Adaptive Generative Modeling in Resource-Constrained Environments

Jung-Eun Kim
Computer Science
Yale University
New Haven, CT, USA
jung-eun.kim@yale.edu

Richard Bradford
Commercial Avionics Engineering
Collins Aerospace
Cedar Rapids, IA, USA
richard.bradford@collins.com

Max Del Giudice
Computer Science
Yale University
New Haven, CT, USA
max.delgiudice@yale.edu

Zhong Shao
Computer Science
Yale University
New Haven, CT, USA
zhong.shao@yale.edu

***Abstract*—Modern generative techniques, deriving realistic data from incomplete or noisy inputs, require massive computation for rigorous results. These limitations hinder generative techniques from being incorporated in systems in resource-constrained environment, thus motivating methods that grant users control over the time-quality trade-offs for a reasonable "payoff" of execution cost. Hence, as a new paradigm for adaptively organizing and employing recurrent networks, we propose an architectural design for generative modeling achieving flexible quality. We boost the overall efficiency by introducing non-recurrent layers into stacked recurrent architectures. Accordingly, we design the architecture with *no redundant* recurrent cells so we avoid unnecessary overhead.**

I. Introduction

In a machine learning module, processing of data cannot be completed within strict time limits in an "all-or-nothing" manner. Furthermore, for the last decade or so, deep learning has been advancing its performance by increasing neural network depth and complexity, but it is neither straightforwardly explainable nor predictable how an extensively fabricated network can impact quality of the overall result. In particular, to be deployed in a *resource-constrained* environment, machine learning modules could allow trade-offs between time and quality of results that could be controlled depending on the application context. Hence, in this paper, we address time-quality trade-offs for *generative models*, which have proven useful for applications for deriving realistic forms of data from abstract ones, *e.g.*, enhancing resolution of pictures (from lower resolution to higher resolution, called upsampling or super-resolution), removing occlusions from pictures, generating an estimated picture with certain designated features, explaining a picture (scene) with texts to describe it, and so on. However, obtaining high-quality results from a generative model imposes a huge computational burden due to the complexity of the numbers of internal parameters required in the network layers.

Techniques present in the existing literature do not grant users flexible control over the time-quality trade-offs. In this paper, we propose novel techniques for increasing the prediction granularity and efficiency of models, which can be generally applied in a recurrent neural network (RNN), an architectural platform for generative modeling. Through our architecture, if time is short, quality is sacrificed. In these contexts, a "sufficient" quality depends on the application, data set and user. More emphasis is placed on the flexibility of intervals between obtaining intermediate results. Intervals between earlier results are short whereas later intervals are long, meaning our framework provides a "quick glance" at a solution first, and rigorous details emerge later. We name this new paradigm as *Adaptive Generative Modeling*.

Our approach is to use a recurrent neural network structure to break down this complexity into smaller operations, so as to facilitate flexibility in obtaining intermediate results. We vary the execution time per iteration so as to expend processing resources when the current candidate solution is likely to be in the neighborhood of the optimum. To decrease the processing time for a given active layer, our technique switches recurrent convolution layers into standard (non-recurrent) convolution layers unless it degrades the overall performance. The resources used by a standard convolutional layer are necessarily less since no manipulations involving state need to be performed. We find that in certain conditions, eliminating some recurrent layers results in no necessary performance impact.

This work is supported in part by NSF grants 1945541, 1763399, 1521523, and GPU Grant by NVIDIA Corporation. All views expressed here are those of the authors and not necessarily those of sponsors.

II. Adaptive Generative Modeling

As opposed to discriminative modeling (*e.g.*, image classification), generative modeling attempts to learn the shape of the underlying (unknown) data distribution. Assuming training is successful, this effectively makes generative models capable of "generating" data from the unknown distribution. In this paper, our target application is *super-resolution* which is a process to produce a high-resolution image from a low-resolution image. That is, its 'data-generation' process involves generating additional pixels to increase the resolution of an input image, and the unknown data distribution we want to learn is the high resolution image space.

The aim of building of our framework is to provide intermediate computational results when interrupted. It is designed so that these intermediate results improve over time for adaptive timeliness. Thus, if only a small amount of time is available, the system can sacrifice quality of the result, and vice versa. Note that in this context, time is concerned only with the generating (*i.e.*, inferring) stage; offline training is assumed to be done for a sufficient amount of time.

A. Underlying Architecture

We introduce our recurrent super-resolution generative model. The model is trained to take in a low-resolution input image of dimensions $L \times L \times 3$, and output a high-resolution output image of dimensions $H \times H \times 3$, where $H > L$. Hereafter, we refer to the inputs and outputs of networks (and the intermediate calculations) as *tensors*, multidimensional arrays. The training dataset consists of pairs of low-resolution images and their high-resolution counterparts (x, y). The training process for super-resolution proceeds as follows:

1) Feed a mini-batch of inputs X (subset of our training dataset) through the generator.
2) The generator computes the super-resolution outputs, O. We can treat the actual generator as a black box for the present. Details will be given in the following sections.
3) Compute the distance (or loss, see Sec. II-E for details) between the outputs O and the actual high-resolution images Y.
4) Use backpropagation to update the weights.
5) Return to step (1).

