Deep Image Inpainting

Charles Burlin cburlin@stanford.edu
Yoann Le Calonnec yoann@stanford.edu
Louis Duperier duperier@stanford.edu

Abstract

We present a new take on several image inpaiting techniques on small, simple images from CIFAR10. We improved context encoders by implementing several major training tricks on GAN as well as adapting the network to WGAN. We also compare encoders and discriminators based on existing state-of-the-art models against basic CNN architectures. In parallel, we worked on density-based methods and implemented Pixel CNN, Diagonal BiLSTM and Row LSTM. We propose a variation on the first, and propose a simpler model Flattened Row LSTM. We show that we can get good results on CIFAR10 and reconcile L_2 loss and visual quality.

Introduction

Image Inpainting consists in rebuilding missing or damaged patches of an image. Typical applications are old photos or paintings restoration, as well as image editing: Photoshop has a powerful completion tool (which can be used a as removal tool). As Convolutional Neural Networks (CNNs) now yield better-than-human classification accuracy on ImageNet [2], inpainting results are still visibly worse than human predictions. This can be explained by the fact that there are about 50,000 different ways to fill in a small 8x8x3 section, when ImageNet only has 32,000 classes. Interestingly, it is easy for us to mentally reconstruct a missing section of the image. We compare the context with our knowledge of the world. This allows us to recognize scenes, objects and extrapolate missing parts from memory, stitching them to the context. Artificial methods rely on the same principles.

There are two types of methods. Local methods only use informations of the context, such as color or texture. Then, they try to expand and merge them in a smooth way. Those need very little training or prior knowledge. A nose removed from a face would be replaced by a patch of skin-textured pixels. Indeed, the model cannot infer the existence of an object - a nose - at this position when it is completely missing. They work great for watermark removal, but not for larger missing patches. More powerful methods are global, context-based and semantic. They learn to recognize patterns in images (a window, a car) and use them to fill the gaps. These methods understand the necessity of the presence of a nose at a certain position of a face and are able to construct one that fits the context from their knowledge.

An interesting aspect of this type of problems is the ability to easily generate massive training datasets: any image dataset (we used CIFAR10 and ImageNet) can be preprocessed by simply altering images. One can generate hundreds of millions of training examples and train very deep networks.

1. Problem Statement

Each image is divided into two sections: the missing part that we try to reconstruct, and the context. For clarity, we assume the missing section is a square of dimensions $n \times n$, but the network would work identically for arbitrary removals.

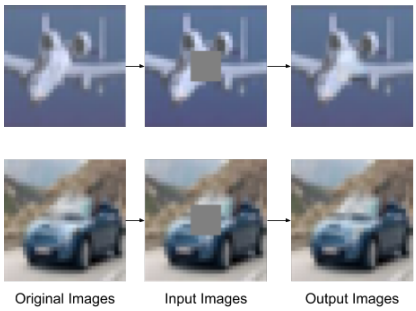

Figure 1. Definition of Image Inpainting

Image Inpainting is usually framed as a constrained image generation problem. The network has to be able to take a context as an input, and to output an image of the same dimensions as the missing patch. Final evaluation is based on the average element-wise L_2 distance between the original missing section $Y \in \mathbb{R}^{n \times n \times 3}$ and the prediction $\hat{Y} \in \mathbb{R}^{n \times n \times 3}$. In our running example on CIFAR10, $n = 8$. For a sample i, the loss is

$$L_i = \frac{1}{n^2} \sum_{p,q,r} (Y^{(i)}_{p,q,r} - \hat{Y}^{(i)}_{p,q,r})^2$$

Another interesting evaluation-only metric we propose is the approximate exact-match (AEM). We notice that a

shift of a few units in a channel pixel value has very little visual impact (see figure 2). Hence, if a predicted pixel is within ± 5 from the exact image on each of the three channels, it can be considered an exact match. We will report the mean-AEM, or MAEM, where 100% means that the image is visually almost indistinguishable from the ground truth.

Figure 2. Original image (left), ± 5 randomly added to each pixel channel (right)

There are two types of inpainting: blind – where the network does not know the position and the shape of the missing area – and non-blind, where those informations are passed as inputs. From our literature review, it appears that blind inpainting is a very hard problem. Although more documented, there is still a lot of room for improvement in non-blind inpainting. Hence, our decision to focus on the latter.

