pytorch - GPU & Transformers

Lecture 26

Dr. Colin Rundel

CUDA

> CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing unit (GPU) for general purpose processing, an approach called general-purpose computing on GPUs (GPGPU). CUDA is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

Core libraries:

- cuBLAS
- cuSOLVER
- cuSPARSE
- cuTENSOR
- cuFFT
- cuRAND
- Thrust
- cuDNN

CUDA Kernels

```
// Kernel - Adding two matrices MatA and MatB
__global__ void MatAdd(float MatA[N][N], float MatB[N][N], float MatC[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        MatC[i][j] = MatA[i][j] + MatB[i][j];
}

int main()
{
    ...
    // Matrix addition kernel launch from host code
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(
        (N + threadsPerBlock.x -1) / threadsPerBlock.x,
        (N+threadsPerBlock.y -1) / threadsPerBlock.y
    );

    MatAdd<<<numBlocks, threadsPerBlock>>>(MatA, MatB, MatC);
    ...
}
```

GPU Status

```
1 nvidia-smi
```

```
Tue Apr 14 09:11:37 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 590.48.01              Driver Version: 590.48.01      CUDA Version: 13.1     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA RTX A4000               Off |   00000000:01:00.0 Off |                  Off |
| 41%   35C    P8             14W /  140W |    2699MiB /  16376MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA RTX A4000               Off |   00000000:68:00.0 Off |                  Off |
| 41%   37C    P8             13W /  140W |       4MiB /  16376MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
```

Torch GPU Information

```
1 torch.cuda.is_available()
```

```
True
```

```
1 torch.cuda.device_count()
```

```
2
```

```
1 torch.cuda.get_device_name("cuda:0")
```

```
'NVIDIA RTX A4000'
```

```
1 torch.cuda.get_device_name("cuda:1")
```

```
'NVIDIA RTX A4000'
```

```
1 torch.cuda.get_device_properties(0)
```

```
_CudaDeviceProperties(name='NVIDIA RTX A4000', major=8, minor=6, total_memory=15976MB, multi_proce
```

GPU Tensors

Usage of the GPU is governed by the location of the Tensors - to use the GPU we allocate them on the GPU device.

```
1 cpu = torch.device('cpu')
2 cuda0 = torch.device('cuda:0')
3 cuda1 = torch.device('cuda:1')
```

```
1 x = torch.linspace(0,1,5, device=cuda0); x
```

```
tensor([0.0000, 0.2500, 0.5000, 0.7500,
        1.0000], device='cuda:0')
```

```
1 y = torch.randn(5,2, device=cuda0); y
```

```
tensor([[-3.3676,  0.4284],
        [ 1.0317, -0.3296],
        [-1.6607, -0.5143],
        [-0.7477, -0.5811],
        [ 1.3505,  0.1542]], device='cuda:0')
```

```
1 z = torch.rand(2,3, device=cpu); z
```

```
tensor([[0.8394, 0.5169, 0.5356],
        [0.5584, 0.5428, 0.0929]])
```

```
1 x @ y
```

```
tensor([ 0.2173, -0.6211], device='cuda:0')
```

```
1 y @ z
```

```
RuntimeError: Expected all tensors to be on the
```

```
1 y @ z.to(cuda0)
```

```
tensor([[-2.5874, -1.5080, -1.7639],
        [ 0.6819,  0.3544,  0.5219],
        [-1.6811, -1.1375, -0.9373],
        [-0.9521, -0.7019, -0.4545],
        [ 1.2197,  0.7817,  0.7377]],
       device='cuda:0')
```