Our framework is based on *Generative Adversarial Networks* (GANs) [8]. A GAN consists of two neural networks competing against each other. Let $\mathbf{D}$ and $\mathbf{G}$ be a *discriminator* (critic, interchangeably) and *generator* respectively. They are functions defined over the domains $\mathbf{D} : P_r \longrightarrow [0, 1]$ and $\mathbf{G} : P_g \longrightarrow P_r$, where P_r represents our target data distribution, and P_g represents the noisy input to the generator. In super-resolution, P_g corresponds to low resolution images, and P_r is the high resolution counterparts. The discriminator $\mathbf{D}$ outputs a value close to 1 if it determines the image to be realistic and close to 0 otherwise.

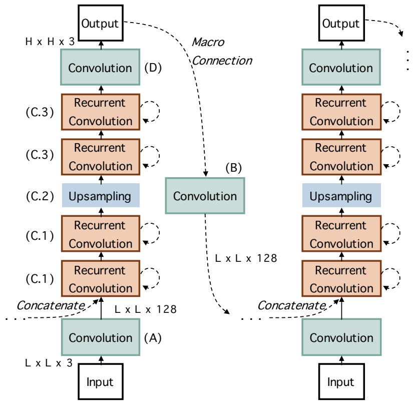


Fig. 1: Underlying architecture.

The specific generator architecture is shown in Fig. 1. A single forward pass shares a similar base structure to the one introduced in [20]. By choosing a simpler base model we can focus on the two architectural techniques, *AEL* and *ORL*. Our techniques are general and can be applied to most stacked recurrent architectures.

The generator network consists of an initial convolutional layer (labeled as **A**). The features from the output of the previous iteration (**B**) are concatenated to create a tensor to pass to the next portion of the network. The inner four layers are convolutional LSTMs. The first two convolutional LSTMs (**C.1**) operate in the $L \times L$-dimensional image space, and are split by a static upsampling layer (**C.2**), which simply transforms the tensor dimensions into the high-resolution image space. The next two convolutional LSTMs (**C.3**) learn image features in this $H \times H$ image space. Finally, one more convolution (**D**) transforms our tensor into the correct 3-dimensional (RGB) feature space.

B. Adaptively Expanding Layers (AEL)

Considering a general stacked recurrent network optimized to produce iterative predictions, we encounter a trade-off: if the network is too deep (referring to layer count), one pass through the network is computationally expensive. If the network is too shallow, the learning capacity is limited. When trying to build a model that can produce

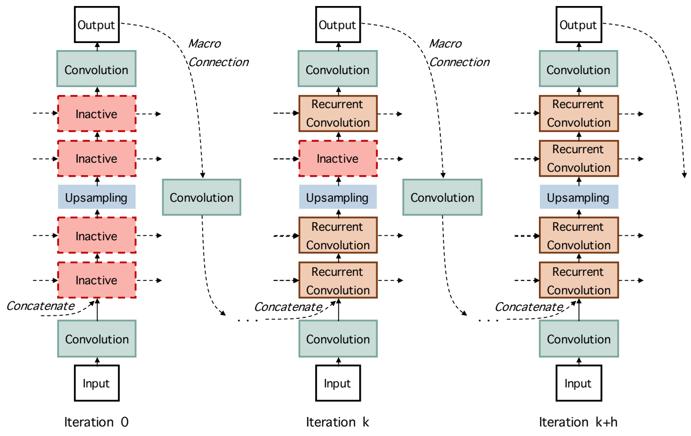


Fig. 2: Overview of AEL. The three recurrent convolutions become active at iteration k, and the full stacked network becomes active at iteration $k + h$. An inactive layer is simply an identity operation, *i.e.*, the output is same as the input. An iteration with *more inactive layers* will yield a *quicker pass*, and vice versa.

adaptive predictions, we ideally want to maintain the benefits of both architectures. To achieve this, we make the *Time Per Iteration* (TPI) for early iterations fast, while later iterations can expend more time for the payoff of increased quality of output. To do so, certain layers are designated as *inactive* for earlier iterations (effectively making them *identity* operations), making one full forward pass take less time. In subsequent iterations, the inactive layers become active, further refining the result at the tradeoff of slower inference speed. This technique is referred to as *Adaptively Expanding Layers (AEL)*.

Variable TPI can be considered through the lens of human visual understanding. For example, if someone were to glance at a car for a split second, they may only notice broad features, *e.g.*, the car is a red truck. Given more time, they process additional information – the make and model, what is in the car, the condition of the tires, etc. The earlier iterations of this model can be thought of as conducting a quick glance, whereas the later ones perform a more thorough inspection.