Our main goal is to leverage the latest and successful architectures and techniques in computer vision to build an efficient and robust inpainter. We hope to achieve acceptable results in terms of L_2 loss compared to state-of-the-art models. Our preliminary tests and the literature suggest very small practical differences between either having rectangular masks of randomly chosen shape and position or having a square-mask in the center of the images. Hence, as a coding simplification, we will mostly work on centered square masks of constant size. On the CIFAR10 dataset, we achieve this by removing a 8x8 patch in the middle of each image.

2. Related Work

Several authors explored a large variety of techniques to address similar issues. In [3], Pathak et al. uses an approach that was our initial inspiration. They modify the typical GAN architecture by inputting the image context instead of random noise to predict the missing patch. They emphasize the importance of Leaky ReLU in the discriminator and of the absence of pooling. Compressions and decompressions are made using strides different than 1. They use a combination of L_2 loss and adversarial loss (how well the generator can fool the discriminator) to train the model. However, they use in several instances fully connected layers that we found increase the risk of overfitting. Also, they use relatively simple CNN architectures for encoder and decoders: we wanted to try modern architectures that yield state-of-the-art results on other problems: VGG, Inception, etc.

In [4], published by a team of researchers from Google DeepMind, the authors propose a way to model the distribution of images as a product of conditional probabilities. The idea is to predict the pixels of an image in a specific order, from the top-left corner to the bottom-right corner for instance. The conditional probability functions are modeled either by a convolutional neural network on a neighborhood of pixels (Pixel CNN), or using a recurrent neural network to encapsulate the information from pixels located in the top rows of the image. Though this method is initially meant to generate images, it can be used to solve our problem of maximizing the likelihood of the reconstructed image given the pixels we already know.

In [11], Yang et al. propose a method to tackle inpainting of large sections on large images. The issue of traditional models on this tasks is that they tend to produce blurry results, with visible edges between context and reconstitution. They adapt multi-scale techniques to generate high-frequency details on top of the reconstructed object to get high resolution inpainting. They train two networks: a content network evaluated with a holistic content loss, similar to [3] approach, and a texture network that minimizes a local texture loss. They use neural matches to ensure that the texture inside the constructed crop is similar to the texture of the local context. This dual approach yields much sharper results visually, but in our case (small images) is not necessary.

The architecture/pipeline is a heavily adapted version of CIFAR10 classification pipeline from Tensorflow tutorials. The only part we kept was the queuing system and monitoring session. The autoencoders/GANs/WGANs we use are not available online in Tensorflow. One repository provides an adaptation of [3] in Tensorflow, but it is not efficient nor functional, we did not use it. When experimenting with state-of-the-art models (VGG, Inception, ResNet), those come from the Tensorflow model zoo, and we had to modify them to adapt them to CIFAR10 rather than ImageNet (usually simplifying the model as our images are simpler). PixelCNNs exist online but we did not use the existing code as we wanted to gain experience implementing them ourselves. We also did some changes compared to the article as described below, so using the pre-existing implementation would not be very useful.

3. Dataset

Our main dataset is CIFAR10: composed of 32*32*3 images, it contains 50,000 train examples and 10,000 test examples. CIFAR10 is a subset of the 80M Tiny Images dataset. Once our algorithms are stabilized, we leverage this larger dataset to train on a greater variety of images (CIFAR10 only contains 10 classes), and the gigantic number of samples is a very efficient solution to overfitting. Our pipeline will allow us to perform this scaling very easily

