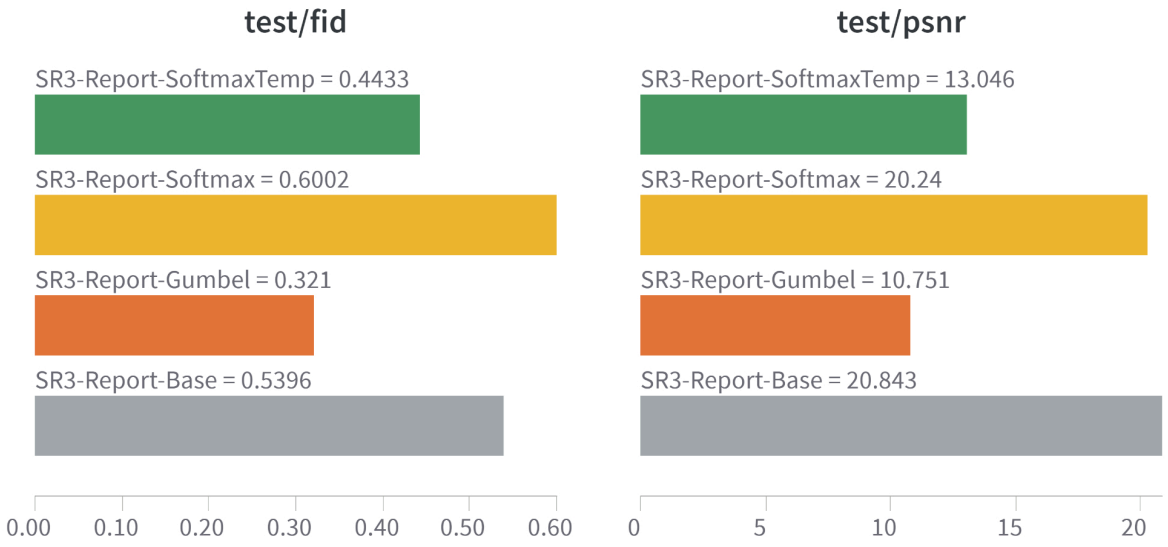


Figure 5.5: FID and PSNR metrics on the test dataset for all models

Figure 5.5 shows that the simple Softmax model has the worst FID score but the 2nd best PSNR. This shows conflicting measurements and more can be see by looking at the actual image samples generated by the models.

Table 5.1: NAS Model comparison

Model	NAS-Softmax	Terra FLOPs ↓	Latency ↓	FID ↓	PSNR ↑	No. params ↓
Base SR3	-	140.32	31.23	0.54	20.84	155,147,779
NAS Supernet SR3	-	841.44	326.11	-	-	475,298,249
NAS Subnet SR3	Simple Softmax	134.92	42.62	0.60	20.24	133,100,803
NAS Subnet SR3	Temperature Softmax	213.39	63.91	0.44	13.41	147,061,507
NAS Subnet SR3	Gumbel-Softmax	184.62	78.35	0.32	10.75	110,851,843

Table 5.1 shows a summary of the FLOPs, latency, FID, PSNR and parameter count of all the models that are generated using NAS. It also shows these values for the base model inspired by [4] and the FLOPs, latency and parameter count of the supernet that is used to perform DARTS [1] like NAS. In this table, green highlights the best value for a metric and red highlights the worst. We see that NAS with Simple softmax outperforms the other two NAS approaches in optimizing the FLOPs and latency, while it sits in between the other two approaches for parameter count. We even see that Gumbel-Softmax shows the best FID score contradicting the images in Fig. 5.9. Overall we notice that PSNR corresponds to a better representation of the quality of images generated by any model.