To achieve this level of flexibility, we propose a recurrent generator with AEL as shown in Fig. 2. This network behaves like the model in Fig. 1, but each iteration has specified active and inactive layers. With this structure, predictions at earlier timesteps sacrifice accuracy for speed because earlier iterations only traverse part of the network. In the subsequent iterations, additional layers become active to improve the output quality. The capacity of the network to produce more detailed results increases with time. Our network is flexible in that it can iterate indefinitely. In practice there will be a number of iterations, contingent upon the task at hand and the particularities of the architecture, where improvement of the result levels off.

While AEL is a general design technique, whether it can be employed depends on certain architectural restrictions. Most significantly, the input tensors and output tensors of skipped layers (i.e., inactive layers) must align in a sensible manner. Given a four layer architecture where the first several iterations skip the inner two layers, the dimensions of the output of Layer 2 must match the dimensions of the input of Layer 4 (see Fig. 1). Outside of the immediate architectural restrictions, AEL also introduces two key design considerations: (i) which layers should be deactivated? (ii) at which point should inactive layers be activated? To partially answer the second one, let us consider the state of a recurrent convolution cell at time t: $z_t = U * x_t + W * h_{t-1} + b$, where x_t is the input at time t and h_{t-1} is the state from the previous timestep. Through time, the receptive field of the recurrent convolution cell expands. Once this field expands beyond the size of the original input, more iterations (without the addition of new layers) would result in no improvement.

C. Optional Recurrent Layers (ORL)

AEL increases the flexibility of the model in that individual iterations can have variable processing times, but we also want to consider global changes, *e.g.*, changes that affect the inference speed of the entire system. In other words, how can we decrease TPI for every iteration? Our approach is to decrease the number of *unnecessary* recurrent layers in the forward pass. A recurrent layer must perform multiple convolutions, involving both state from the previous timestep and input data, in order to produce a result. A non-recurrent convolutional layer necessarily uses fewer resources since no manipulations involving state need to be performed. Fig. 3 shows example versions of possible configurations. The naming convention of ORL configuration uses `r` to indicate a recurrent layer and `n` to indicate a non-recurrent (*i.e.,* standard, feed-forward) layer. `r` or `n` is labeled for the inner four layers from the bottom.

We chose convolutional LSTMs [21] for the recurrent convolution layer due to their ubiquity and success in resolving the vanishing gradient problem. Then, the four inner layers may have either convolutional LSTM or non-recurrent convolution layer. The convolutional LSTM uses the standard LSTM gates and state update rules, but the critical operation is convolution rather than matrix multiplication:

$$
\begin{aligned}
i_t &= \sigma(W_{xi} * x_t + W_{hi} * h_{t-1} + W_{ci} \circ c_{t-1} + b_i) \\
f_t &= \sigma(W_{xf} * x_t + W_{hf} * h_{t-1} + W_{cf} \circ c_{t-1} + b_f) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tanh(W_{xc} * x_t + W_{hc} * h_{t-1} + b_c) \\
o_t &= \sigma(W_{xo} * x_t + W_{ho} * h_{t-1} + W_{co} \circ c_t + b_o) \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}
$$

where W_* and b_* values are learnable parameters, $*$ denotes convolution, $\circ$ denotes Hadamard product, and σ denotes the sigmoid function. h_t and o_t denote the cell state and output, respectively, at time t. Non-recurrent convolution layer is described as $u = \mathrm{BN}(\ell(C_1 * x_t + a_1))$ and $o_t = \mathrm{BN}(C_2 * u + a_2)$, where BN is batch normalization, $*$ is convolution, and ℓ is a leaky rectified linear unit (LReLU).

D. Combining AEL and ORL

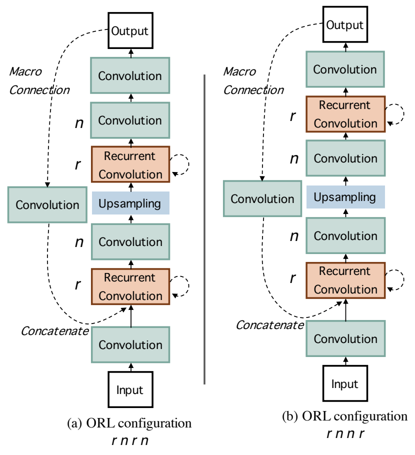


Fig. 3: Two configurations of ORL mechanism: `r` stands for "recurrent" whereas `n` stands for "non-recurrent". It is labeled from the bottom for the inner four layers.

The combination of AEL and ORL induces four possible configurations for an inner layer: (i) active convolutional LSTM, (ii) inactive convolutional LSTM, (iii) active non-recurrent convolution, and (iv) inactive non-recurrent convolution. Combining ORL and AEL introduces an important question: will a network with an ORL architecture perform the same when an AEL strategy is placed on top? In other words, given all 16 ORL configurations for our model trained on a dataset and ordered by test performance, will this order remain the same when an AEL strategy is placed on top of each of the models? Or will AEL fundamentally alter the performance of the model, resulting in a totally new ordering of models.

