Checkered Convolutional Neural Networks

Improving the Resolution of CNN Feature Maps with Multisampling

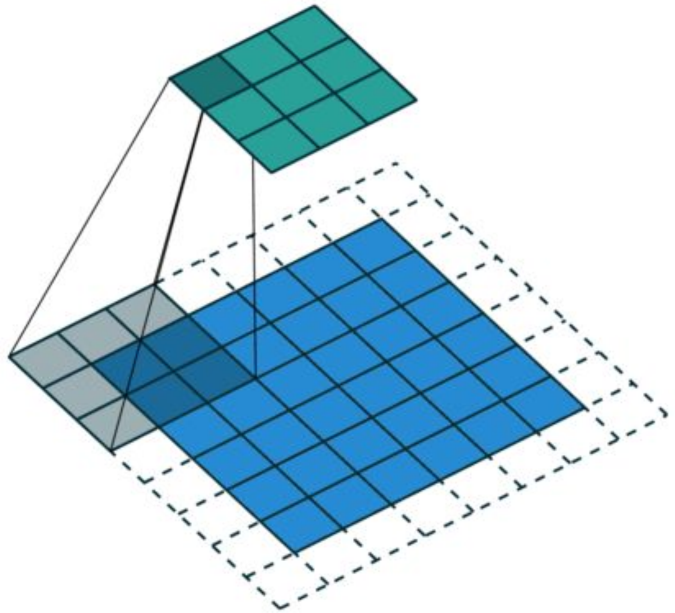
Paper: https://arxiv.org/abs/1805.10766

Code: https://github.com/ShayanPersonal/checkered-cnn

Prepared and presented by **Shayan Sadigh** in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Introduction

- An important component of convolutional neural networks (CNNs) are pooling layers and strided convolutional layers, which are used to reduce the size of feature maps.
- In general, we call these **subsampling layers**.
- Subsampling layers are used to increase the receptive field of kernels.
    - A 3x3 kernel on a 10x10 image has a much wider view of the image than a 3x3 kernel on a 100x100 image.
- They also reduces the computational complexity of deep layers, which can apply many thousands of feature detectors onto feature maps.
    - The smaller the feature map, the faster it’s processed.



Hyperparameters of layer:

- kernel_size=3
- padding=1
- **stride=2**

The blue feature map is the input, the green feature map is the output.
A 6x6 feature map is reduced to 3x3.
Assume padding is used as necessary to preserve the edges (outermost row and columns) of the input.

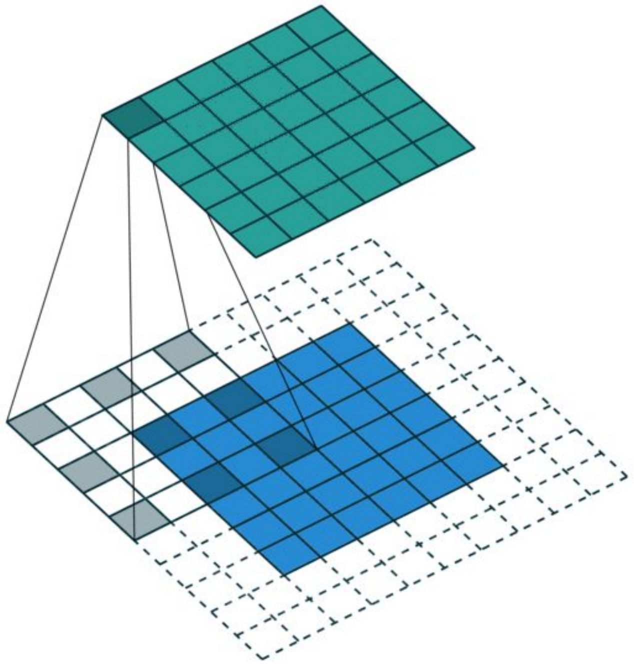
Animation from https://github.com/vdumoulin/conv_arithmetic

Problem

- We (people who design CNNs) lose a lot of spatial information whenever we use a subsampling layer.
- The most conservative subsampling layer, layers that use a stride length of 2, reduce the spatial capacity of the feature map by **75%.**
    - E.G. a 10x10 (100 elements) feature map is reduced to a 5x5 feature map (25 elements).
- This is a huge problem in generative models and tasks like semantic segmentation, where we need to be able to produce high-resolution, nice looking images from our feature maps.
- After just 5 subsampling layers we've already reduced the capacity of the feature map by **1024 times** (4^5).

Naive solution

- How should we increase the receptive field of neurons without losing so much information?
- We could use **dilated convolutions**.
- By permanently increasing dilation at certain points in the network, we can simulate the receptive field increase of subsampling, but without any actual subsampling.



Hyperparameters of layer:

- kernel_size=3
- padding=2
- **stride=1**
- **dilation=2**

The blue feature map is the input, the green feature map is the output.
Both the input and the output are 6x6.
In order to preserve the edges of the input, dilation needs a lot of padding.

Problems with dilation

- Now the spatial capacity of the feature map is never reduced, so the resolution of the input is the same as the resolution of the output.
- This is **extremely computationally expensive.**
- Deeper layers look for drastically more complex features than early layers. We need to reduce the resolution of the feature map so deep layers are more tractable.

What to do...

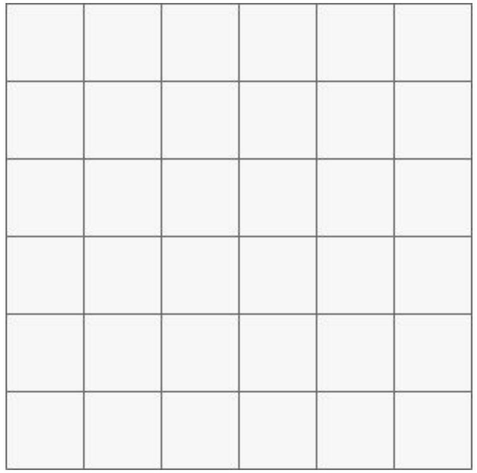
- We could go **back-and-forth** between increasing dilation and using a subsampling layer.
    - This is what semantic segmentation models like DeepLab do.
- But deciding when to increase dilation and when to use a subsampling layer is fairly awkward and arbitrary.
    - Adds even more engineering overhead and headaches when designing CNNs.
- Furthermore, each time you use a subsampling layer, there’s a sharp and sudden drop-off in the capacity of the feature map (75% loss at best). Could this be hurting learning?