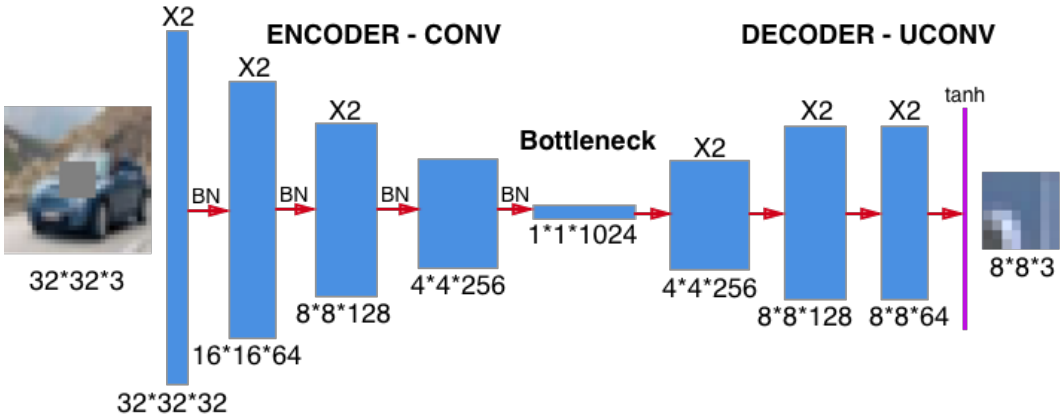

Figure 3. Overview of our basic autoencoder architecture

and achieve high training performance. So far, our models train in a less than a few hours on the 50,000 examples of CIFAR10.

As an intermediate step, we use data augmentation on CIFAR10 to increase the robustness of our algorithms to small changes, and virtually generate a larger number of samples. We apply small, random hue, saturation, contrast changes to the images, as well as random gaussian blur.

We replace the center 8*8*3 crop by zeros. Some articles suggest that using random Gaussian noise or mean context color for the mask would yield better results, but we did not notice significant improvements.

4. Autoencoders

Inpainting is part of a large set of image generation problems. The goal is creating or modifying pixels: deblurring, denoising, text removal (i.e. small-scale blind inpainting). Methods to solve those problems usually rely on autoencoders. Autoencoders are made of two networks: the encoder and the decoder. First, we encode the image into a lower-dimensional representation of an image (an embedding). Then, the decoder will work from this embedding to reconstruct the original image. The two networks are trained jointly to minimize the difference between input and output. This architecture forces the encoder to try to encode as much information as possible into the embedding. Since this is a small vector (a bottleneck), the encoder has to learn high-level, smarter features to compress the input with minimum loss. Those high-level features are hence more robust to noise and changes than raw pixels. For CIFAR10, the input image is a vector of length 3072, and we found that using a bottleneck of 1024 or 512 worked well. Smaller ones would not allow for enough space to encode the input information. Larger ones would not be a strong enough incentive for the autoencoder to find concise representations: storing patches of raw pixels does the work.

In the literature, the ratio between image size and bottleneck size is usually greater than ours (3 to 6). Paradoxically, this is because we are working on small images which makes it harder to spot coherent objects and elements on the image. Higher compression can be achieved on ImageNet (12 or more for instance) because it is easier to isolate high-level features on those.

In order to establish a baseline, we first built a simple CNN architecture. Input is $32 \times 32 \times 3$, and output is $8 \times 8 \times 3$. Literature encourages us to use several small filters rather than bigger ones. Indeed, we can get the same receptive field with deeper networks. For the entire project, we only used filters of size 3. We tested size 5 and 7 several times, but they always performed worse.

We experimented with a [CONV-CONV-POOL2]$\times$3 architecture, followed by 2 fully connected layers or 2 convolutional layers. This architecture is analog to a canonical well-performing architecture on CIFAR10 and allows for smooth dimension reduction along with the increase of the number of filters (starts at 32 or 64 and doubles at each pooling). We then eliminated max pooling by entirely replacing it with convolution layers of stride 2, the number of filters follows the same progression. Instead of using fully connected, moving from the 4*4*256 convolutions to the bottleneck can be seen as a stride 4 operation with 1024 filters.

In every case, ReLU proved to work better. Using Batch Normalization on each layer also yielded significant (around +15%) performance improvements with similar training times: the difference is probably lower if we wait until convergence in both cases.

Qualitatively, as discussed before, blurriness is an issue: colors are often right but details, textures are lost and predicted sections are usually blobs that do not blend very well into the image. We want to reduce the visible continuity errors between the predicted section and the context by predicting a patch that slightly overlaps with the context and by adding a very strong penalty to the loss on this overlap.

Figure 4. Example results from the Vanilla CNN approach

5. Generative Adversarial Networks (GANs)

Our initial model worked fairly well in terms of L_2 loss. However, output images were always blurry and lacking details (see figure 4). L_2 loss encourages the network to take little risk and make "safe" predictions: no hard shapes, no major changes throughout the patch. This is because outputting a blurry, mean-color image based on context guarantees a relatively low error because of the impossibility to make big mistakes. Therefore, although good in terms of L_2 norm, the outputs are not realistic to a human eye.