We systematically examine the effects of toggling layers in the recurrent context, and the effects of including recurrent/non-recurrent layers. Our work uses a super-resolution GAN to explore these enhancements but not the particular problem area of super-resolution.

E. Loss Function

Our model uses Wasserstein GAN (WGAN), introduced in [1]. It uses the following loss functions:

$$
\begin{aligned}
L_D &= -\mathbf{E}_{x\sim P_r}[\mathbf{D}(x)] - \mathbf{E}_{x\sim P_g}[\mathbf{D}(\mathbf{G}(x))]; \\
L_G &= -\mathbf{E}_{x\sim P_g}[\mathbf{D}(\mathbf{G}(x))].
\end{aligned}
$$

In this case, the discriminator $\mathbf{D}$ now maps to the range $[0, 1]$ – rather than returning 1 or 0, $\mathbf{D}$ can be thought of as a critic judging the input quality. In addition, we compare the output super-resolution image to the true image. This results in an additional *visual loss* term added to our generator loss:

$$L_G = L_{G_{GAN}} + L_{G_{VIS}} = -\mathbf{E}_{x\sim P_g}[D(G(x))] + L_{G_{VIS}}$$

The simplest definition of $L_{G_{VIS}}$ is mean squared error (MSE). Given two $m \times n$ images X and Y, where $X(i, j)$ represents the value of the pixel at row i and column j, MSE is defined as

$$MSE(X, Y) = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [X(i, j) - Y(i, j)]^2$$

This performs a pixel-by-pixel comparison of the super-resolution image and ground truth image. Because the generator outputs a prediction at each timestep, we compute the MSE for each iteration's generated prediction. Suppose we train a network for T timesteps. Let X_t be the predicted (super-resolved) image at time step t, and let Y be the label image (ground truth). Our visual loss is defined as

$$L_{G_{VIS}} = \frac{1}{T} \sum_{t=1}^{T} MSE(X_t, Y)$$

This loss term is added to the adversarial loss for the generator:

$$L_G = L_{G_{VIS}} - 10^{-3} * L_{G_{GAN}}; \qquad L_D = L_{D_{GAN}}$$

The generator GAN loss is weighted by 10^{-3}, as in [16] – in experimentation, we chose this weight to balance training stability with generator convergence.

III. Experiments

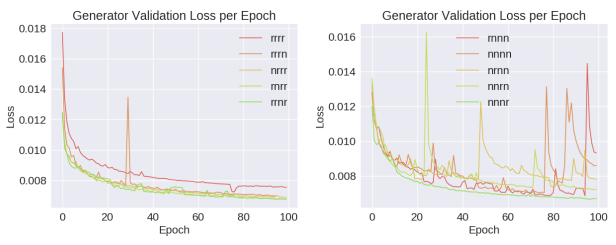
A. Experimental Setup

For the ORL architecture comparison, we train each of the possible 16 models for 100 epochs. All datasets consist of $64 \times 64 \times 3$ RGB images scaled to $[0, 1]$. These images are downsampled by a factor of 2 using bicubic interpolation to create the $32 \times 32 \times 3$ inputs to the network. Hyperparameters and other specifications are as follows:

- **Optimizer**: We use AdamOptimizer [12] for both the critic (discriminator) and generator. Both share the following parameters:
  - **Learning rate**: 10^{-4}. Once we find the correct direction to move the weights by computing the gradient of the loss function, the learning rate specifies the magnitude of the step we take in that direction. A high learning rate results in faster training with a higher risk of divergence, while a lower learning rate results in smoother training at the expense of speed.
  - **Epsilon**: 10^{-5}. Similar to the learning rate, ε is a stability constant affecting the smoothness of the training procedure. A smaller value of ε results in quicker training at the expense of stability, while a lower value of ε stabilizes training but with slower convergence. This value was chosen because certain architectures were highly volatile during training. Other architectures were relatively stable due to a higher ε value, but we kept $\varepsilon = 10^{-5}$ across all models for the sake of comparison.
- **Number of epochs**: 100 • **Mini-batch size**: 16
- **Gradient clipping**: Enabled with a threshold of 0.5 - important in preventing gradient explosion when training recurrent networks.
- **Hardware**: All networks were trained either on an NVIDIA Tesla K80, Tesla P100, or an Quadro P6000. The model quality is independent from where the architectures are trained.
- **Iteration count**: All networks are trained for 100 iterations.

1) Data Sets: We use three datasets to assess the impact of data complexity/homogeneity on model performance.