Rethinking subsampling layers

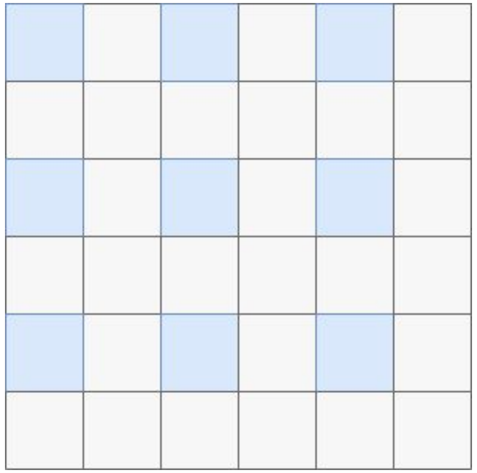
- Could there be a way to perform subsampling and increase receptive field, but not lose such a drastic amount of information in the process?

- **Yes.** There is a way to increase receptive field by 2x like a traditional subsampling layer with a stride length of 2, but only lose **50%** of the resolution of the input (rather than 75%).

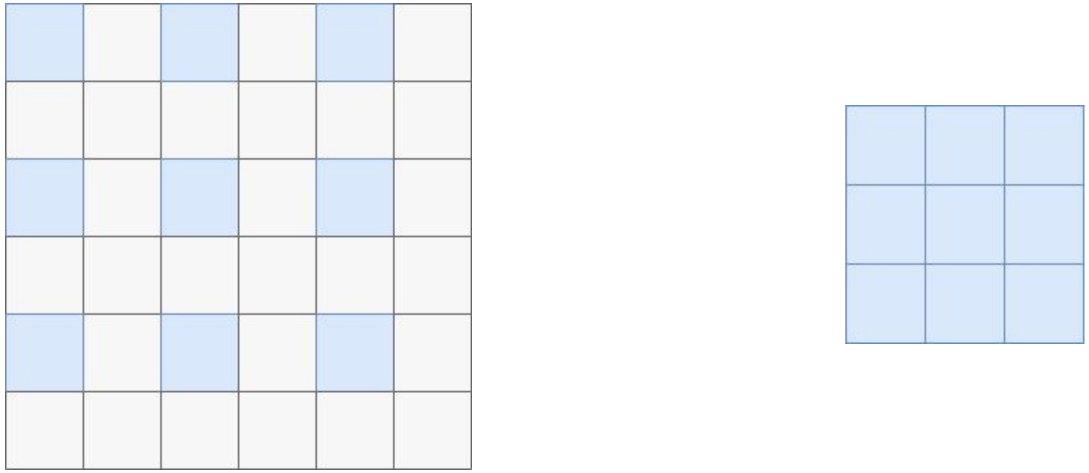
- In order to understand how this is possible, we need a new visualization of subsampling layers.



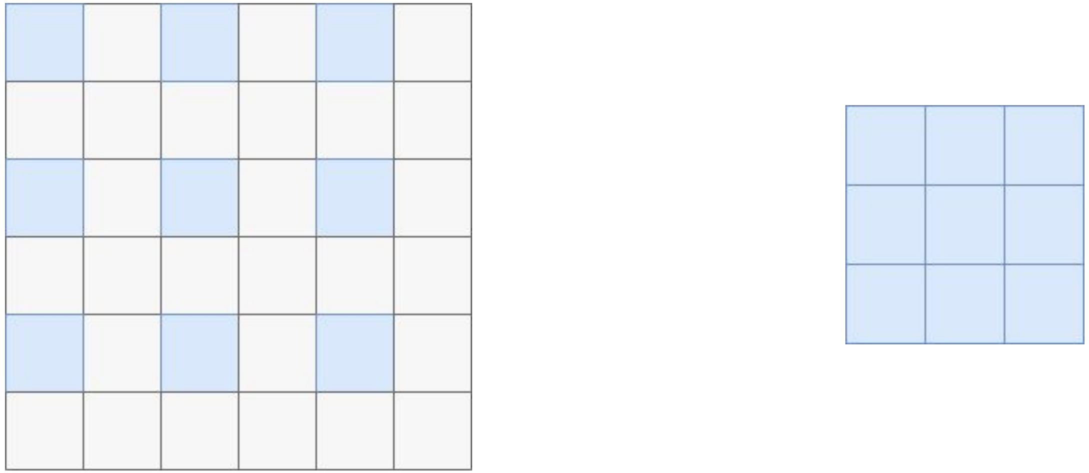
- Suppose we have this 6x6 input.
- Suppose we want to process it with a subsampling layer with stride=2 to increase our receptive field by 2x.



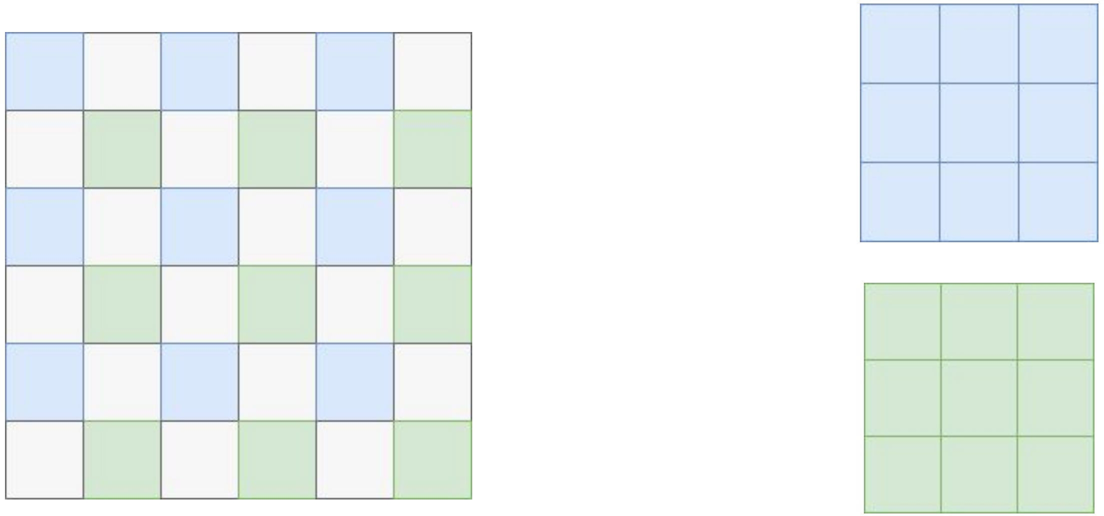
- Suppose we have this 6x6 input.
- Suppose we want to process it with a subsampling layer with stride=2 to increase our receptive field by 2x.
- We choose the blue elements to apply our convolution/pooling operation to.
    - The operation is lined up / centered on the blue squares, and padding is applied as necessary.