NN Layers + GPU

NN layers (parameters) also need to be assigned to the GPU to be used with GPU tensors,

```
1 nn = torch.nn.Linear(5,5)
2 X = torch.randn(10,5).cuda()
```

```
1 nn(X)
```

```
RuntimeError: Expected all tensors to be on the same device, but got mat1 is on cuda:0, different
```

```
1 nn.cuda()(X)
```

```
tensor([[ 0.0665, -0.1522, -0.1733,  0.4627,
         -0.3899],
        [ 0.4550,  0.2700, -0.8572,  0.0812,
          0.5769],
        [-0.1594,  0.1351, -0.3831, -0.0761,
          0.4204],
        [-0.5051,  0.6921,  0.0977, -0.0305,
          0.3425],
        [-0.9259,  0.9657, -0.2817,  1.1706,
         -0.3354],
        [ 0.8163, -0.4542,  0.6201,  0.4417,
         -0.9929],
        [ 0.7821, -0.2014,  1.0838, -1.0487,
          0.1281],
        [-0.1962,  0.5809,  0.0408,  0.9464,
         -0.1634],
        [-0.1362,  0.3502, -0.5095,  0.3304,
         -0.1045],
        [ 0.4283,  0.1325,  0.3059, -0.9481,
          0.5690]], device='cuda:0',
       grad_fn=<AddmmBackward0>)
```

```
1 nn.to(device="cuda")(X)
```

```
tensor([[ 0.0665, -0.1522, -0.1733,  0.4627,
         -0.3899],
        [ 0.4550,  0.2700, -0.8572,  0.0812,
          0.5769],
        [-0.1594,  0.1351, -0.3831, -0.0761,
          0.4204],
        [-0.5051,  0.6921,  0.0977, -0.0305,
          0.3425],
        [-0.9259,  0.9657, -0.2817,  1.1706,
         -0.3354],
        [ 0.8163, -0.4542,  0.6201,  0.4417,
         -0.9929],
        [ 0.7821, -0.2014,  1.0838, -1.0487,
          0.1281],
        [-0.1962,  0.5809,  0.0408,  0.9464,
         -0.1634],
        [-0.1362,  0.3502, -0.5095,  0.3304,
         -0.1045],
        [ 0.4283,  0.1325,  0.3059, -0.9481,
          0.5690]], device='cuda:0',
       grad_fn=<AddmmBackward0>)
```

CIFAR10

homepage

Image Classification Benchmarks

CIFAR-10

- 60,000 color images (32×32)
- 10 classes (airplane, car, bird, cat, deer, dog, frog, horse, ship, truck)
- 50k train / 10k test
- ~170 MB

CIFAR-100

- 60,000 color images (32×32)
- 100 classes grouped into 20 superclasses
- 50k train / 10k test
- ~170 MB

ImageNet (ILSVRC)

- ~1.2 million color images (variable, typically 224×224)
- 1,000 classes
- ~1.2M train / 50k val / 100k test
- ~150 GB

All three are standard benchmarks for evaluating CNN architectures — CIFAR-10/100 are common for rapid prototyping due to their small image size, while ImageNet is the large-scale challenge used to evaluate models (AlexNet, VGG, ResNet, etc.).

Loading the data

```
import torchvision
training_data = torchvision.datasets.CIFAR10(
    root="/data",
    train=True,
    download=True,
    transform=torchvision.transforms.ToTensor()
)
test_data = torchvision.datasets.CIFAR10(
    root="/data",
    train=False,
    download=True,
    transform=torchvision.transforms.ToTensor()
)
```

Downloads data to “/data/cifar-10-batches-py” which is ~178M on disk.

CIFAR10 data

```
1 training_data.classes
```

```
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```

