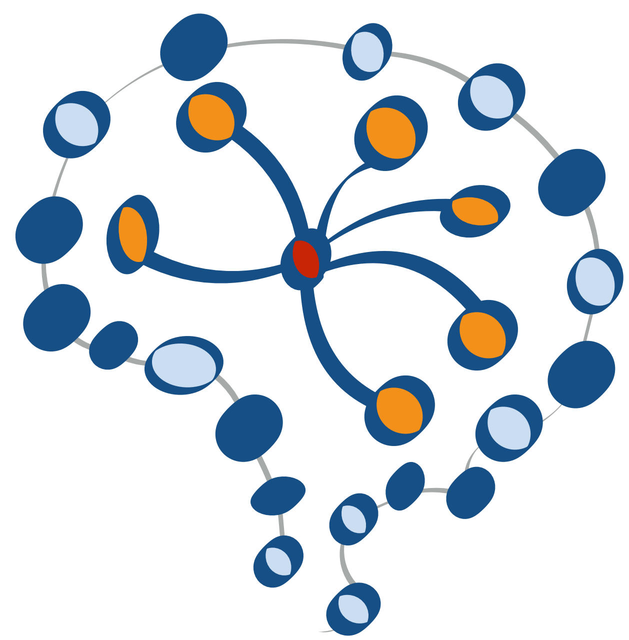
STAT 453: Introduction to Deep Learning and Generative Models

Sebastian Raschka
http://stat.wisc.edu/~sraschka



Deep Learning & AI News #6

Interesting Things Related to Deep Learning

Mar 6th, 2021

March 04, 2021

PyTorch 1.8 Release, including Compiler and Distributed Training updates, and New Mobile Tutorials

https://pytorch.org/blog/pytorch-1.8-released/

https://github.com/pytorch/pytorch/releases/tag/v1.8.0

- AMD GPU support via ROCm (binaries available directly from the installer menu)
- Now possible to fit large models onto GPUs w/o external libraries: pipeline and model parallelism
- Determinants & eigenvalues via torch.linalg w/o switching to NumPy

PyTorch adds binaries for AMD GPU support

PyTorch Build	Stable (1.8.0)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 10.2	CUDA 11.1	ROCm 4.0 (beta)	None
Run this Command:	`pip install torch -f https://download.pytorch.org/whl/rocm4.0.1/torch_stable.html`			

Previous versions of PyTorch >

https://pytorch.org (Currently only for Linux though)

```
The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    cudatoolkit-11.1.1         |       h6406543_8        1.20 GB  conda-forge
    pytorch-1.8.0              |py3.8_cuda11.1_cudnn8.0.5_0        1.27 GB  pytorch
    torchaudio-0.8.0           |             py38         4.4 MB  pytorch
    ------------------------------------------------------------
                                           Total:         2.48 GB

The following packages will be UPDATED:

  cudatoolkit                            11.0.3-h15472ef_8 --> 11.1.1-h6406543_8
  pytorch             1.7.1-py3.8_cuda11.0.221_cudnn8.0.5_0 --> 1.8.0-py3.8_cuda11.1_cudnn8.0.5_0
  torchaudio                                   0.7.2-py38 --> 0.8.0-py38


Proceed ([y]/n)? y


Downloading and Extracting Packages
pytorch-1.8.0        | 1.27 GB   | #9                                                     |   3%
```