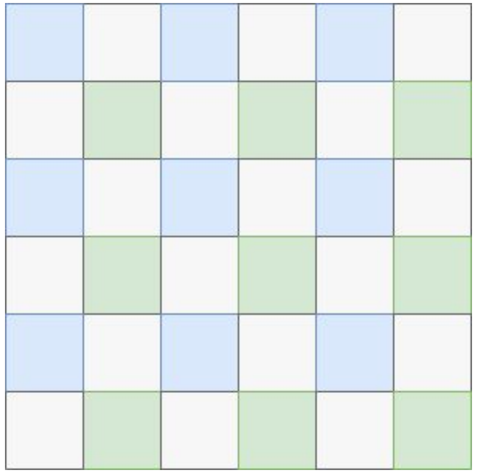
- Suppose we have this 6x6 input.
- Suppose we want to process it with a subsampling layer with stride=2 to increase our receptive field by 2x.
- We choose the blue elements to apply our convolution/pooling operation to.
    - The operation is lined up / centered on the blue squares, and padding is applied as necessary.
- The result is that we've applied our operation 9 times (that is, with respect to 9 samples). The 9 outputs are saved as the blue 3x3 output.

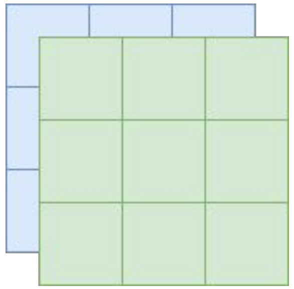


- This is exactly what current layers do.
- Notice how we only chose 1 in 4 of the elements in the input?
- **What if we choose more?**

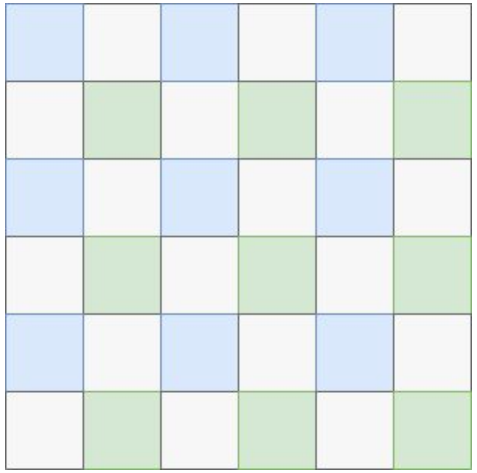


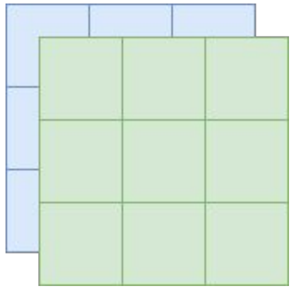
- Here we’ve chosen half the elements of the input to line up our operation on.
- The original set of blue samples are chosen as usual, the operation is lined up on each sample, and the results are saved to the blue feature map.
- But now we’ve added an additional set of green samples. The operation is also lined up with these green samples and the results are saved to their own green feature map, independent of the blue samples.





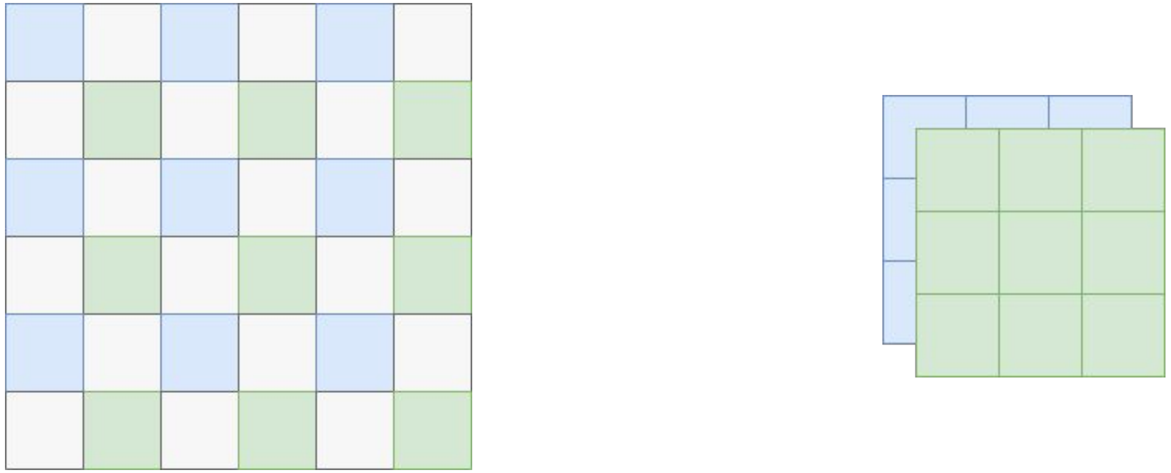
- Instead of thinking about them as separate feature maps, let’s think about them as feature **submaps** that belong to the same feature map.
- We just extend our feature maps with a third “spatial” dimension, akin to a depth dimension.





- As before, the height and width of our output is **3x3**, so the receptive field of future layers will be increased by 2 times.
- But now we have 2 submaps, so in total our output is **2x3x3** (# submaps * height * width).