5.1 Test images

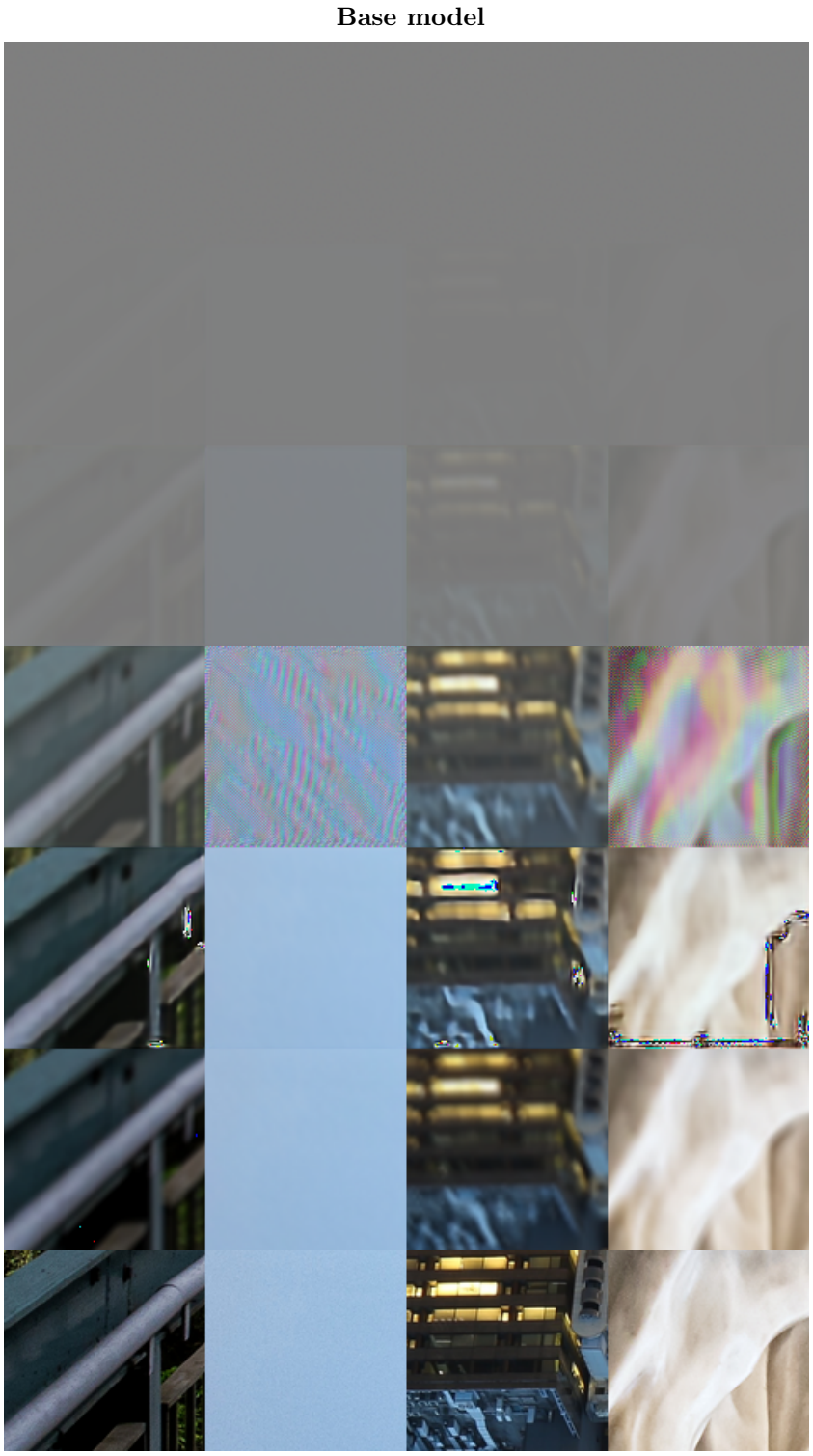


Figure 5.6: Image generated by the Base model. Last row = target hr image, Second last row = input lr image, Third last row = Final model output image