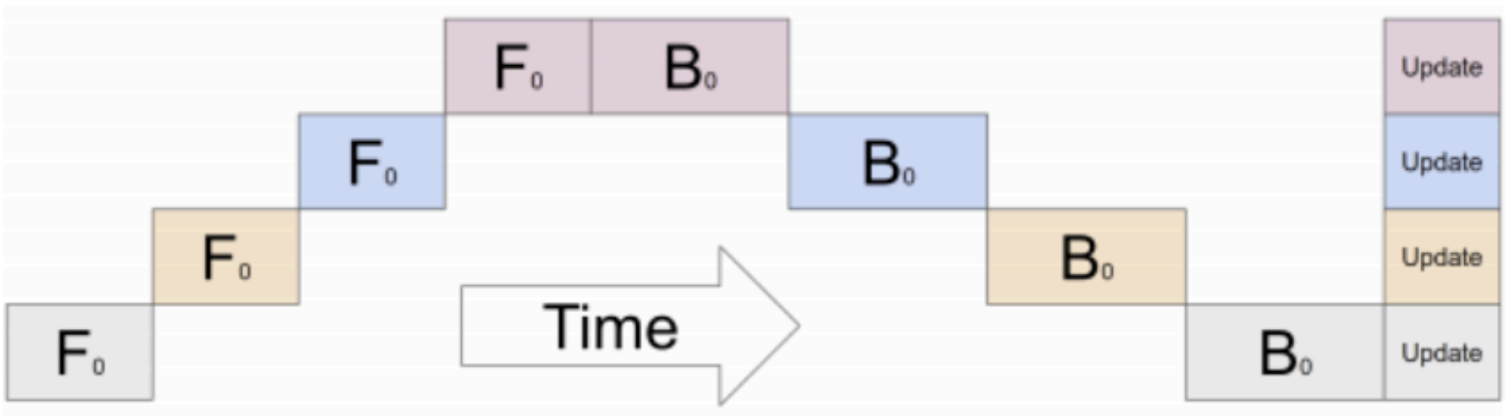
Distributed Training

- [Beta] Pipeline Parallelism
- [Beta] DDP Communication Hook
- ...
- (Prototype) ZeroRedundancyOptimizer
- (Prototype) CUDA-support in RPC using TensorPipe

Model Parallelism using multiple GPUs

https://pytorch.org/docs/1.8.0/pipeline.html?highlight=pipeline#

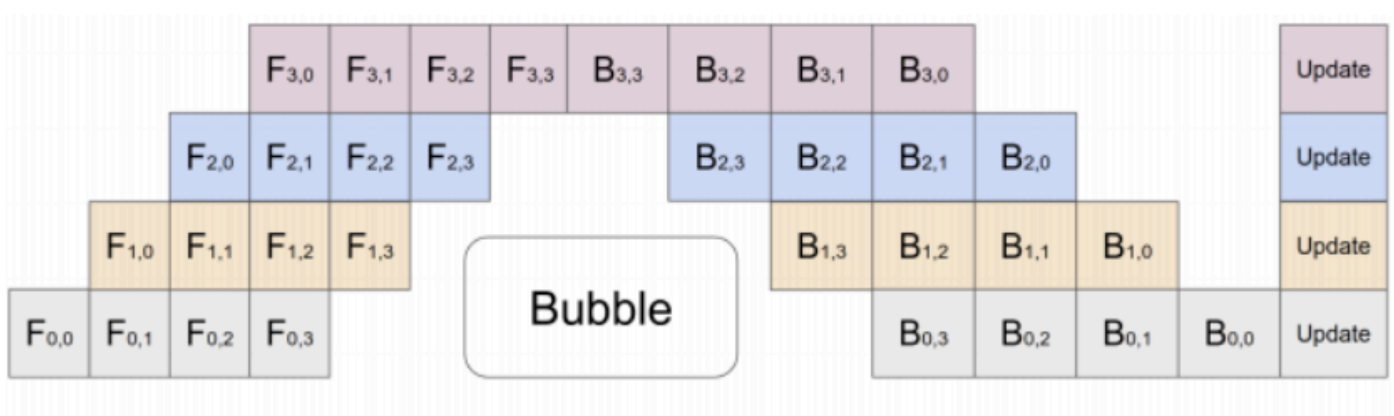
Typically for large models which don't fit on a single GPU, model parallelism is employed where certain parts of the model are placed on different GPUs. Although, if this is done naively for sequential models, the training process suffers from GPU under utilization since only one GPU is active at one time as shown in the figure below:



The figure represents a model with 4 layers placed on 4 different GPUs (vertical axis). The horizontal axis represents training this model through time demonstrating that only 1 GPU is utilized at a time (image source).

Pipelined Execution

To alleviate this problem, pipeline parallelism splits the input minibatch into multiple microbatches and pipelines the execution of these microbatches across multiple GPUs. This is outlined in the figure below:



The figure represents a model with 4 layers placed on 4 different GPUs (vertical axis). The horizontal axis represents training this model through time demonstrating that the GPUs are utilized much more efficiently. However, there still exists a bubble (as demonstrated in the figure) where certain GPUs are not utilized. (image source).