```
1 training_data.data.shape
```

```
(50000, 32, 32, 3)
```

```
1 test_data.data.shape
```

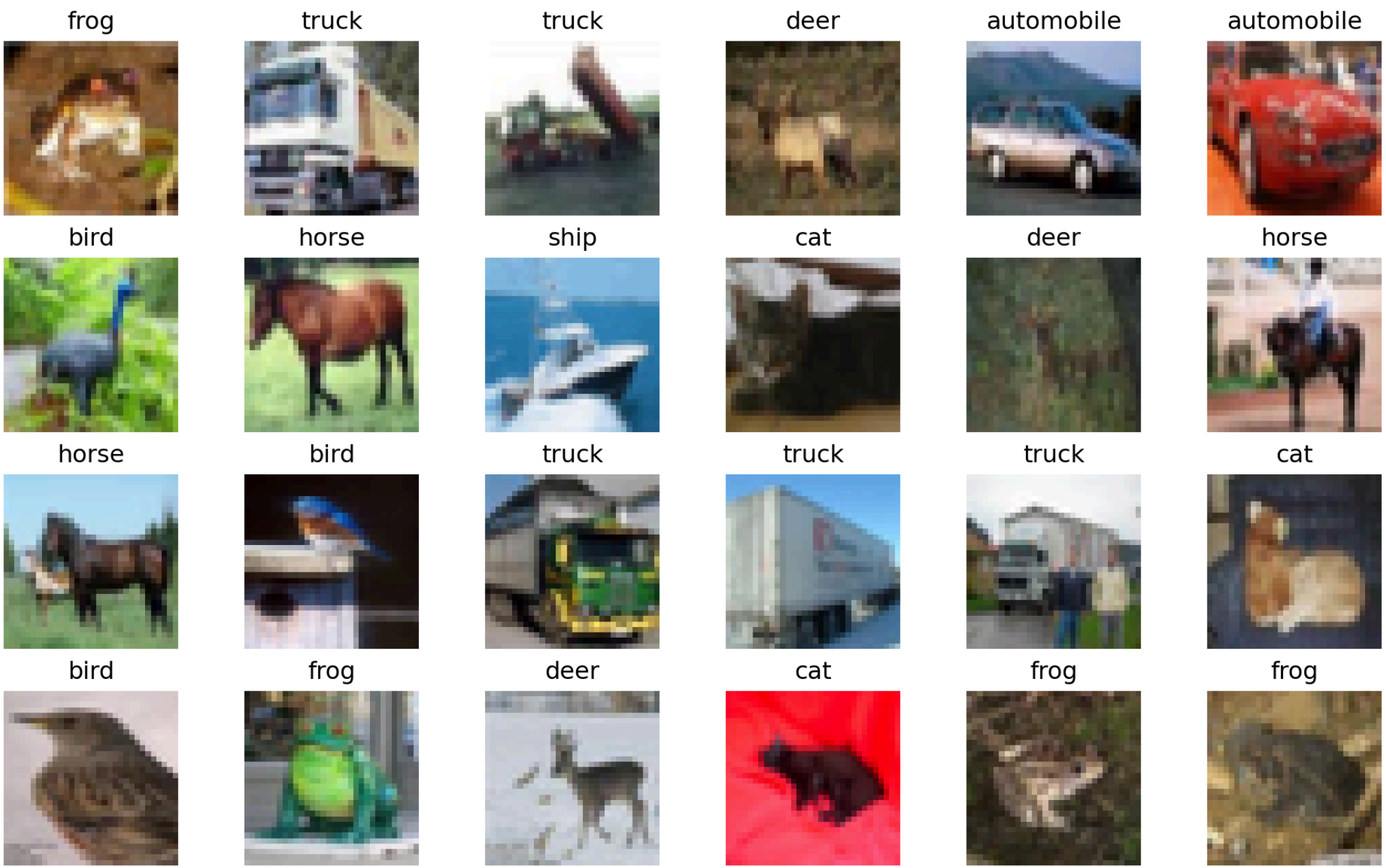
```
(10000, 32, 32, 3)
```

```
1 training_data[0]
```

```
(tensor([[[0.2314, 0.1686, 0.1961, 0.2667,
          0.3843, 0.4667, 0.5451, 0.5686,
          0.5843, 0.5843, 0.5137, 0.4902,
          0.5569, 0.5647, 0.5373, 0.5059,
          0.5373, 0.5255, 0.4863, 0.5451,
          0.5451, 0.5216, 0.5333, 0.5451,
          0.5961, 0.6392, 0.6588, 0.6235,
          0.6196, 0.6196, 0.5961, 0.5804],
         [0.0627, 0.0000, 0.0706, 0.2000,
          0.3451, 0.4706, 0.5020, 0.4980,
          0.4941, 0.4549, 0.4157, 0.3961,
          0.4118, 0.4431, 0.4275, 0.4392,
          0.4667, 0.4275, 0.4118, 0.4902,
          0.4980, 0.4784, 0.5137, 0.4863,
          0.4745, 0.5137, 0.5176, 0.5216,
          0.5216, 0.4824, 0.4667, 0.4784],
```

```
[0.0980, 0.0627, 0.1922, 0.3255,
 0.4314, 0.5059, 0.5098, 0.4745,
 0.4431, 0.4392, 0.4392, 0.4157,
 0.4118, 0.5020, 0.4863, 0.5098,
 0.4980, 0.4784, 0.4510, 0.4706,
 0.5098, 0.5137, 0.5451, 0.4980,
```

Example data



Data Loaders

Torch handles large datasets and minibatches through the use of the `DataLoader` class,