Softmax with temperature model

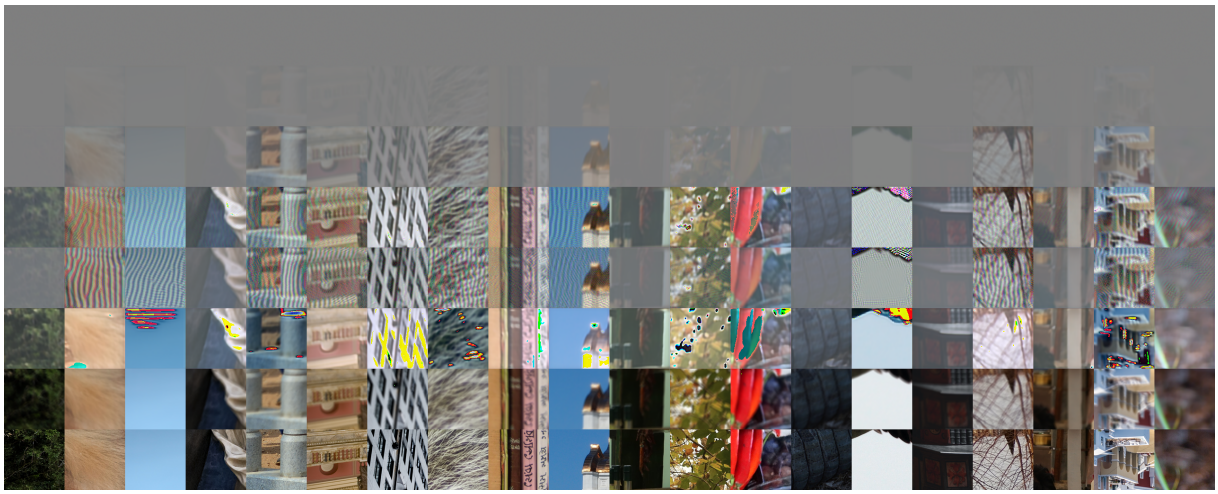


Figure 5.7: Image generated by the Softmax with temperature model. Last row = target hr image, Second last row = input lr image, Third last row = Final model output image

Simple Softmax model

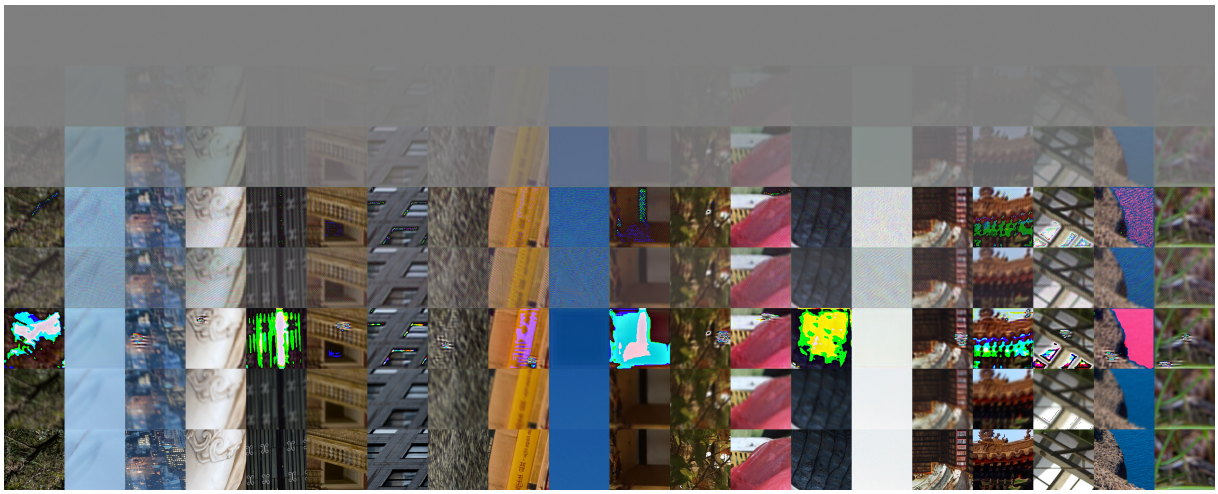


Figure 5.8: Image generated by the simple Softmax model. Last row = target hr image, Second last row = input lr image, Third last row = Final model output image