5.1. DCGANs

In order to force our network to take more risks, we decided to try adversarial networks. The idea is to reproduce the judgment of the human eye: if the missing part is a car window, predicting a car window even if it does not perfectly fit the car is better than predicting a blurry red patch with a black indistinct shape in the middle. An average blurry image will never be realistic. The main idea in GANs is to train in parallel a discriminator network ($\mathbb{D}$) that will learn to assess how real an image looks. The goal of the discriminator is to distinguish between the real patches and the generated ones. It is trained on a "dataset" composed of real and generated sections with corresponding labels (1 is a real image, 0 is an artificial one). Now, we penalize our generator ($\mathbb{G}$) by increasing its loss if the output image is considered artificial by our discriminator. In order to fool the discriminator, the generator will have to output real-looking, sharp and well defined images. The discriminator will improve by training on those more sophisticated examples. Hopefully this positive feedback loop continues where both networks improve by trying to fool each other.

The L_2 loss will not necessarily improve because our generator might make larger mistakes than before. However, images will look more real, and that is what matters in the end: if the predicted car window does not look at all like the ground truth, L_2 loss would be high but as long as the insertion is believable in the context of the car, this is a satisfying result.
The new loss is $L = \alpha L_{rec} + (1 - \alpha) L_{disc}$, where L_{rec} is the L_2 loss defined above, and L_{disc} is a sigmoid cross entropy on the probabilities outputted by the discriminator about the generated images.

$$L_{disc} = -\sum_i \log(p_i) = -\sum_i \log(\mathbb{D}(\hat{Y}^{(i)}))$$

Since α changes the scale of the loss, we can not use the loss itself to optimize this crucial hyperparameter. This issue is connected to the aforementioned fact that a lower L_2 does not necessarily mean a better, sharper visual result. We choose α to optimize the visual output of our samples and to ensure convergence.
The discriminator is a 6 CONV network, with every other CONV having a stride of 2, followed by a final CONV1 reducing depth to 1. Output is a scalar. One can see the discriminator as performing a classification problem. Hence, several state-of-the-art models could be adapted to work as a generator. Tensorflow offers a vast zoo of pretrained models that we leveraged to improve our discriminator: VGG, Inception, Resnet. Those models typically take larger inputs (ImageNet for instance), so we tried padding our images but results were poor. Hence, we adapted and retrained those models to work on our smaller inputs. We reduced depth by removing all the first layers with dimensions higher than 32x32 and injected our input in the first smaller layers. As our input contains much less information, we also decreased the number of filters to avoid overfitting and improve training time. The same holds for the encoder: we tested the same variety of high-performing models.

Figure 5. Example results from the DCGANs approach

5.2. Learning tricks

We have not been able to properly train a GAN. Visually, predicted images present colored artifacts (see 5). Upon careful analysis, shapes and color gradients appear to correspond to the context, but colors are deformed (it is particularly visible on the plane). Because we have not been able to get a good convergence on the GANs, L_2 loss is higher than vanilla autoencoders. With our current architecture, randomness plays a major role: the exact same code will sometimes converge or diverge.

They are unstable, and very sensitive on the network architectures. Finding the right hyper parameters and architecture details is tedious. The other common issue is the fact that one of the discriminator or the generator would get

much better than the other, and outpace, overwhelm it. For instance, if the discriminator gets too strong, the generator is unable to fool him at all and the adversarial loss skyrockets and plateaus at high numbers that conceal the L_2 loss. On the other, the generator might become so strong that it can perfectly cheat the discriminator, which stops learning and assigns the exact same probabilities to real and fake images. In this case, this is analog to not having adversarial loss (see previous part). We tried several learning tricks to alleviate those issues:

- Instead of mixing real and fake samples in a batch, we feed two separate, pure, batches. This allows more "clear, direct" updates for each batch: either we improve our knowledge of real examples, either we improve our detection of fakes, but not a confusing mix of the two.
- We use soft and noisy labels (see [13]): true images have a random label between 0.9 and 1.1 and fakes between 0 and 0.2. Labels are also sometimes randomly flipped between classes. Those tricks add noise and increase the robustness of the discriminator.
- Instead of minimizing $log(1 - D)$ where D is the output of the discriminator, we maximize $logD$. On this domain, those formulations are equivalent but the second does not suffer from vanishing gradients early in the training process.