```
training_loader = torch.utils.data.DataLoader(
    training_data,
    batch_size=100,
    shuffle=True,
    num_workers=4,
    pin_memory=True
)

test_loader = torch.utils.data.DataLoader(
    test_data,
    batch_size=100,
    shuffle=True,
    num_workers=4,
    pin_memory=True
)
```

Loader as generator

The resulting `DataLoader` class is iterable and "yields" the features and targets for each batch when iterated over,

```
1 training_loader
```

```
<torch.utils.data.dataloader.DataLoader object at 0x7f2d82a3e900>
```

```
1 X, y = next(iter(training_loader))
2 X.shape
```

```
torch.Size([100, 3, 32, 32])
```

```
1 y.shape
```

```
torch.Size([100])
```

Custom Datasets

In this case we got our data (`training_data` and `test_data`) directly from torchvision which gave us a `dataset` object to use with our `DataLoader`. If we do not have a `Dataset` object then we need to create a custom class for our data telling torch how to load it.

Your class must define the methods: `__init__()`, `__len__()`, and `__getitem__()`.

```
class data(torch.utils.data.Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

CIFAR CNN

```
class cifar_conv_model(torch.nn.Module):
    def __init__(self, device):
        super().__init__()
        self.device = torch.device(device)
        self.epoch = 0
        self.model = torch.nn.Sequential(
            torch.nn.Conv2d(3, 6, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(2, 2),
            torch.nn.Conv2d(6, 16, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(2, 2),
            torch.nn.Flatten(),
            torch.nn.Linear(16 * 5 * 5, 120),
            torch.nn.ReLU(),
            torch.nn.Linear(120, 84),
            torch.nn.ReLU(),
            torch.nn.Linear(84, 10)
        ).to(device=self.device)

    def forward(self, X):
        return self.model(X)
```

CNN Performance

CPU - 1 step | GPU - 1 step | CPU - 1 epoch | GPU - 1 epoch

```python
X, y = next(iter(training_loader))

m_cpu = cifar_conv_model(device="cpu")
tmp = m_cpu(X)

with torch.autograd.profiler.profile(with_stack=True) as prof_cpu:
    tmp = m_cpu(X)
```

```python
print(prof_cpu.key_averages().table(sort_by='self_cpu_time_total', row_limit=10))
```

```
---------------------------------  ------------  ------------  ------------  ------------  ------------  ------------
                             Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg    # of Calls
---------------------------------  ------------  ------------  ------------  ------------  ------------  ------------
         aten::mkldnn_convolution        44.61%     552.034us        45.78%     566.422us     283.211us             2
    aten::max_pool2d_with_indices        31.12%     385.069us        31.12%     385.069us     192.535us             2
                 aten::clamp_min         9.45%     116.901us         9.45%     116.901us      29.225us             4
                     aten::addmm         6.38%      78.921us         7.59%      93.919us      31.306us             3
                aten::convolution         1.41%      17.472us        47.93%     593.023us     296.511us             2
                       aten::relu         1.06%      13.144us        10.51%     130.045us      32.511us             4
                     aten::copy_          0.91%      11.272us         0.91%      11.272us       3.757us             3
               aten::_convolution         0.74%       9.129us        46.52%     575.551us     287.775us             2
                      aten::empty         0.65%       8.076us         0.65%       8.076us       2.019us             4
                       aten::view         0.57%       7.064us         0.57%       7.064us       7.064us             1
---------------------------------  ------------  ------------  ------------  ------------  ------------  ------------
Self CPU time total: 1.237ms
```

Loaders & Accuracy

```
def accuracy(model, loader, device):
    total, correct = 0, 0
    with torch.no_grad():
        for X, y in loader:
            X, y = X.to(device=device), y.to(device=device)
            pred = model(X)
            # the class with the highest energy is what we choose as prediction
            val, idx = torch.max(pred, 1)
            total += pred.size(0)
            correct += (idx == y).sum().item()

    return correct / total
```