- CarData: The Stanford Cars Dataset [13] consists of 16,185 RGB images of 196 classes of cars. The data is preprocessed by cropping around the cars, omitting most non-car background pixels. Each image is downsampled to $64 \times 64 \times 3$ using bicubic interpolation.
- FaceData: The CyberExtruder Ultimate Face Matching Data Set [7] contains consists of 10,205 RGB images of 1000 faces scraped



(a) # of recurrent layers = 3. `rrrr` is shown for comparison. (b) # of recurrent layers = 1. `nnnn` is shown for comparison.

Fig. 4: Model trained against full CarData

from the internet. The dataset is highly varied, containing differences in light, shadow, expression, facial occlusions, etc. Each image is downsampled to $64 \times 64 \times 3$ using bicubic interpolation.

- GeoData: The Sat-4 Airborne Dataset [3] consists of 500,000 $28 \times 28 \times 4$ image patches covering four land types: barren land, trees, grassland, and a miscellaneous class. We created a sub-dataset of size $16,000$ by randomly selecting 64,000 image patches, grouping sets of four together in a single 56×56 image (2×2 grid), and then upsampling to 64×64 using bicubic interpolation. The original data has four color channels, but the infrared channel is omitted.

2) Performance Measures: In addition to the MSE defined in Section II-E, we use two other performance metrics:

- Peak signal-to-noise ratio (PSNR): PSNR is defined in terms of mean squared error,

$$PSNR(X,Y) = 10 \cdot \log\Big(\frac{MAX_X^2}{MSE(X,Y)}\Big),$$

where X is the ground-truth image and Y is the super-resolved image. PSNR is functionally equivalent to a regularization of MSE. PSNR ranges from 0 to ∞ (in the case where MSE is 0, *i.e.*, the images are the same), with higher values being better.
- Structural similarity (SSIM): SSIM is a more advanced image similarity metric. It works by examining sub-windows of the images X and Y. For $n \times n$ windows x and y, the SSIM is defined as

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}.$$

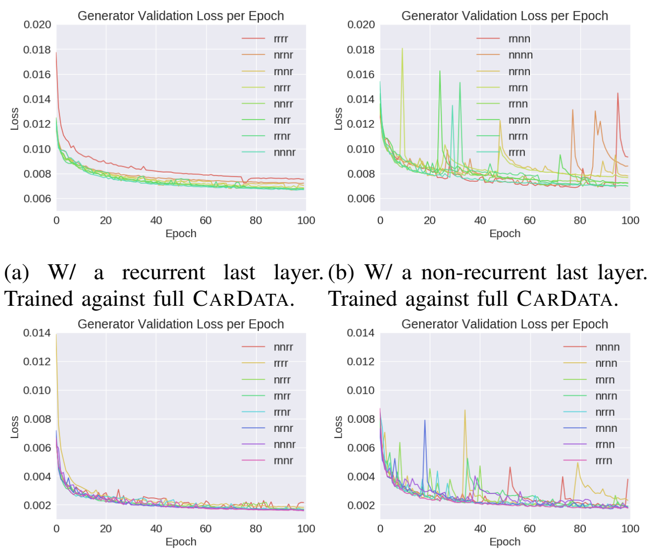
SSIM values for all subwindows are averaged. SSIM ranges from 0 to 1, with higher values being better.

To compute model performance, we use a labeled test set of images (low-resolution and high-resolution pairs). We feed the low-resolution images through the model and calculate the difference (using MSE, PSNR, or SSIM) between the output and the high-resolution label. In the following sections, *performance* or *test performance* refers to the MSE by default. Because our architecture is iterative and produces an output at each timestep, we average the MSE across all iterations.

B. Applying Optional Recurrent Layers (ORL)

1) Number of Convolutional LSTM Layers: Fig. 4(a) shows that removing a single recurrent layer has a *positive* impact on the final loss – `rrrr` had the largest loss. With the exception of `rrrn` which, although the final loss value is less than `rrrr`, suffers from unstable training, all architectures with one non-recurrent layer converge smoothly and more quickly than `rrrr`. The overall result shows that all recurrent layers is not necessarily the strongest.

We also examine the case with three non-recurrent layers, shown in Fig. 4(b). (`nnnn` is shown for a comparison purpose.) Here, all networks with the exception of `nnnr` have an `n` in the last layer, and all non-`nnnr` validation curves display the characteristic spikiness



(a) W/ a recurrent last layer. Trained against full CarData. (b) W/ a non-recurrent last layer. Trained against full CarData.

(c) W/ a non-recurrent last layer. Trained against full FaceData. (d) W/ a non-recurrent last layer. Trained against full FaceData.

Fig. 5: Impact of last recurrent layer on stability.

found in the other two cases. Moreover, the spikes increase with the total number of non-recurrent layers. Interestingly, `nnnr` is the top performing model among all 16 architectures. We can attribute this to the faster convergence of the model, as the non-recurrent `n` cells have fewer parameters than the `r` cells.