5.3. WGANs

Wasserstein ([9]) have been proposed to ease the training of GANs. They rely on the Earth-Mover distance, which is a good metric for learning distributions. WGANs minimize the EM distance (or rather, a sound approximation of it). WGANs hence allow us to estimate the EM metric during training, and empirically this metric correlates well with the visual quality of predictions (contrarily to L_2 loss). WGANs do not rely on a realness probability from the discriminator, rather we call it a score: it is unbounded and has no "meaning" per se. In practice, we can just remove the final sigmoid layer in the discriminator. Also, a key assumption of WGANs is that functions are Lipschitz. We use gradient clipping to ensure this. Finally, we do not need to maintain a fine balance between $\mathbb{G}$ and $\mathbb{D}$: we can train $\mathbb{D}$ to convergence at every step. In practice, we train $\mathbb{D}$ 10 times at each iteration.

5.4. Reducing continuity errors

Using WGANs finally allowed us to train a decent adversarial network. However, a very visible problem remained: the border between the context and the reconstitution was very visible. Visually, it is hard to explain why this border appears so clearly. After careful analysis, it turns out that a systematic color mistake along a line of pixel is very visible, whereas when the error is random it blends (see figure 2). Even good predictions presented this border issue. We use a trick that consists in predicting a 16x16 center square. This square includes our 8x8 original target and 4 pixels of overlap on each side. The reconstruction L_2 loss on the overlap is 20 times more penalized, but it is easy for the network to predict it correctly since it is part of its input. This trick greatly improves the quality of the border merging, because the network will make sure there is no major discontinuity within its output. Thus, since the overlap will be very close to the real context, by preventing discontinuities between the overlap and the target we prevent discontinuities between the context and the target. We only keep the 8x8 target when displaying the results.

Figure 6. Output without (left) and with (right) overlap trick

Before using this trick, we could not feed the entire image to the discriminator because it would use the obvious border to determine if the image was real or fake. With less visible borders, we have been able to feed the entire image to the discriminator and let it assess it as a whole. Indeed, whether it comes from a real image or not, a 8x8 square usually has little meaning in itself, hence the difficulty for the discriminator to make a decision. With the entire image, the judgment is easier.

6. Density based models

6.1. Key idea

Instead of using supervised techniques to fill in a localized gap in the center of the image, we decided to implement unsupervised models as well. The idea here is to estimate the probability mass function $\mathbf{p}$ of images as a product of conditional probabilities. Hence, given an image $\mathbf{x}$ with shape $n \times n$, whose pixels are $\{x_1, x_2, ..., x_{n^2}\}$, the probability mass function is given by

$$\mathbf{p}(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, ..., x_{i-1})$$

where $p(x_i|x_1, ..., x_{i-1})$ is the probability that the i-th pixel takes the value x_i given that the $i-1$ previous pixels take the values $x_1, ..., x_{i-1}$.

As each pixel is defined by a triplet of integer $(R, G, B) \in \{0, 255\}^3$, using the notation $x_i =$

$(x_{i,R}, x_{i,G}, x_{i,B})$ and $x_{<i} = (x_1, .., x_{i-1})$, we can write

$$p(x_i|x_{<i}) = p(x_{i,R}|x_{<i})p(x_{i,G}|x_{<i}, x_{i,R}) \times p(x_{i,B}|x_{<i}, x_{i,R}, x_{i,G})$$

Thus our final probability $\mathbf{p}(\mathbf{x})$ is the product of $3n^2$ terms and when reconstructing an image, our goal is to fill in the hole with the pixels by maximizing that probability.

Our approach is a greedy one, which means that assuming we have learned the characteristics of the function $\mathbf{p}$ and are given an image with m missing pixels at positions $i_1 < i_2 < ... < i_m$, we will first set the value $x_{i_1,R}$ given $x_{<i_1}$, then the value of $x_{i_1,G}$ given $(x_{<i_1}, x_{i_1,R})$, and eventually we set $x_{i_m,B}$ given $(x_{<i_m-1}, x_{i_m,R}, x_{i_m,G})$. Hence, though this approach is easy to implement and is computationally efficient, we do not have any guaranty that our reconstructed image maximizes the likelihood $\mathbf{p}(x)$.