Model fitting

```
1 m = cifar_conv_model("cuda")
```

Round 1 | Round 2 | Round 3

```
1 m.fit(training_loader, epochs=10, n_report=500, lr=0.01)
```

```
## [Epoch 1, Minibatch  500] loss: 2.098
## [Epoch 2, Minibatch  500] loss: 1.692
## [Epoch 3, Minibatch  500] loss: 1.482
## [Epoch 4, Minibatch  500] loss: 1.374
## [Epoch 5, Minibatch  500] loss: 1.292
## [Epoch 6, Minibatch  500] loss: 1.226
## [Epoch 7, Minibatch  500] loss: 1.173
## [Epoch 8, Minibatch  500] loss: 1.117
## [Epoch 9, Minibatch  500] loss: 1.071
## [Epoch 10, Minibatch  500] loss: 1.035
```

```
1 accuracy(m, training_loader, "cuda")
```

```
## 0.63444
```

```
1 accuracy(m, test_loader, "cuda")
```

```
## 0.572
```

The VGG16 model

```
class VGG16(torch.nn.Module):
    def make_layers(self):
        cfg = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']
        layers = []
        in_channels = 3
        for x in cfg:
            if x == 'M':
                layers += [torch.nn.MaxPool2d(kernel_size=2, stride=2)]
            else:
                layers += [torch.nn.Conv2d(in_channels, x, kernel_size=3, padding=1),
                           torch.nn.BatchNorm2d(x),
                           torch.nn.ReLU(inplace=True)]
                in_channels = x
        layers += [
            torch.nn.AvgPool2d(kernel_size=1, stride=1),
            torch.nn.Flatten(),
            torch.nn.Linear(512,10)
        ]

        return torch.nn.Sequential(*layers).to(self.device)

    def __init__(self, device):
        super().__init__()
        self.device = torch.device(device)
```

Model

```
1 VGG16("cpu").model
```

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (5): ReLU(inplace=True)
  (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (9): ReLU(inplace=True)
  (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (12): ReLU(inplace=True)
  (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (16): ReLU(inplace=True)
  (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (19): ReLU(inplace=True)
  (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

VGG16 performance

CPU - 1 forward step | GPU - 1 forward step

```python
X, y = next(iter(training_loader))
m_cpu = VGG16(device="cpu")
tmp = m_cpu(X)

with torch.autograd.profiler.profile(with_stack=True) as prof_cpu:
    tmp = m_cpu(X)
```

```python
print(prof_cpu.key_averages().table(sort_by='self_cpu_time_total', row_limit=10))
```

```
-------------------------------  ------------  ------------  ------------  ------------  -------
                           Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU tim
-------------------------------  ------------  ------------  ------------  ------------  -------
        aten::mkldnn_convolution        77.21%      53.228ms        77.45%      53.398ms       4.
         aten::native_batch_norm        11.68%       8.050ms        11.86%       8.175ms     628.
    aten::max_pool2d_with_indices         7.60%       5.241ms         7.60%       5.241ms       1.
                aten::clamp_min_         1.91%       1.315ms         1.91%       1.315ms     101.
                     aten::empty         0.33%     227.335us         0.33%     227.335us       1.
                      aten::add_         0.22%     151.716us         0.22%     151.716us      11.
                aten::convolution         0.21%     145.065us        77.81%      53.647ms       4.
                     aten::relu_         0.20%     137.109us         2.11%       1.452ms     111.
               aten::_convolution         0.15%     103.739us        77.60%      53.502ms       4.
     aten::_batch_norm_impl_index         0.12%      83.299us        11.99%       8.264ms     635.
-------------------------------  ------------  ------------  ------------  ------------  -------
Self CPU time total: 68.941ms
```

Fitting

lr = 0.01 | lr = 0.001

```
m = VGG16(device="cuda")
fit(m, training_loader, epochs=10, n_report=500, lr=0.01)
```

```
## [Epoch 1, Minibatch  500] loss: 1.345
## [Epoch 2, Minibatch  500] loss: 0.790
## [Epoch 3, Minibatch  500] loss: 0.577
## [Epoch 4, Minibatch  500] loss: 0.445
## [Epoch 5, Minibatch  500] loss: 0.350
## [Epoch 6, Minibatch  500] loss: 0.274
## [Epoch 7, Minibatch  500] loss: 0.215
## [Epoch 8, Minibatch  500] loss: 0.167
## [Epoch 9, Minibatch  500] loss: 0.127
## [Epoch 10, Minibatch  500] loss: 0.103
```

```
accuracy(model=m, loader=training_loader, device="cuda")
```

```
## 0.97008
```

```
accuracy(model=m, loader=test_loader, device="cuda")
```

```
## 0.8318
```