3) VGG16 with Pipeline Parallelism

Below we first define the blocks we are going to wrap into the model:

```
block_1 = torch.nn.Sequential(
        torch.nn.Conv2d(in_channels=3,
                  out_channels=64,
                  kernel_size=(3, 3),
                  stride=(1, 1),
                  # (1(32-1)- 32 + 3)/2 = 1
                  padding=1),
        torch.nn.ReLU(),
        torch.nn.Conv2d(in_channels=64,
                  out_channels=64,
                  kernel_size=(3, 3),
                  stride=(1, 1),
                  padding=1),
        torch.nn.ReLU(),
        torch.nn.MaxPool2d(kernel_size=(2, 2),
                     stride=(2, 2))
)

block_2 = torch.nn.Sequential(
        torch.nn.Conv2d(in_channels=64,
                  out_channels=128,
```

https://github.com/rasbt/deeplearning-models/blob/master/pytorch_ipynb/mechanics/model-pipeline-vgg16.ipynb

2. The chunks refer to the `microbatches` , for more details, see https://pytorch.org/docs/1.8.0/pipeline.html?higl

```
from torch.distributed.pipeline.sync import Pipe


block1 = block_1.cuda(0)
block2 = block_2.cuda(0)
block3 = block_3.cuda(2)
block4 = block_4.cuda(2)
block4 = block_5.cuda(3)
block4 = classifier.cuda(0)

model_parallel = torch.nn.Sequential(
    block_1, block_2, block_3, block_4, block_5, classifier)
model_parallel = Pipe(model_parallel, chunks=8)
optimizer = torch.optim.Adam(model_parallel.parameters(), lr=learning_rate)
```

```python
start_time = time.time()
for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.to(device)
        targets = targets.to(device)

        # FORWARD AND BACK PROP
        logits = model(features)
        if isinstance(logits, torch.distributed.rpc.api.RRef):
            logits = logits.local_value()
        loss = loss_fn(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        # UPDATE MODEL PARAMETERS
        optimizer.step()

        # LOGGING
        log_dict['train_loss_per_batch'].append(loss.item())

        if not batch_idx % logging_interval:
            print('Epoch: %03d/%03d | Batch %04d/%04d | Loss: %.4f'
                  % (epoch+1, num_epochs, batch_idx,
                     len(train_loader), loss))

    if not skip_epoch_stats:
        model.eval()
```

https://github.com/rasbt/deeplearning-models/blob/master/pytorch_ipynb/mechanics/model-pipeline-vgg16.ipynb

[Submitted on 28 Feb 2021]

Virus-MNIST: A Benchmark Malware Dataset

David Noever, Samantha E. Miller Noever

https://arxiv.org/abs/2103.00602

The work generalizes what other malware investigators have demonstrated as promising convolutional neural networks originally developed to solve image problems but applied to a new abstract domain in pixel bytes from executable files.

We converted the CSV format [16] to greyscale images using the intermediate NetPBM text format (PGM) to create ASCII-raw images, then the ImageMagick [25] command-line tools for compressing the image to viewable JPEG files.