One obvious thing to do is to define an order on the pixels: from the top-left corner to the bottom-right one, scanning the rows one after the other. Once the order is set, we must choose the kind of conditional probability that we want to learn. We will introduce two different methods, inspired from [4]: a pixelCNN, where the conditional probability of a pixel is defined through a convolution neural network on pixels in the neighborhood, and a flattened row LSTM, which differs a lot from the one proposed in [4], where this probability is the result of the pixels from previous rows fed to an LSTM network.

6.2. Pixel CNN

One way to model the conditional probability $p(x_i|x_{<i})$ is to use a CNN. We tried several architectures mixing convolutional and leaky ReLU layers, and ending with one fully connected - softmax layer for each of the 3 channels (R,G,B). We hence used the softmax loss function, that is for an image with n^2 pixels with estimated probabilities for the true value $\hat{p}_{i,\text{true}}$,

$$L_{softmax} = -\sum_{i=1}^{n^2} \log\left(\hat{p}_{i,\text{true}}\right)$$

One could argue that the softmax loss is not relevant as we are not performing a classification task: predicting 115 instead of 116 in not as bad as predicting 115 instead of 230. Indeed, if the true value for $x_{i,R}$ is 117, then a low probability $\hat{p}(x_{i,R} = 117)$ is not necessarily a problem as long the weight assigned to values in the range $[110, 125]$ is big enough. However, given the large number of training examples compared to the 256 possible values for each channel, this loss provided interesting results.

Our final model uses a zone of size 5×5 in the image for the first convolutional layer, located at the top-left side of the pixel, as shown on figure 7.

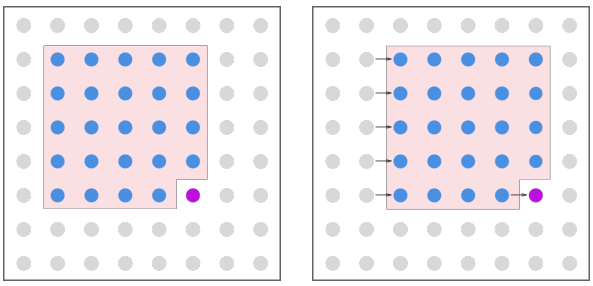

Figure 7. The blue pixels are in the neighborhood of the purple one, and are used to predict the conditional distribution. We then use the same network translated one step to the right to estimate the distribution of the next pixel, etc.

The advantage of this method is that the same network (and thus the same parameters) are used when computing each conditional probability. Since we always use the same number of pixels (in our case 24) to compute the distribution of the next pixel, the complexity does not depend on the size of the image. However, with a higher resolution such as 128×128 images, we might have had to use a larger neighborhood.

An example of probability mass function for a given pixel is given in figure 8. We notice that the color of this pixel is slightly unclear to the network, otherwise we would observe a sharper less spread out distribution.

Figure 8. Probability mass function of the three channels (R,G,B) of a pixel obtained with the CNN-FC-Softmax network

The results obtained with our Pixel CNN model are not quite as good as the one we got with GANs when considering the L_2 loss, which can be explained by two factors:

- We trained our Pixel CNN with a softmax loss, hence it is not surprising that the achieved L_2 loss, 6.98, is higher than the one obtained with GANs.
- The Pixel CNN scans the image from left to right, and the value given to a pixel is determined by the 24 neighbors at its top-left side. Thus the pixel at the bottom of the image are not used when filling the gap.

This partially explains the results we get on figure 9.

We can clearly see that the bottom and right parts of the image, which are dominated by clear colors (light grey, white), have not been taken into account when reconstructed the bird. Indeed, the discontinuity is much more visible at the bottom and at the right sides of the reconstructed square, and the few dark feathers at the top-left corner of the missing section have been extended to a large part of this square.

Figure 9. Original image (left) and reconstructed image (right) using the Pixel CNN model.