Report

Code | Results

```
from sklearn.metrics import classification_report

def report(model, loader, device):
    y_true, y_pred = [], []
    with torch.no_grad():
        for X, y in loader:
            X = X.to(device=device)
            y_true.append( y.cpu().numpy() )
            y_pred.append( model(X).max(1)[1].cpu().numpy() )

    y_true = np.concatenate(y_true)
    y_pred = np.concatenate(y_pred)

    return classification_report(y_true, y_pred, target_names=loader.dataset.classes)
```

Transformers & GPT

Attribution & Resources

The following slides are a very brief and simplified look at a “modern” GPT model using Torch.

The code comes from Andrej Karpathy’s minGPT repo.

> A PyTorch re-implementation of GPT, both training and inference. minGPT tries to be small, clean, interpretable and educational, as most of the currently available GPT model implementations can be a bit sprawling.

It is an impressive piece of work and well worth looking through. More recently, Karpathy put together microgpt which is a pure-Python follow-up with no dependencies (including full autograd).

I would also recommend growingswe.com/blog/microgpt — an excellent explainer and companion that walks through the microGPT code in detail.

Language Models

A language model assigns a probability to a sequence of tokens:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1})$$

Training objective: given the previous tokens, predict the next one.

- Tokens can be words, sub-words, characters, …
- At each step the model sees all *prior* context (but not future tokens)
- Loss is cross-entropy averaged over all positions

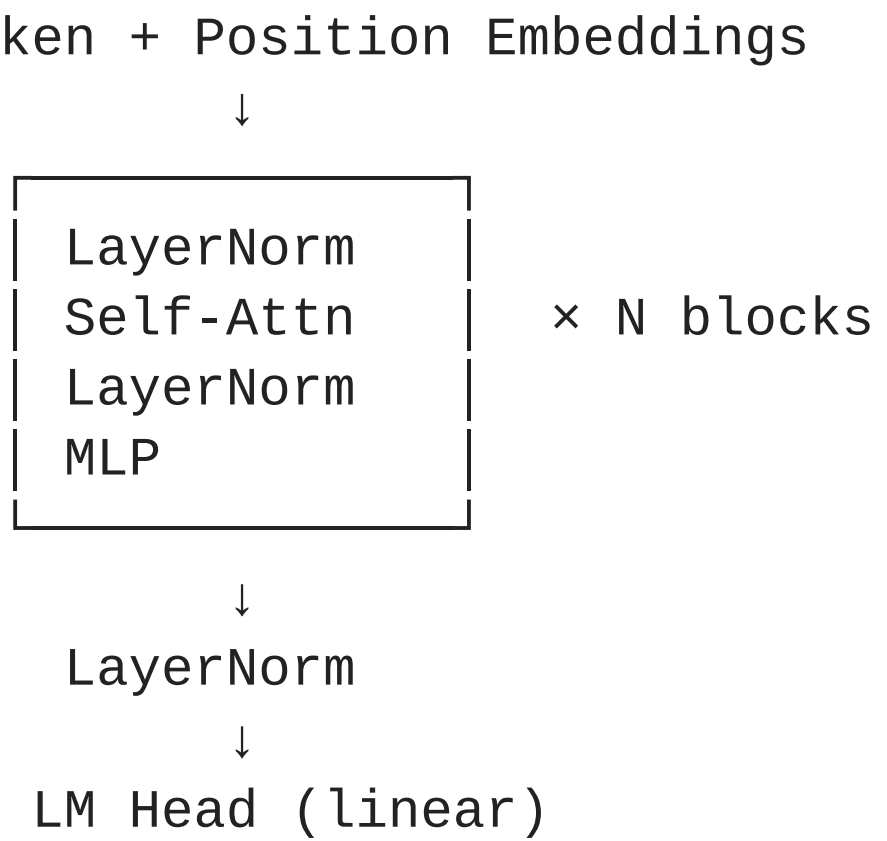
Transformer Architecture

Introduced in *Attention Is All You Need* (Vaswani et al., 2017).