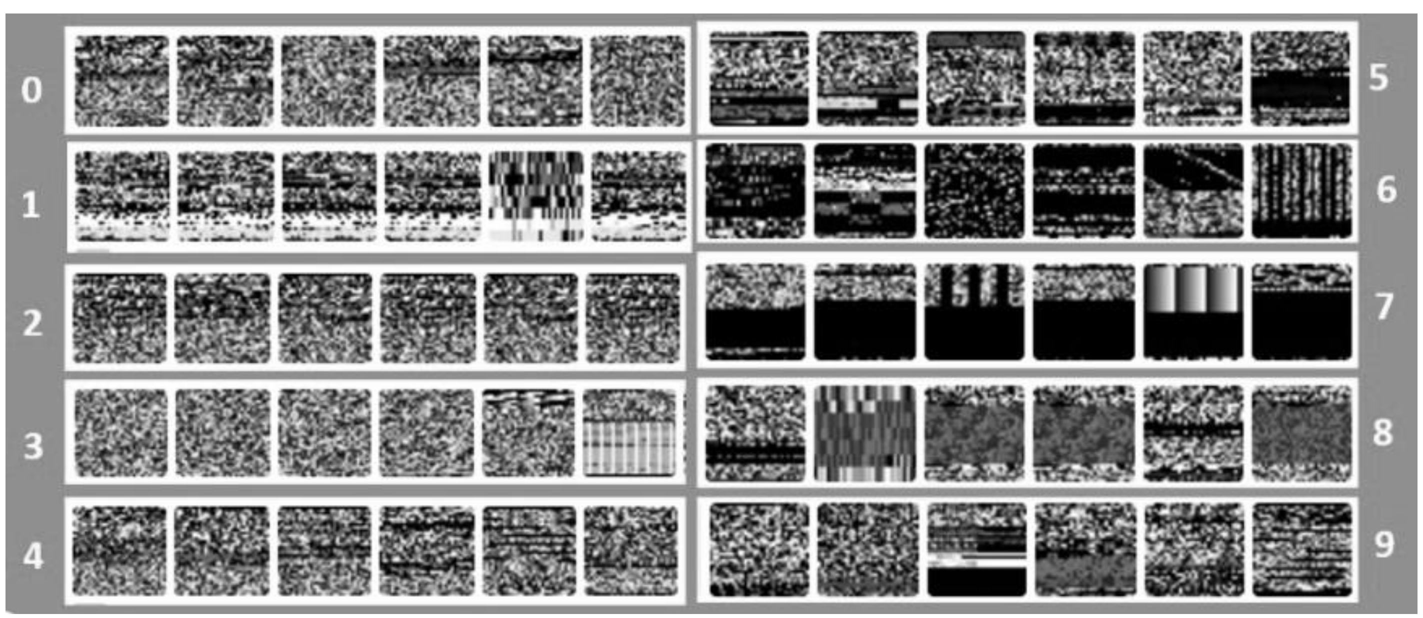


Figure 1 Virus-MNIST showing 10 classes. The "0" class represents non-malicious examples. The other 9 virus families were clustered using a K-means method to match with the standard MNIST format and multi-class

Class	Count	Group	Type	Example
0	2516	Beneware	Good	putty.exe
1	7684	Malware	Adware	IESettings
2	3037	Malware	Trojan	Supreme.exe
3	2404	Malware	Trojan	myfile.exe
4	796	Malware	Installer	myfile.exe
5	6662	Malware	Backdoor	myfile.exe
6	15377	Malware	Crypto	Powershell
7	7494	Malware	Backdoor	BitTorrent.exe
8	2571	Malware	Downloader	myfile.exe
9	3339	Malware	Heuristic	myfile.exe

Figure 2. Class distributions and example types for malware and beneware PE File headers.

🚧 Simple considerations for simple people building fancy neural networks

Published February 25, 2021 – Initially published on Medium, Sept 2020.

Update on GitHub


VictorSanh
Victor Sanh

https://huggingface.co/blog/simple-considerations

1) Put aside machine learning and simply focus on your data:

- Are the labels balanced?
- Are there gold-labels that you do not agree with?
- How were the data obtained? What are the possible sources of noise in this process?
- Are there any preprocessing steps that seem natural (tokenization, URL or hashtag removing, etc.)?
- How diverse are the examples?
- What rule-based algorithm would perform decently on this problem?

🚧 Simple considerations for simple people building fancy neural networks

Published February 25, 2021 – Initially published on Medium, Sept 2020.

Update on GitHub

VictorSanh
Victor Sanh

https://huggingface.co/blog/simple-considerations

2) Start as simple as possible to get a sense of the difficulty of your task and how well standard baselines would perform (e.g., use a logistic regression baseline)