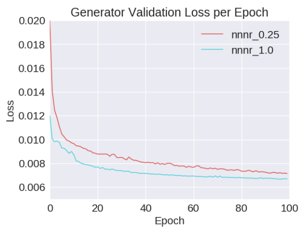
The improved performance of some models over `rrrr` architecture (i.e., fully-recurrent) suggests that, when building a stacked recurrent model, the designer should take into account whether all layers need to be recurrent. Indeed, the top performing model from this run of experiments was `nnnr` as explained above (Table I), which had 74.8% fewer parameters than the fully recurrent version. Aside from reducing inference time by introducing "non-recurrency", our work shows that one may expect performance boosts and improvements in training with a sensible number of recurrent layers.

TABLE I: Top-performing models w.r.t. active recurrent layers

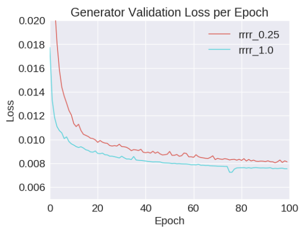
Model	# Parameter	MSE
`nnnr`	1335683	0.0068
`nnrr`	2220419	0.0070
`rrnr`	3105155	0.0069
`rrrr`	3989891	0.0075

2) Placement of recurrent/non-recurrent layers: Where the recurrent layers are placed within the stacked recurrent network (when there are the same number of recurrent layers) also is critical to the performance of the network. A significant result that draws attention is the impact of the last recurrent/non-recurrent layer on stability. Fig. 5 shows the impact of the existence of a recurrent last layer - models in Fig. 5(a) (for CarData) and Fig. 5(c) (for FaceData) have a recurrent last layer, and ones in Fig. 5(b) (for CarData) and Fig. 5(d) (for FaceData) have a non-recurrent last layer. (In Fig. 5(a) and Fig. 5(c) there is no model that is entirely stable like the ones in Fig. 5(b) or Fig. 5(b).) The final recurrent layer's ability to record state close to the output has a larger impact than earlier recurrent layers, likely due to its proximity to the final result.

3) Dataset Size and Type: The previous sections showed that more recurrent layers do not equate to a better performing model. This, however, is contingent on dataset size and complexity. Fig. 6 shows the impact of dataset size on loss value - the charts compare a case of full and $1/4$ of CarData. Although only select models are shown in Fig. 6 (for the purpose of clearly showing the gap), the trend



(a) On model nnnr.



(b) On model rrrr.

Fig. 6: Impact of dataset size. Trained against full and 1/4 of CarData

that small dataset sizes suffer a higher loss is consistent with every model by apple-to-apple comparison. However, with a smaller dataset, the model will likely converge earlier but with a much higher loss value. Comparing Fig. 6(a) and Fig. 6(b), when we take only 25% of the original dataset, models with more recurrent layers tend to learn significantly less quickly. Indeed, nnnr exceeds the performance of all other models on the reduced CarData set (which is also consistent with the cases of the original dataset). However, additional data is not a cure-all. More training data increases training time, and if the model is too simple (i.e., has too few trainable parameters) to capture the information conveyed by the additional data, then the result is a less-efficient training process with no performance benefit. Striking the balance of trainable parameters to dataset size is critical in determining the efficiency and effectiveness of model training.

Fig. 7 displays the best models (with respect to the number of recurrent layers) for each of the three datasets: CarData, FaceData, and GeoData. Notice that rrrr is the worst performing among all datasets. Indeed, for GeoData, we even experience training instability with rrrr, whereas the other models converge steadily.

C. Applying Adaptively Expanding Layers (AEL)

- BicubicUpsampling: The baseline deterministic algorithm for upsampling an image. All models should exceed this point.
- NoAEL: ORL models without AEL laid on top. We examine the following subset of models: rrrr, rrnr, nnrr, and nnnr.
- AEL+ORL: The ORL models listed above, with one of the three AEL configurations enumerated below applied on top.

Let $x \in \{\mathtt{r}, \mathtt{n}\}$ indicate an active recurrent or non-recurrent layer, and let "–" indicate an inactive layer. For example, r−−r would represent the outer two layers being active & recurrent and the inner two layers inactive. The notation r−−r (p) indicates that the given configuration repeats p times. We examine the following configurations:

1) **Config. A**: −−−x (2) — x−−x (2) — xx−x (3) — xxxx (3)
2) **Config. B**: x−−x (2) — xx−x (2) — xxxx (6)

We restrict our attention to CarData for the purposes of exploring the effects of AEL on model performance. Each model is tested on 1600 test images, and the scores and computation time are averaged.

Config. A: Fig. 8 shows the performance of AEL configuration 1 on the four ORL models above. In models (a) rrrr and (b) rrnr, we can see particularly near the start of the curves, that AEL is able to output predictions earlier than ORL for a reduced time cost. Indeed, for these ORL configurations, the first iteration of the AEL model finishes in roughly half the time as the ORL model. For time-sensitive applications that require a strict time constraint, this behavior is critical. The rrnr and rrrr AEL models do slightly underperform the NoAEL models. However, this stems from the added complexity of the AEL strategy requiring a longer training.