Key ideas:

- Replace recurrence with self-attention — every token attends to every other token in a single step
- Stack N identical transformer blocks
- Add positional encodings so the model knows token order
- Scale to billions of parameters via the same architecture

GPT (decoder-only) stack:



Causal (Masked) Self-Attention

The core operation — each token produces a *query*, *key*, and *value*:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V$$

where M is a causal mask (upper-triangle $= -\infty$) that prevents attending to future positions.

With h heads, each head learns a different projection:

$$\text{head}_i = \text{Attention}(QW_i^Q,\ KW_i^K,\ VW_i^V), \quad W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$$

outputs are concatenated and projected back:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W^O$$

Implementation - `CausalSelfAttention`

```python
class CausalSelfAttention(nn.Module):
    def __init__(self, n_embd, n_head, block_size, dropout=0.1):
        super().__init__()
        assert n_embd % n_head == 0
        self.n_head, self.n_embd = n_head, n_embd
        self.c_attn  = nn.Linear(n_embd, 3 * n_embd)   # Q, K, V in one shot
        self.c_proj  = nn.Linear(n_embd, n_embd)
        self.attn_drop  = nn.Dropout(dropout)
        self.resid_drop = nn.Dropout(dropout)
        # lower-triangular causal mask  (1, 1, T, T)
        mask = torch.tril(torch.ones(block_size, block_size)).view(1, 1, block_size, block_si
        self.register_buffer("mask", mask)

    def forward(self, x):
        B, T, C = x.size()
        hs = C // self.n_head
        q, k, v = self.c_attn(x).split(self.n_embd, dim=2)          # (B, T, C) each
        # reshape to (B, n_head, T, head_size)
        q = q.view(B, T, self.n_head, hs).transpose(1, 2)
        k = k.view(B, T, self.n_head, hs).transpose(1, 2)
        v = v.view(B, T, self.n_head, hs).transpose(1, 2)
        # scaled dot-product attention with causal mask
```

Implementation - `Block`

Each transformer block wraps attention + a feed-forward MLP with residual connections and layer norm:

```
class Block(nn.Module):
    def __init__(self, n_embd, n_head, block_size, dropout=0.1):
        super().__init__()
        self.ln1  = nn.LayerNorm(n_embd)
        self.attn = CausalSelfAttention(n_embd, n_head, block_size, dropout)
        self.ln2  = nn.LayerNorm(n_embd)
        self.mlp  = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.GELU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        x = x + self.attn(self.ln1(x))   # attention  + residual
        x = x + self.mlp(self.ln2(x))    # feed-forward + residual
        return x
```

Implementation - GPT

```python
class GPT(nn.Module):
    def __init__(self, vocab_size, block_size, n_layer=6, n_head=6, n_embd=192, dropout=0.1):
        super().__init__()
        self.block_size = block_size
        self.transformer = nn.ModuleDict(dict(
            wte  = nn.Embedding(vocab_size, n_embd),             # token embeddings
            wpe  = nn.Embedding(block_size, n_embd),             # position embeddings
            drop = nn.Dropout(dropout),
            h    = nn.ModuleList([Block(n_embd, n_head, block_size, dropout)
                                  for _ in range(n_layer)]),
            ln_f = nn.LayerNorm(n_embd),
        ))
        self.lm_head = nn.Linear(n_embd, vocab_size, bias=False)
        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, (nn.Linear, nn.Embedding)):
            nn.init.normal_(m.weight, std=0.02)
            if hasattr(m, 'bias') and m.bias is not None:
                nn.init.zeros_(m.bias)

    def forward(self, idx, targets=None):
```

Model Sizes

Variant	Layers	Heads	Embed dim	~Params
gpt-nano	3	3	48	45 K
gpt-micro	4	4	128	660 K
gpt-mini	6	6	192	3 M
GPT-2 small	12	12	768	124 M
GPT-2 medium	24	16	1024	350 M
GPT-2 XL	48	25	1600	1.6 B

For our example use case, character-level name generation, we only need a tiny model — the vocabulary is ~27 characters and sequences are short.