- At the next subsampling layer, the blue and green submaps are processed separately. Each is split into 2 smaller submaps, resulting in a total of four submaps.



- The process of generating multiple submaps at a subsampling layer is called **multisampling**.
- This particular instance of multisampling is called **checkered subsampling** due to the checkerboard patterns it produces.

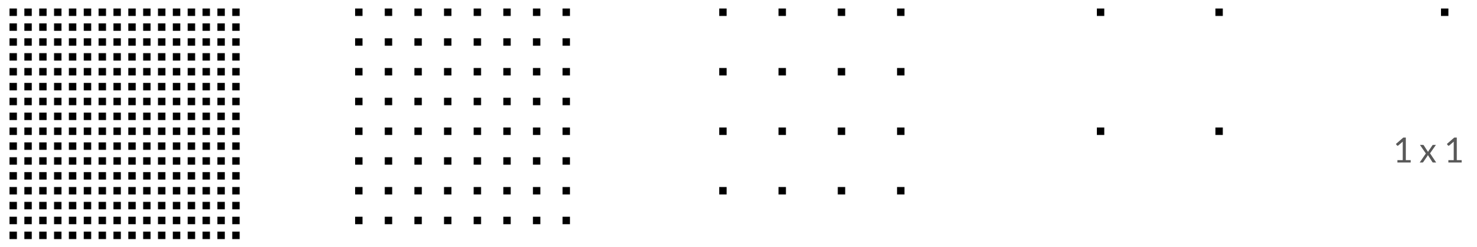
Checkered Subsampling

- **Checkered subsampling** increases receptive field by 2x like a traditional subsampling layer with a stride length of 2, but only loses **50%** of the resolution of the input (rather than 75%).

- After 5 checkered subsampling layers, the resolution of feature maps is only reduced by **32x** (as opposed to **1024x** with traditional subsampling layers).

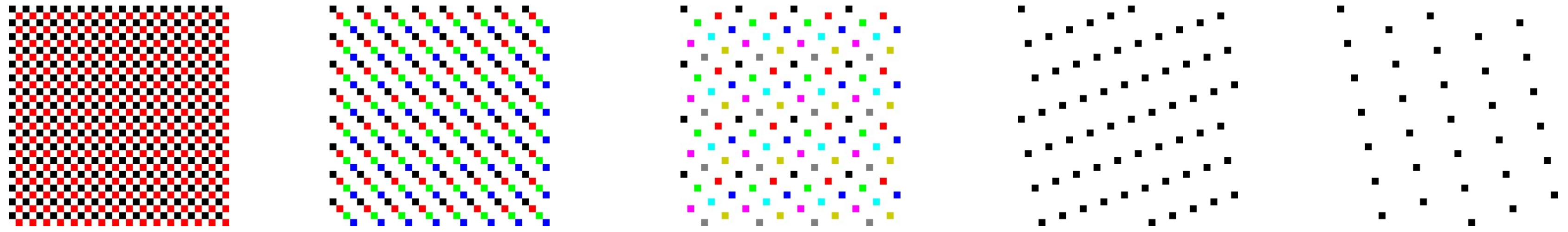
- A nice property of checkered subsampling is that it’s easily compatible with current architectures.
    - Because, by far, the most common subsampling layer in CNNs are those with stride=2.

A 32x32 feature map is processed by 1, 2, 3, 4, and 5 traditional layers with stride=2.
The resolution feature map is quickly reduced to nothing.



Checkered subsampling kept **32x** the information after 5 steps (2^5).

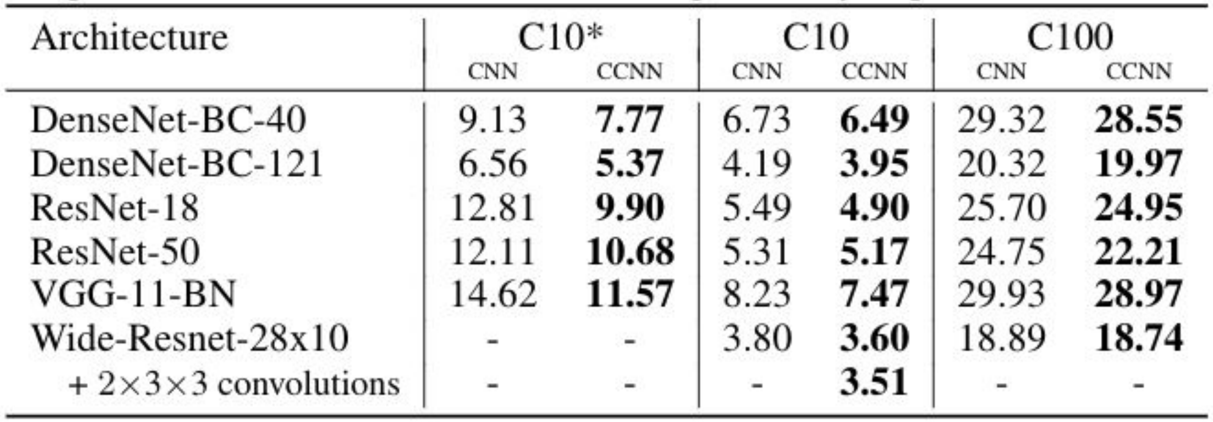
A 32x32 feature map is processed by 1, 2, 3, 4, and 5 checkered subsampling layers
First 3 steps are colored to help with the inuition (elements of the same color belong to the same submap).



Checkered Convolutional Neural Networks (CCNN)

- We call CNNs that use checkered subsampling at their subsampling layers **checkered CNNs (CCNNs)**.
- Just about all image classification models can be converted to a CCNN (ResNet, DenseNet, VGG, PyramidNet, SqueezeNet, SENets, MobileNet, and more…)
- It doesn't require any modifications to the parameters or architecture of the network. The network is still the same exact size, it just performs striding differently when stride > 1, and is able to generate and process **submaps**.

Table 1: Test error on CIFAR after training as a CNN and as a CCNN. The asterisk (*) indicates *without* data augmentations. Conversion to a CCNN significantly improves all models we test.