We also acknowledge the existence of the Row LSTM and Diagonal BiLSTM from the same article. We implemented and ran those methods, which worked well. Since we do not bring any new take on those techniques, we do not include a detailed report. However, given their complexity in terms of implementation and performance, we tried to propose a much simpler architecture that would still be able to bring relatively good performances on CIFAR10 and allow interesting analysis.

6.3. Flattened Row LSTM

We now propose a model which takes into account the information from all the previous rows of the image to compute the probability distribution of a pixel. We called this model Flattened Row LSTM, and its architecture is displayed on figure 10.

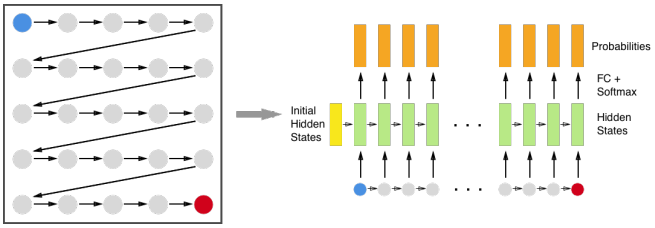

Figure 10. Flattened Row LSTM Architecture

The whole image is flattened from the top-left corner to the bottom-right one. The pixel channels are fed to an LSTM as 256-dimensional one-hot vectors in the predefined order $R \rightarrow G \rightarrow B$. The hidden dimension is 64, and the fully connected - softmax layer maps the hidden vectors to 256-dimensional output vectors.

The figure 11 shows an example of reconstructed image using our Flattened Row LSTM.

The result does not look as good as the one obtained with the pixel CNN model. We can give several explanations:

- The LSTM scans the rows in an order that gives more influence to the pixels located immediately on the left of the reconstructed pixel. Whereas we were using a dense neighborhood of 24 pixels at the top left corner of our pixel with the pixel CNN, this does not happen anymore.
- The strength of the LSTM model is that it captures all the information from the previous rows before making a prediction. However, the current model is probably too simple to make sense of this information. By using several LSTM networks or by increasing the hidden dimensions, we might be able to make more accurate predictions.

Figure 11. Original image (left) and reconstructed image (right) using the Flattened Row LSTM model.

7. Results

7.1. Vanilla Autoencoders

We used Adam Optimizer for training, and we cross-validated the learning rate. Using learning rate decay proved to be useful on all our tests to combine fast initial learning and smaller, later-stage decrease in loss. We have not been able to cross validate the decay rate and decay factor since such a process is complex.

We used dropout at 0.5 on every activations, and our trials did not show significant changes in accuracy within the reasonable range of values (40% - 80%) for the drop probability.

We studied the effect of the bottleneck size on our basic architecture as it is one of the most important parameter. That said, the best results do not stem from the basic architecture.

Bottleneck size	Train L_2 Loss	Val L_2 Loss
2000	4.21	8.11
1000	4.86	7.46
512	6.34	7.2
256	7.01	7.94

We can see that overfitting is an issue. Reducing the bottleneck size (hence the fully-connected layer size) greatly decreases the number of parameters, since this layer accounts for most of the parameters of the model. We can see overfitting decrease from 2000 to 512. 512 seems to be the sweet spot, and 256 is too small and probably not expressive enough. We already used early stopping and training

variables averaging (i.e. we do not use the final variables but a fading average of them throughout training, which increase stability). We plan to further combat overfitting by: using a larger dataset and/or data augmentation, implementing explicit L_2 regularization, and experimenting with dropconnect (dropping random CNN connections at train time, similar to dropout).

7.2. WGANs

As mentioned above, contrarily to our vanilla autoencoder, GANs do not optimize only for L2 loss. Theoretically, L2 loss results should not be better for GANs. That said, we spent much more time working on GANs because our best L2-only architecture gave very poor results visually: the loss function optimized and the visual quality do not align. Hence, our GANs have better results as a consequence of this experimental bias.

Since we have not been able to properly train a DCGAN, we only report result for our implementation of WGAN, which works. The score reported in the paper was a mistake: it corresponds to a coefficient 0 for the adversarial loss (hence it is a vanilla CNN).

Best results have been obtained with a VGG-like architecture. Again, we are not claiming that this is the best architecture as we have not been able to spend as much time on other alternative.