Note that the (c) nnrr and (d) nnnr AEL models underperform when compared to the other AEL models. Consider nnnr – our configuration starts as −−−x, meaning all non-recurrent layers are inactive and the one recurrent layer at the end is active. Indeed, this AEL configuration *only* cancels non-recurrent layers for the ORL layout nnnr. Recurrent layers have significantly more parameters than non-recurrent layers, hence the arithmetic operations take more time. Consequently, the time-savings yielded by only canceling non-recurrent layers are more marginal, making the cross markers shift rightward along the x-axis. A similar argument applies to nnrr.

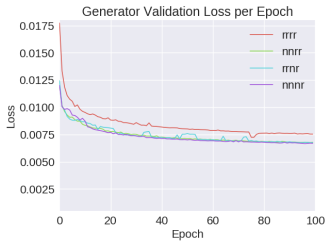
Config. B: Fig. 9 shows the performance of AEL configuration 2 on the four ORL models above. In all configurations, we are able to extract several earlier predictions than the NoAEL model, since more layers are active on the first iteration.

IV. Related Work

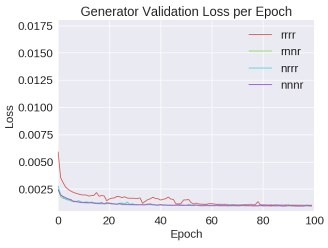
As a specific type of generative technique, GANs [8] have shown powerful performance on general-purpose modeling problems. Improvements on the original form were proposed in literature including [1] (WGAN) or [17] (SRGAN).

Although frequently used for sequential data, RNNs have proven useful where feed-forward networks are traditionally used. Recent architectures have combined RNN and CNN features, from simple LSTM implementations that incorporate convolutions [19] [10] [14], to complex networks implementing LSTMs [21], dilated convolutions [6], and quasi-recurrence [5]. Liang and Hu [19] designed recurrent convolutional layers (RCLs). The authors note that using recurrent connections within a convolutional layer broadens the receptive field of the convolution operation, making it possible for lower layers to recognize complex features. Based on [19], [10] developed a deeply-recursive convolutional network for super resolution.

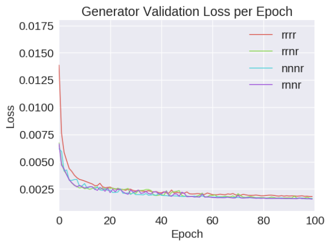
Originally developed with visual applications in mind [15], attention provides a way for a discriminative model to focus on parts of an input image that are most important to the final classification. Attention mechanisms have found widespread use in a number of neural network architectures [22], including RNNs [2], [23] and GANs [24]. While related to our goals, attention falls short of solving the problem of visual understanding. Implementing an attention mechanism also



(a) CarData

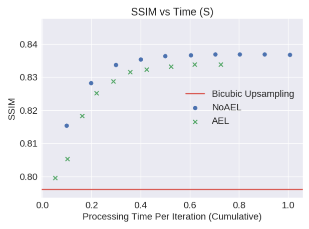


(b) GeoData

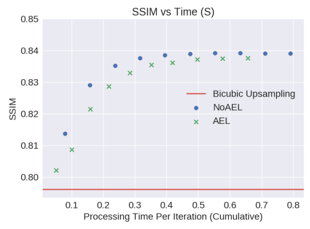


(c) FaceData

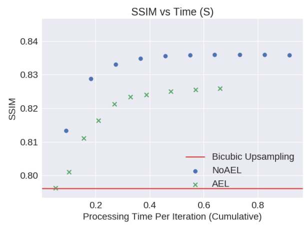
Fig. 7: Comparison of best models for each dataset. rrrr was not in any dataset's top models, but it is shown for comparison.



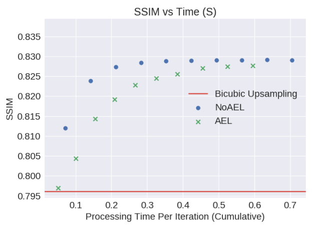
(a) `rrrr`



(b) `rrnr`

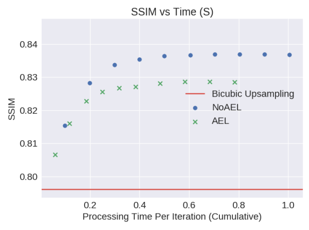


(c) `nnrr`

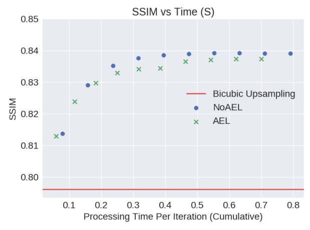


(d) `nnnr`

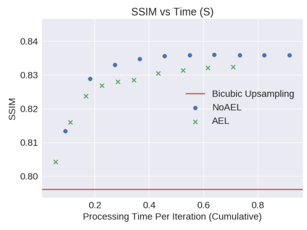
Fig. 8: Impact of AEL (Config. A) on ORL layouts.