Architecture	C10*		C10		C100	
	CNN	CCNN	CNN	CCNN	CNN	CCNN
DenseNet-BC-40	9.13	**7.77**	6.73	**6.49**	29.32	**28.55**
DenseNet-BC-121	6.56	**5.37**	4.19	**3.95**	20.32	**19.97**
ResNet-18	12.81	**9.90**	5.49	**4.90**	25.70	**24.95**
ResNet-50	12.11	**10.68**	5.31	**5.17**	24.75	**22.21**
VGG-11-BN	14.62	**11.57**	8.23	**7.47**	29.93	**28.97**
Wide-Resnet-28x10	-	-	3.80	**3.60**	18.89	**18.74**
+ $2\times3\times3$ convolutions	-	-	-	**3.51**	-	-

- Our initial results training some popular architectures out-of-the-box as a CCNN without any hyperparameter tuning.
- Significant performance improvement across the board.
  - Dramatic performance improvement when data augmentations aren't allowed.
- These results demonstrate that low-resolution feature maps are significantly bottlenecking the performance of CNNs even in image classification.

Improving models without training

- We found that it’s not necessary to train new models from scratch to take advantage of checkered subsampling.
- Some pretrained ImageNet models, converted to a CCNN, showed significantly improved accuracy **without any training or fine-tuning**.
    - The learned filters of the CNN can be transferred to it’s CCNN counterpart without any modifications.

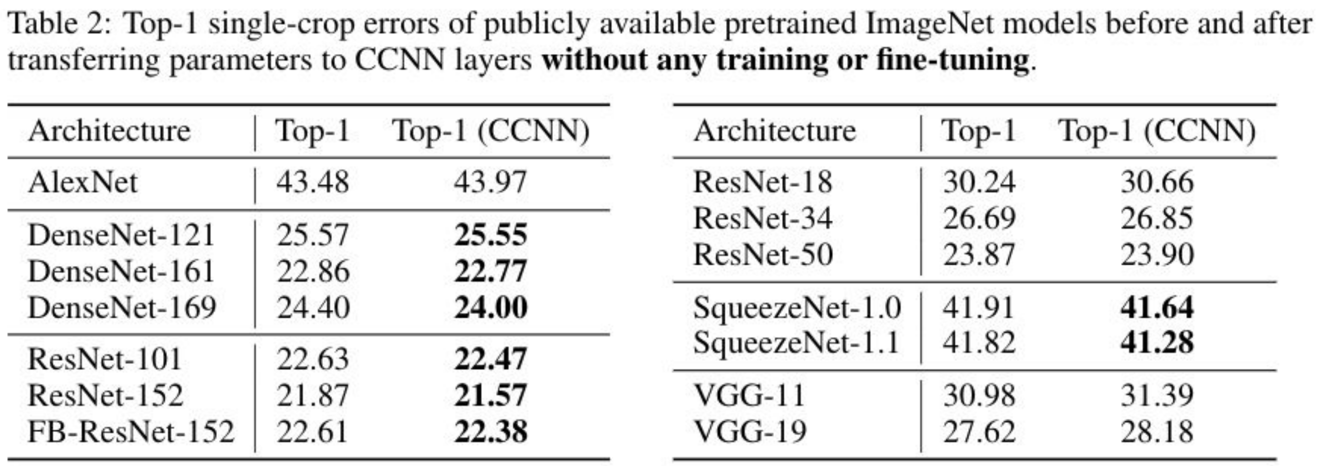


Table 2: Top-1 single-crop errors of publicly available pretrained ImageNet models before and after transferring parameters to CCNN layers **without any training or fine-tuning**.

Architecture	Top-1	Top-1 (CCNN)
AlexNet	43.48	43.97
DenseNet-121	25.57	**25.55**
DenseNet-161	22.86	**22.77**
DenseNet-169	24.40	**24.00**
ResNet-101	22.63	**22.47**
ResNet-152	21.87	**21.57**
FB-ResNet-152	22.61	**22.38**

Architecture	Top-1	Top-1 (CCNN)
ResNet-18	30.24	30.66
ResNet-34	26.69	26.85
ResNet-50	23.87	23.90
SqueezeNet-1.0	41.91	**41.64**
SqueezeNet-1.1	41.82	**41.28**
VGG-11	30.98	31.39
VGG-19	27.62	28.18

No training!

- As a CCNN, the pretrained model will generate 2^s submaps where *s* is the number of subsampling layers (typically 5 in ImageNet models).
- We simply average across the submap dimension to generate a single submap, which can then be treated as a traditional feature map and fed to the unmodified classifier.
- Absolutely **no training or fine-tuning** is performed.

Complexity

- The advantage of a CCNN in terms of spatial capacity grows *exponentially* with the number of subsampling layers.
- This leads to drastically more informative feature maps, but also means there will be a performance hit compared to a traditional CNN.
- However, due to how current CNNs are designed, the performance hit in practice is lower than you would expect.

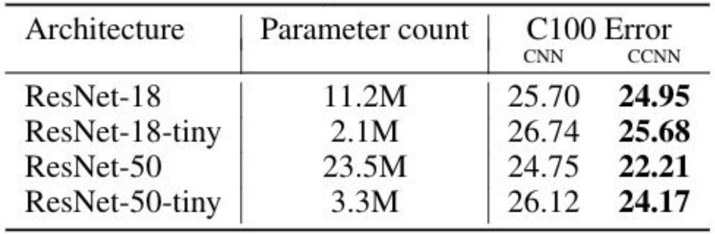
Memory Consumption

- Most of the memory usage in classification models is due to the **earliest** layers of the model, while checkered subsampling has its biggest effect **late** into the model.
    - Thus, memory usage does not grow exponentially with how many checkered subsampling layers you use.
    - In practice we observed 1.1 - 2 times memory usage training CIFAR models (2-5 subsampling layers).
- The memory impact during training can be completely eliminated (and more) with **gradient checkpointing** (now available in Pytorch and Tensorflow).