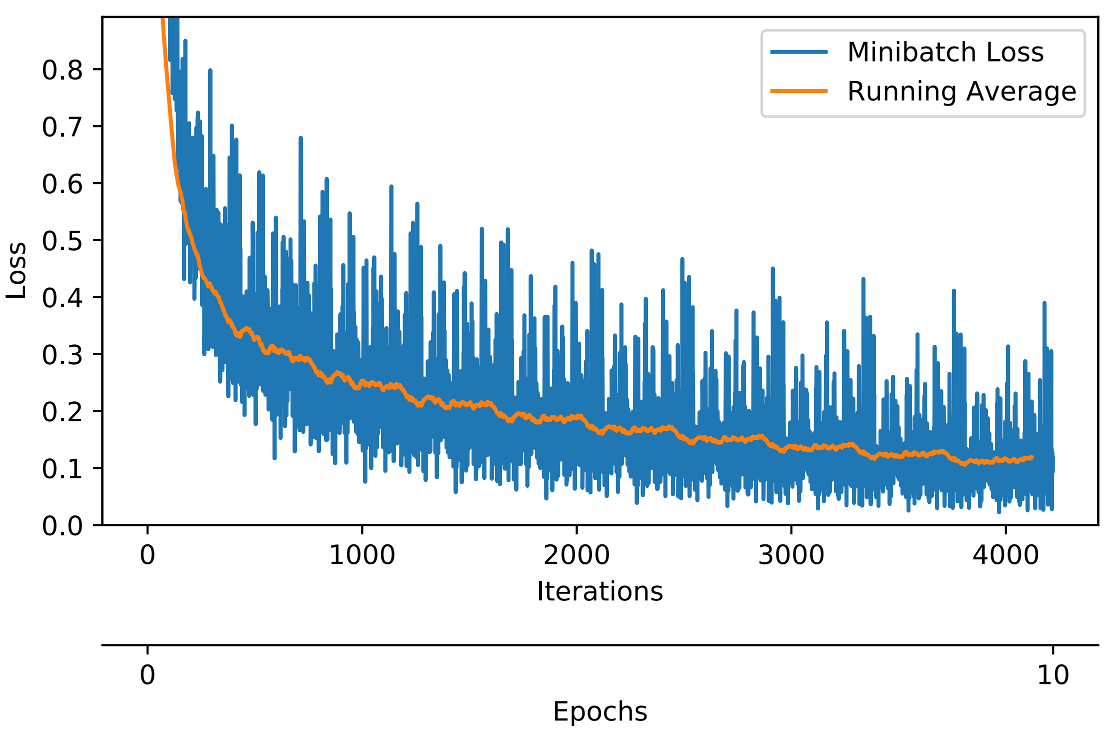
- How would a random predictor perform (especially in classification problems)? Dataset can be unbalanced…
- What would the loss look like for a random predictor?
- What is (are) the best metric(s) to measure progress on my task?
- What are the limits of this metric? If it’s perfect, what can I conclude? What can’t I conclude?
- What is missing in “simple approaches” to reach a perfect score?
- Are there architectures in my neural network toolbox that would be good to model the inductive bias of the data?