7.3. Density methods

Our Pixel CNN conditional probability function uses several convolutional - dropout layers. We did not use any pooling layers since it did not seem to improve the quality of our reconstructed images. Since the parameters of the CNN involved are shared across all the pixels, training the Pixel CNN comes down to performing a classification task using a 24-pixels zone ($5 \times 5 - 1$ as explained previously) as input and an integer in the range $[0, 255]$ as target. We used the Adam Optimizer with a cross-entropy loss and cross-validated the learning rate. The dropout rate was set to 50%.

Our Flattened Row LSTM model achieved a quadratic loss of 7.63 on the test set, though it was using the softmax cross-entropy loss. The hidden dimension 64 was cross-validated but due to computation time, we could not test as many values as we would have wanted. With several LSTM networks, we probably could have obtained a better L_2 loss.

7.4. Comparison

The results of the best run for each type of model on the test set is presented in the table 1 below.

A few held-out results from our best run are presented in figure 12..

Model	L_2 Loss
L_2 CNN	7.49
WGAN	**4.26**
Pixel CNN	6.98
Row LSTM	7.63

Table 1. L_2 losses on the test set.

Figure 12. Held-out results of our best run

Conclusion

We built a relatively efficient standalone CNN-based image inpainter. We identified the need for an adversarial loss, that we implemented using a GAN. We applied several tricks on GANs to ease training, and extended [3] to use WGANs and to leverage several existing state-of-the-art architectures (VGG, Inception, ResNet). We also propose applying the overlap trick to smooth borders and facilitate discriminator training. We also studied density-based methods. We implemented PixelCNNs from [4], and proposed our own model Flattened Row LSTM. We compared qualitatively and quantitatively the result to understand the shortcomings of the models. Overall, we are very satisfied with the results of our best architectures.

Future improvements would include transposing our models to larger images. Performance issues aside, this would prove interesting as large images present more objects, at a larger scale, and are overall more complicated. We would also like to work on our Flattened Row LSTM model to make it more symmetric and improving its generating power.

We would like the teaching team for providing us with this great learning opportunity, and Google for providing the precious GPUs that allowed us to train some of our heavy models.

References

[1] Ian Goodfellow. *NIPS 2016 Tutorial: Generative Adversarial Networks.* https://arxiv.org/abs/1701.00160

[2] Christian Szegedy, Sergey Ioffe & Vincent Vanhoucke. *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning.* https://arxiv.org/pdf/1602.07261

[3] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell & Alexei A. Efros. *Context Encoders:*

Feature Learning by Inpainting.
https://arxiv.org/abs/1604.07379

[4] Google DeepMind. *Conditional Image Generation with PixelCNN Decoders.*
https://arxiv.org/pdf/1606.05328

[5] Raymond Yeh, Chen Chen, Teck Yian Lim, Mark Hasegawa-Johnson & Minh N. Do. *Semantic Image Inpainting with Perceptual and Contextual Losses.*
https://arxiv.org/pdf/1607.07539

[6] H. Noori, S. Saryazdi & H. Nezamabadi-pour. *A Convolution Based Image Inpainting (2010).*

[7] Xiao-Jiao Mao, Chunhua Shen & Yu-Bin Yang. *Image Restoration Using Convolutional Auto-encoders with Symmetric Skip Connections.*
https://arxiv.org/pdf/1606.08921

[8] Alhussein Fawzi, Horst Samulowitz, Deepak Turaga & Pascal Frossard. *Image Inpainting Through Neural Networks Hallucinations.*

[9] Martin Arjovsky, Soumith Chintala & Leon Bottou. *Wasserstein GAN.*
https://arxiv.org/pdf/1701.07875

[10] Diederik P. Kingma & Max Welling. *Auto-Encoding Variational Bayes.*
https://arxiv.org/pdf/1312.6114

[11] Chao Yang et al. *High-Resolution Image Inpainting using Multi-Scale Neural Patch Synthesis.*
https://arxiv.org/pdf/1611.09969.pdf

[12] Tijana Rui & Aleksandra Piurica. *Context-Aware Patch-Based Image Inpainting Using Markov Random Field Modeling.*

[13] Salimans et al. *Improved Techniques for Training GANs*
https://arxiv.org/pdf/1606.03498.pdf