Compute time / FLOPs

- Compute time is less predictable and highly dependent on architecture and implementation.
  - In practice we observed 2 - 7 times increase in training time in CIFAR.
  - Note that we're using an unoptimized implementation using high-level Pytorch operations.
- Architectures built from the ground-up with checkered subsampling may be able to reduce the amount of computation in their later layers. This is because we suspect that current architectures need so many channels in their deep layers to **"remember" information that was lost through subsampling**, **and with a better subsampling scheme you could get rid of those extra channels.**

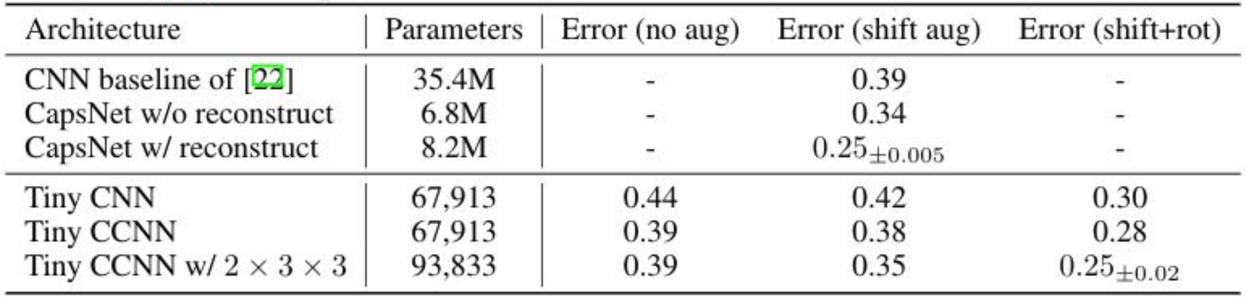
Table 7: Our tiny ResNet CCNNs are competitive with their full-sized CNN counterparts.



Architecture	Parameter count	C100 Error	
		CNN	CCNN
ResNet-18	11.2M	25.70	**24.95**
ResNet-18-tiny	2.1M	26.74	**25.68**
ResNet-50	23.5M	24.75	**22.21**
ResNet-50-tiny	3.3M	26.12	**24.17**

- We train “tiny” versions of ResNet and observe they are competitive with or better than their full-sized CNN versions when trained as a CCNN with over 7x fewer parameters.

Table 8: We create toy CNNs to test on MNIST and report their errors. We include state-of-the-art results from [22] for comparison.



Architecture	Parameters	Error (no aug)	Error (shift aug)	Error (shift+rot)
CNN baseline of [22]	35.4M	-	0.39	-
CapsNet w/o reconstruct	6.8M	-	0.34	-
CapsNet w/ reconstruct	8.2M	-	$0.25_{\pm 0.005}$	-
Tiny CNN	67,913	0.44	0.42	0.30
Tiny CCNN	67,913	0.39	0.38	0.28
Tiny CCNN w/ $2 \times 3 \times 3$	93,833	0.39	0.35	$0.25_{\pm 0.02}$

- Our toy CCNNs get pretty competitive with the state-of-the-art results reported in Hinton’s Capsule Network paper with dramatically less parameters.

- In this experiment, we compared checkered subsampling with an alternative method to extract more information from pretrained ImageNet models (**without training**) using dilation.

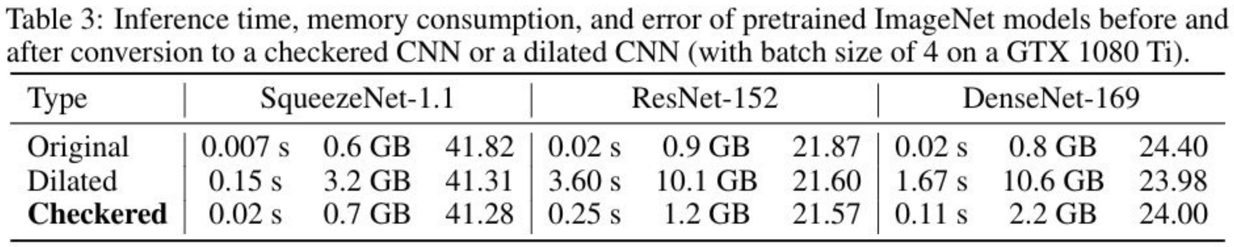


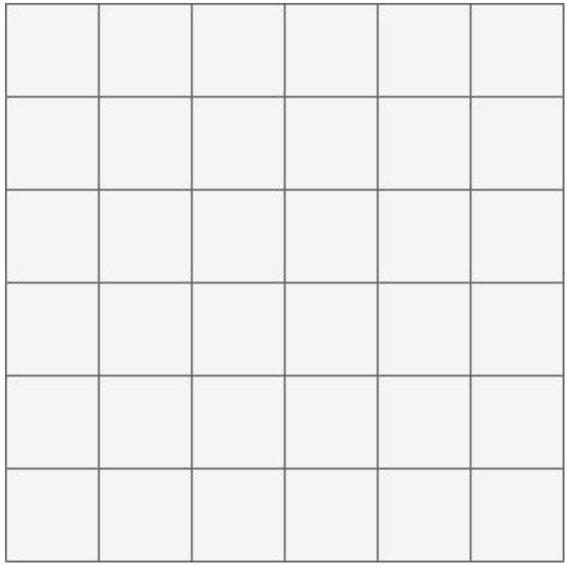
Table 3: Inference time, memory consumption, and error of pretrained ImageNet models before and after conversion to a checkered CNN or a dilated CNN (with batch size of 4 on a GTX 1080 Ti).