Gumbel-Softmax model

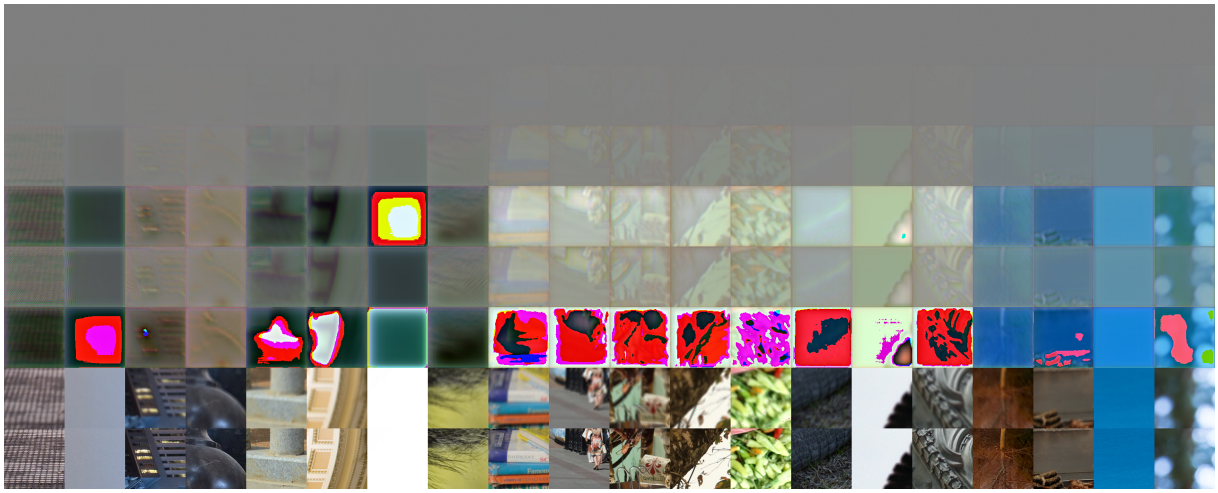


Figure 5.9: Image generated by the Gumbel-Softmax model. Last row = target hr image, Second last row = input lr image, Third last row = Final model output image

6 Summary and Future plan of work

This work has applied Neural Architecture Search (NAS) over denoising diffusion [3] models and has explored the impact of different softmax function variants used on the architecture parameters of a supernet that is trained using a DARTS [1] like approach, while also optimising for reduced FLOPs and latency inspired by [2].

The model search is highly relevant to resource saving since the model ϵ_θ is deployed in a denoising diffusion [3] system where the model ϵ_θ is run to a large number of iterations for each input sample/batch. This means that even a small saving in computational cost is scaled up by the number of time steps T in the diffusion process.

The diffusion process also affects the image quality greatly by making the generated image very sensitive to even the smallest imperfections in the ϵ_θ model's are magnified during the reverse diffusion process. This can be seen in Fig. 5.9 and Fig. 5.8 where some images are slightly visible in the earlier steps of reverse diffusion but the final output over-shoots and clips pixel values.

This work follows [4] and uses a UNet model as the base model for NAS, possible future work in this area can be exploring other model architectures.

References

[1] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," 2019.

[2] Y. Fu, W. Chen, H. Wang, H. Li, Y. Lin, and Z. Wang, "Autogan-distiller: Searching to compress generative adversarial networks," 2020.

[3] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *CoRR*, vol. abs/2006.11239, 2020. [Online]. Available: https://arxiv.org/abs/2006.11239

[4] C. Saharia, J. Ho, W. Chan, T. Salimans, D. J. Fleet, and M. Norouzi, "Image super-resolution via iterative refinement," 2021. [Online]. Available: https://arxiv.org/abs/2104.07636

[5] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2017.

[6] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," 2020.

[7] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," 2019.