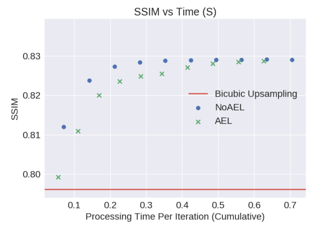
(a) `rrrr`



(b) `rrnr`



(c) `nnrr`



(d) `nnnr`

Fig. 9: Impact of AEL (Config. B) on ORL layouts.

involves introducing many more trainable parameters into a network. Other works have explored adaptive neural networks, including [4] and [11] for a resource-constrained environment. In addition, the idea of neural networks with early exits has been well-explored in the literature including [9], [18]. Many of these early-exit strategies were developed with the goal of improving computational efficiency, skipping layers automatically if the result exceeds certain confidence thresholds midway through the computation.

V. Conclusion

In this work, we proposed a new architectural framework employing a recurrent neural network, with the aim of granting users control over time-quality trade-offs: i) providing adaptive performance between iterations: fast early iterations, slow late iterations; ii) advancing efficiency by using non-redundant recurrent cells only. As the evaluation results demonstrate, decreasing the number of recurrent layers does not necessarily degrade the performance of an entire stacked recurrent network, while it can decrease the overhead of execution cost. To the best of our knowledge, this is a novel finding that enables the architecting of more efficient and adaptive networks.

References

[1] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein GAN. *CoRR*, abs/1701.07875, 2017.
[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.
[3] S. Basu, S. Ganguly, S. Mukhopadhyay, R. DiBiano, M. Karki, and R. R. Nemani. Deepsat - A learning framework for satellite imagery. *CoRR*, abs/1509.03602, 2015.
[4] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama. Adaptive neural networks for efficient inference. *CoRR*, abs/1702.07811, 2017.
[5] J. Bradbury, S. Merity, C. Xiong, and R. Socher. Quasi-recurrent neural networks. *CoRR*, abs/1611.01576, 2016.
[6] S. Chang, Y. Zhang, W. Han, M. Yu, X. Guo, W. Tan, X. Cui, M. J. Witbrock, M. Hasegawa-Johnson, and T. S. Huang. Dilated recurrent neural networks. *CoRR*, abs/1710.02224, 2017.
[7] I. CyberExtruder.com. Cyberextruder ultimate face matching data set. Accessed: 2018-07-01.
[8] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C ourville, and Y. Bengio. Generative adversarial nets. In *NIPS 27*. Curran Associates, Inc., 2014.
[9] A. Graves. Adaptive computation time for recurrent neural networks. *CoRR*, abs/1603.08983, 2016.
[10] J. Kim, J. K. Lee, and K. M. Lee. Deeply-recursive convolutional network for image super-resolution. In *CVPR*, 2016.
[11] J.-E. Kim, R. Bradford, M.-K. Yoon, and Z. Shao. ABC: Abstract prediction before concreteness. In *DATE '20*, page 1103–1108, 2020.
[12] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.
[13] J. Krause, M. Stark, J. Deng, and L. Fei-Fei. 3d object representations for fine-grained categorization. In *4th IEEE Workshop on 3dRR-13*, 2013.
[14] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent convolutional neural networks for text classification. In *AAAI*, 2015.
[15] H. Larochelle and G. E. Hinton. Learning to combine foveal glimpses with a third-order Boltzmann machine. In *Advances in Neural Information Processing Systems 23*. Curran Associates, Inc., 2010.
[16] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. P. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi. Photo-realistic single image super-resolution using a generative adversarial network. *CoRR*, abs/1609.04802, 2016.
[17] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. P. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi. Photo-realistic single image super-resolution using a generative adversarial network. *CoRR*, abs/1609.04802, 2016.
[18] S. Leroux, S. Bohez, E. Coninck, T. Verbelen, B. Vankeirsbilck, P. Simoens, and B. Dhoedt. The cascading neural network: Building the internet of smart things. *Knowl. Inf. Syst.*, 52(3):791–814, Sept. 2017.
[19] M. Liang and X. Hu. Recurrent convolutional neural network for object recognition. In *The IEEE CVPR*, June 2015.
[20] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.
[21] X. Shi, Z. Chen, H. Wang, D. Yeung, W. Wong, and W. Woo. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. *CoRR*, abs/1506.04214, 2015.
[22] K. Xu, J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, abs/1502.03044, 2015.
[23] Z. Yang, D. Yang, C. Dyer, X. He, A. J. Smola, and E. H. Hovy. Hierarchical attention networks for document classification. In *The Conference of the NAACL: Human Language Technologies*, 2016.
[24] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena. Self-attention generative adversarial networks, 2018.