Type	SqueezeNet-1.1			ResNet-152			DenseNet-169		
Original	0.007 s	0.6 GB	41.82	0.02 s	0.9 GB	21.87	0.02 s	0.8 GB	24.40
Dilated	0.15 s	3.2 GB	41.31	3.60 s	10.1 GB	21.60	1.67 s	10.6 GB	23.98
Checkered	0.02 s	0.7 GB	41.28	0.25 s	1.2 GB	21.57	0.11 s	2.2 GB	24.00

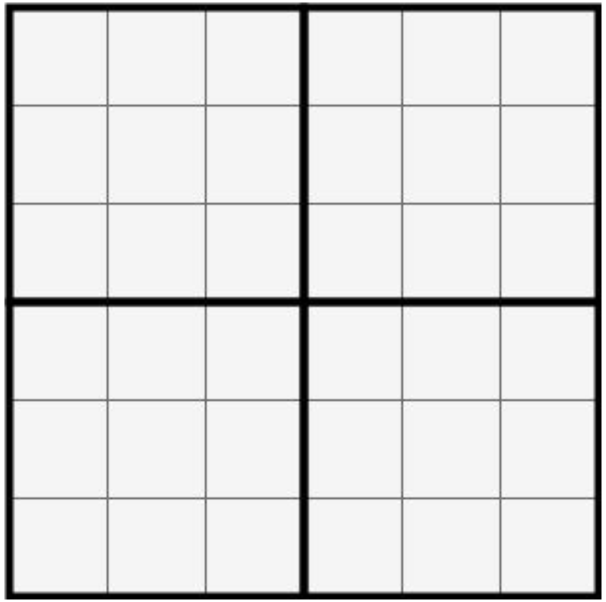
- Checkered subsampling is vastly superior to dilation in terms of performance impact.
- Furthermore, dilation does not offer better accuracy due to **diminishing returns.**
- Each ImageNet architecture is affected differently by checkered subsampling. These observations were made using Pytorch 0.4 on a GTX 1080 Ti.
- This performance impact can be avoided by training CCNNs with fewer parameters as in our tiny-ResNet experiment, or by using traditional subsampling in the earliest layers.

Multisampling

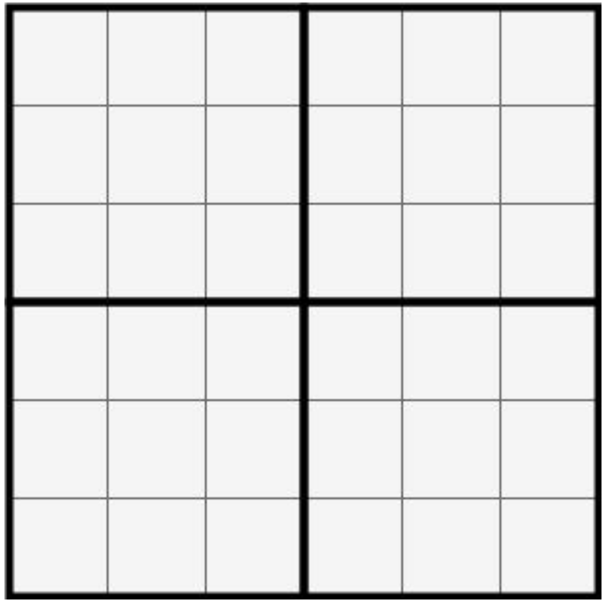
- Checkered subsampling, and traditional subsampling layers, are part of a broader class of subsampling schemes we call **multisampling**.
- In general, instead of setting stride=k, we split the feature map into k by k **sampling windows**.
- From each sampling window we pick a particular set of samples, and each sample within a window is stored in its own submap.
    - The set of samples we choose is determined by the **sampler** (a binary element-selector matrix).

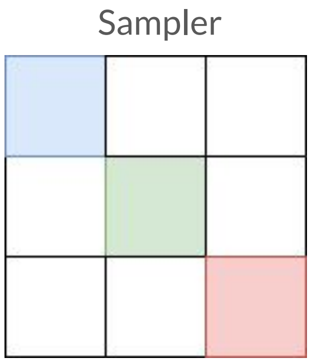


- Suppose we have this **6x6** input, and we want to subsample and increase our receptive field by **3x.**
    - (That is, we want our output height and width to be 2x2).

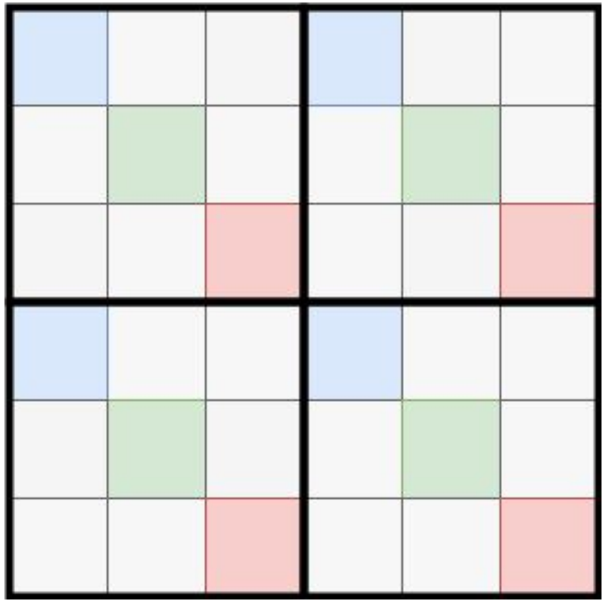


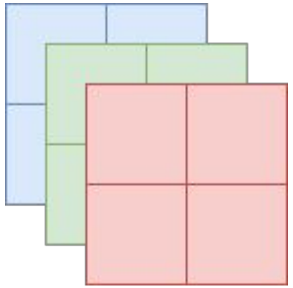
- Suppose we have this **6x6** input, and we want to subsample and increase our receptive field by **3x.**
  - (That is, we want our output height and width to be 2x2).
- We split the input into a grid of 3x3 sampling windows.





- Suppose we have this **6x6** input, and we want to subsample and increase our receptive field by **3x.**
  - (That is, we want our output height and width to be 2x2).
- We split the input into a grid of 3x3 sampling windows.
- We choose a sampler to apply on each sampling window (here's one that takes 3 samples in a diagonal).





- The sampler is applied on each sampling window.
- Each sample that's chosen within a window is stored in its own submap.

- A traditional layer with stride=3 would have chosen only the top left element of each sampling window (the blue elements).

- Notice how we extracted **3 times as much** information from the feature map than what a traditional layer with stride=3 would have extracted.