[8] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," 2018.

[9] E. Liberis, Łukasz Dudziak, and N. D. Lane, "μnas: Constrained neural architecture search for microcontrollers," 2020.

[10] Y. Lin, M. Yang, and S. Han, "Naas: Neural accelerator architecture search," 2021.

[11] W. Chen, Y. Wang, G. Lin, C. Gao, C. Liu, and L. Zhang, "Chanas: coordinated search for network architecture and scheduling policy," *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2021.

[12] X. Chen, L. Xie, J. Wu, and Q. Tian, "Progressive differentiable architecture search: Bridging the depth gap between search and evaluation," 2019.

[13] Y. Zhao, L. Wang, Y. Tian, R. Fonseca, and T. Guo, "Few-shot neural architecture search," 2021.

[14] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," 2020.

[15] J. Sohl-Dickstein, E. A. Weiss, N. Maheswaranathan, and S. Ganguli, "Deep unsupervised learning using nonequilibrium thermodynamics," 2015. [Online]. Available: https://arxiv.org/abs/1503.03585

[16] L. Weng, "What are diffusion models?" *lilianweng.github.io*, 2021. [Online]. Available: https://lilianweng.github.io/posts/2021-07-11-diffusion-models/

[17] E. Agustsson and R. Timofte, "Ntire 2017 challenge on single image super-resolution: Dataset and study," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[18] A. Ignatov, R. Timofte *et al.*, "Pirm challenge on perceptual image enhancement on smartphones: report," in *European Conference on Computer Vision (ECCV) Workshops*, January 2019.

[19] R. Timofte, E. Agustsson, L. Van Gool, M.-H. Yang, L. Zhang, B. Lim *et al.*, "Ntire 2017 challenge on single image super-resolution: Methods and results," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[20] R. Timofte, S. Gu, J. Wu, L. Van Gool, L. Zhang, M.-H. Yang, M. Haris *et al.*, "Ntire 2018 challenge on single image super-resolution: Methods and results," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[21] R. Wang, M. Cheng, X. Chen, X. Tang, and C.-J. Hsieh, "Rethinking architecture selection in differentiable NAS," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=PKubaeJkw3

[22] L. Yao, R. Pi, H. Xu, W. Zhang, Z. Li, and T. Zhang, "Joint-detnas: Upgrade your detector with nas, pruning and dynamic distillation," 2021.

[23] N. Chen, Y. Zhang, H. Zen, R. J. Weiss, M. Norouzi, and W. Chan, "Wavegrad: Estimating gradients for waveform generation," 2020. [Online]. Available: https://arxiv.org/abs/2009.00713

[24]

[25] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3505–3506. [Online]. Available: https://doi.org/10.1145/3394486.3406703

[26] A. Zela, T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, and F. Hutter, "Understanding and robustifying differentiable architecture search," 2020.

[27] C. Li, J. Peng, L. Yuan, G. Wang, X. Liang, L. Lin, and X. Chang, "Blockwisely supervised neural architecture search with knowledge distillation," 2020.

[28] S. Belousov, "Mobilestylegan: A lightweight convolutional neural network for high-fidelity image synthesis," 2021.

[29] M. S. Abdelfattah, Łukasz Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane, "Best of both worlds: Automl codesign of a cnn and its hardware accelerator," 2020.

[30] N. Nayman, Y. Aflalo, A. Noy, and L. Zelnik-Manor, "Hardcore-nas: Hard constrained differentiable neural architecture search," in *ICML*, 2021.

[31] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," 2017.

[32] A. Nonaka, A. S. Almgren, J. B. Bell, M. J. Lijewski, C. M. Malone, and M. Zingale, "MAESTRO: An Adaptive Low Mach Number Hydrodynamics Algorithm for Stellar Flows," , vol. 188, pp. 358–383, Jun. 2010.

[33] L. Biewald, "Experiment tracking with weights and biases," 2020, software available from wandb.com. [Online]. Available: https://www.wandb.com/