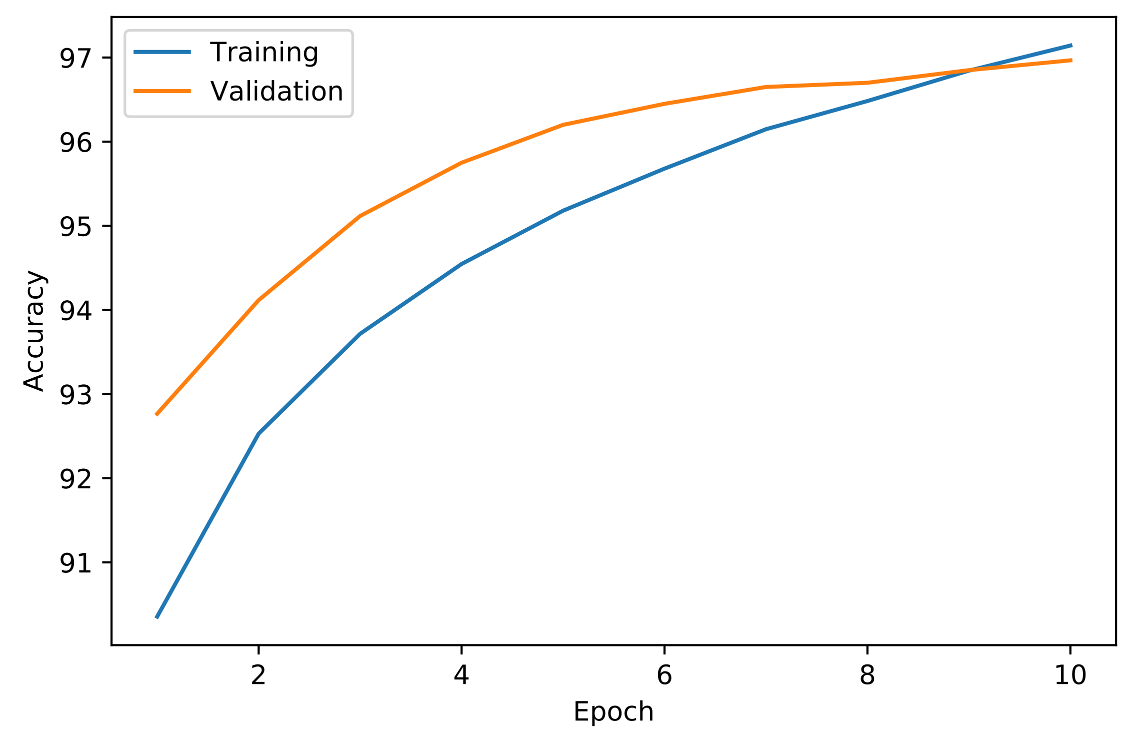
3) Model debugging

Try to overfit a small batch of examples (16 for instance) and get 0-loss. If not possible, there may be a bug.

- You forgot to call **model.eval()** in evaluation mode (in PyTorch) or **model.zero_grad()** to clean the gradients
- Something went wrong in the pre-processing of the inputs
- The loss got wrong arguments (for instance passing probabilities when it expects logits)
- Initialization doesn't break the symmetry (usually happens when you initialize a whole matrix with a single constant value)
- Some parameters are never called during the forward pass (and thus receive no gradients)
- The learning rate is taking funky values like 0 all the time
- Your inputs are being truncated in a suboptimal way

Plot loss curves





Simple considerations for simple people building fancy neural networks

Published February 25, 2021 – Initially published on Medium, Sept 2020.

Update on GitHub

VictorSanh
Victor Sanh

https://huggingface.co/blog/simple-considerations

4. Tune but don’t tune blindly

I generally stick with a random grid search as it turns out to be fairly effective in practice. *Some people report successes using fancy hyperparameter tuning methods such as Bayesian optimization but in my experience, random over a reasonably manually defined grid search is still a tough-to-beat baseline.*

compare a couple of runs with different hyperparameters to get an idea of which hyperparameters have the highest impact

favor (as most as possible) a deep understanding of each component of your neural network instead of blindly (not to say magically) tweak the architecture.

arXiv.org > cs > arXiv:2103.01209

Computer Science > Computer Vision and Pattern Recognition

[Submitted on 1 Mar 2021 (v1), last revised 2 Mar 2021 (this version, v2)]

Generative Adversarial Transformers

Drew A. Hudson, C. Lawrence Zitnick

https://arxiv.org/abs/2103.01209

https://github.com/dorarad/gansformer

Figure 1. Sample images generated by the GANsformer, along with a visualization of the model attention maps.

	CLEVR				LSUN-Bedroom
Model	FID ↓	IS ↑	Precision ↑	Recall ↑	FID ↓
GAN	25.0244	2.1719	21.77	16.76	12.1567
k-GAN	28.2900	2.2097	22.93	18.43	69.9014
SAGAN	26.0433	2.1742	30.09	15.16	14.0595
StyleGAN2	16.0534	2.1472	28.41	23.22	11.5255
VQGAN	32.6031	2.0324	46.55	63.33	59.6333
GANsformer$_s$	10.2585	**2.4555**	38.47	37.76	8.5551
GANsformer$_d$	**9.1679**	2.3654	**47.55**	**66.63**	**6.5085**

	FFHQ				Cityscapes
Model	FID ↓	IS ↑	Precision ↑	Recall ↑	FID ↓
GAN	13.1844	4.2966	67.15	17.64	11.5652
k-GAN	61.1426	3.9990	50.51	0.49	51.0804
SAGAN	16.2069	4.2570	64.84	12.26	12.8077
StyleGAN2	**10.8309**	4.3294	68.61	25.45	8.3500
VQGAN	63.1165	2.2306	67.01	**29.67**	173.7971
GANsformer$_s$	13.2861	**4.4591**	**68.94**	10.14	14.2315
GANsformer$_d$	12.8478	4.4079	68.77	5.7589	**5.7589**