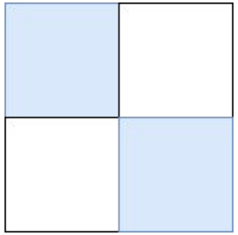
Implementing multisampling

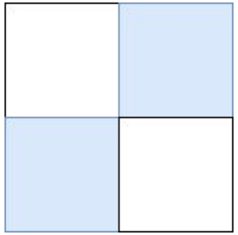
- The same sampler shouldn’t be applied to every submap or it will bias samples to line up in diagonals or clump up.
    - **Ideally, you won’t have to worry about this while designing CNNs**. This should be abstracted away by your deep learning framework.
- In the paper we provide a sequence of samplers that generates a low-discrepancy “lattice” sampling.
- Alternatively you can just randomly apply n-rooks samplers and it will give a nice, randomly-distributed sampling of the input.
- All samplers should satisfy the n-rooks property (no two samples within a sampling window should share the same row or column).

Implementing multisampling

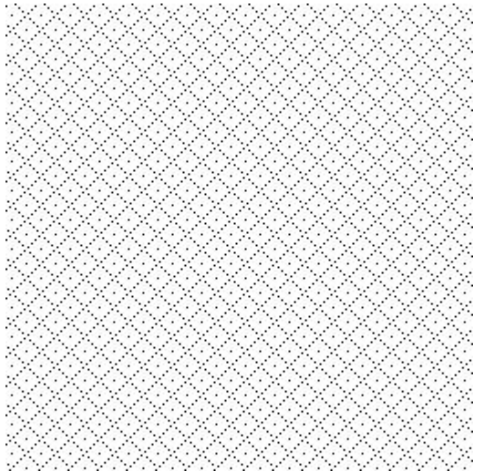
- Don't apply a 2D convolution separately to every submap (slow because the layer has to be executed for every submap). Instead, treat the submap dimension as a "depth" dimension and run a single 3D convolution on it (fast).
    - It's possible to create 3D convolutions that look across multiple submaps at once (E.G. 2x3x3 convolutions). This has some interesting effects, e.g., the convolution can learn "deformed" structures due to the semi-structured nature of the submap dimension.

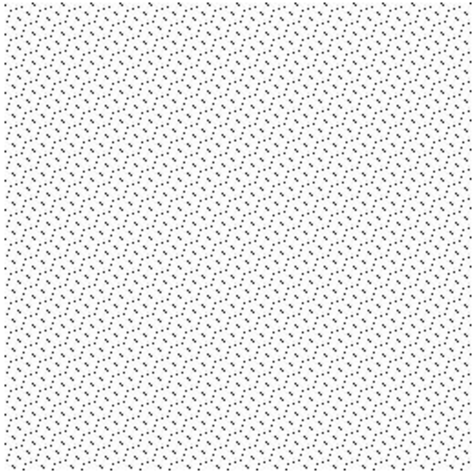
- We end with some patterns generated by applying these two samplers (the samplers used in checkered subsampling).

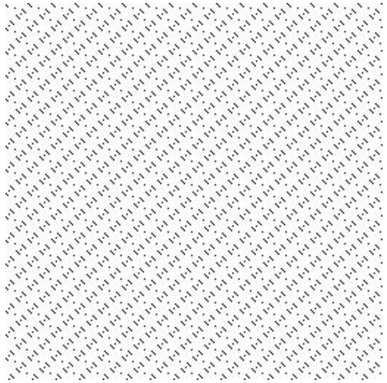


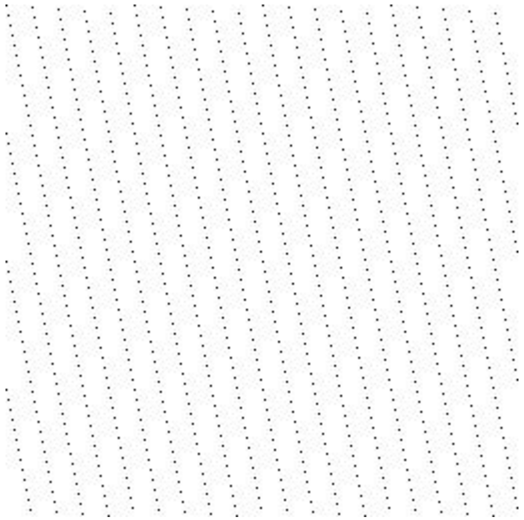


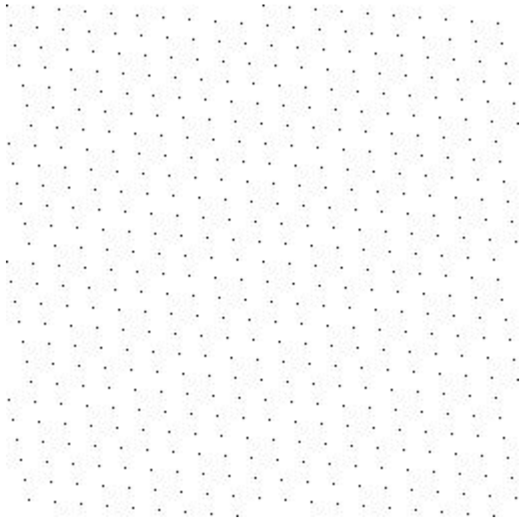
128x128 after 7 steps
(randomly applied)

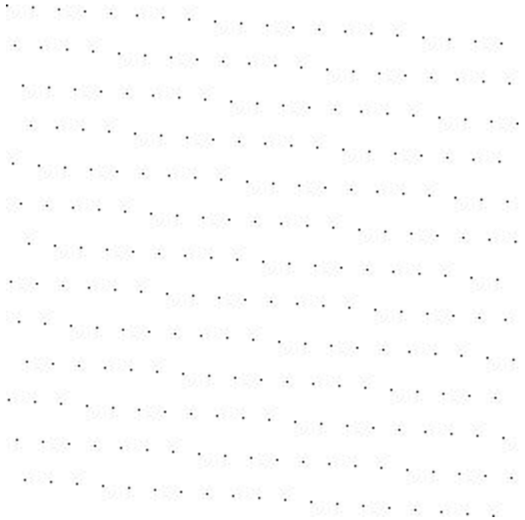

256x256 after 4 steps
(“Dot” pattern)


256x256 after 4 steps
(“DNA” pattern)


256x256 after 4 steps
(“Fighter” pattern)


64 x 4 x 4


128 x 2 x 2


256 x 1 x 1

Our “lattice” sequence after 6, 7, and 8 checkered subsampling layers on a 256 x 256 image.