Character-level Name Generation

The Names Dataset

This is a common classic benchmark for character-level language models — ~32k US baby names, one per line.

```
import urllib.request
url = "https://raw.githubusercontent.com/karpathy/makemore/master/names.txt"
names_txt = urllib.request.urlopen(url).read().decode()
names = names_txt.strip().split('\n')
len(names)
```

```
32033
```

```
names[:10]
```

```
['emma', 'olivia', 'ava', 'isabella', 'sophia', 'charlotte', 'mia', 'amelia', 'harper'
```

- Each name is terminated by `\n`

  — the model learns `\n` -> start of a name *and* name -> `\n`
- Character vocabulary - 26 letters + newline = 27 tokens
- Mean name length - 6.1 characters

`CharDataset`

Each training example is a window of `block_size` characters; the target is the same window shifted by one:

```python
class CharDataset(Dataset):
    def __init__(self, text, block_size):
        chars = sorted(set(text))
        self.stoi = {c: i for i, c in enumerate(chars)}
        self.itos = {i: c for i, c in enumerate(chars)}
        self.vocab_size = len(chars)
        self.block_size = block_size
        self.data = torch.tensor([self.stoi[c] for c in text], dtype=torch.long)

    def __len__(self):
        return len(self.data) - self.block_size

    def __getitem__(self, idx):
        chunk = self.data[idx : idx + self.block_size + 1]
        return chunk[:-1], chunk[1:]   # (x, y) shifted by 1
```

```
BLOCK_SIZE = 32
dataset = CharDataset(names_txt, block_size=BLOCK_SIZE)
```

```
x, y = dataset[0]
```

```
x
```

```
tensor([ 5, 13, 13,  1,  0, 15, 12,  9, 22,
         9,  1,  0,  1, 22,  1,  0,  9, 19,
         1,  2,  5, 12, 12,  1,  0, 19, 15,
        16,  8,  9,  1,  0])
```

```
''.join(dataset.itos[i.item()] for i in x)
```

```
'emma\nolivia\nava\nisabella\nsophia\n'
```

```
y
```

```
tensor([13, 13,  1,  0, 15, 12,  9, 22,  9,
         1,  0,  1, 22,  1,  0,  9, 19,  1,
         2,  5, 12, 12,  1,  0, 19, 15, 16,
         8,  9,  1,  0,  3])
```

```
''.join(dataset.itos[i.item()] for i in y)
```

```
'mma\nolivia\nava\nisabella\nsophia\nc'
```

```
x, y = dataset[1]
```

```
x
```

```
tensor([13, 13,  1,  0, 15, 12,  9, 22,  9,
         1,  0,  1, 22,  1,  0,  9, 19,  1,
         2,  5, 12, 12,  1,  0, 19, 15, 16,
         8,  9,  1,  0,  3])
```

```
''.join(dataset.itos[i.item()] for i in x)
```

```
'mma\nolivia\nava\nisabella\nsophia\nc'
```

```
y
```

```
tensor([13,  1,  0, 15, 12,  9, 22,  9,  1,
         0,  1, 22,  1,  0,  9, 19,  1,  2,
         5, 12, 12,  1,  0, 19, 15, 16,  8,
         9,  1,  0,  3,  8])
```

```
''.join(dataset.itos[i.item()] for i in y)
```

```
'ma\nolivia\nava\nisabella\nsophia\nch'
```

Training Setup

```
BLOCK_SIZE = 32
dataset = CharDataset(names_txt, block_size=BLOCK_SIZE)
loader  = DataLoader(dataset, batch_size=256, shuffle=True)

model = GPT(
    vocab_size  = dataset.vocab_size,
    block_size  = BLOCK_SIZE,
    n_layer     = 4,
    n_head      = 4,
    n_embd      = 128,
    dropout     = 0.1,
).to(device)

optimizer = torch.optim.AdamW(model.parameters(), lr=3e-3, weight_decay=0.1)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=5000)
```

Model parameter count:

```
sum(p.numel() for p in model.parameters() if p.requires_grad)
```

```
804352
```

`CosineAnnealingLR` decays the learning rate from `lr_max` to 0 following a cosine curve over `T_max` steps — starts fast,

Training

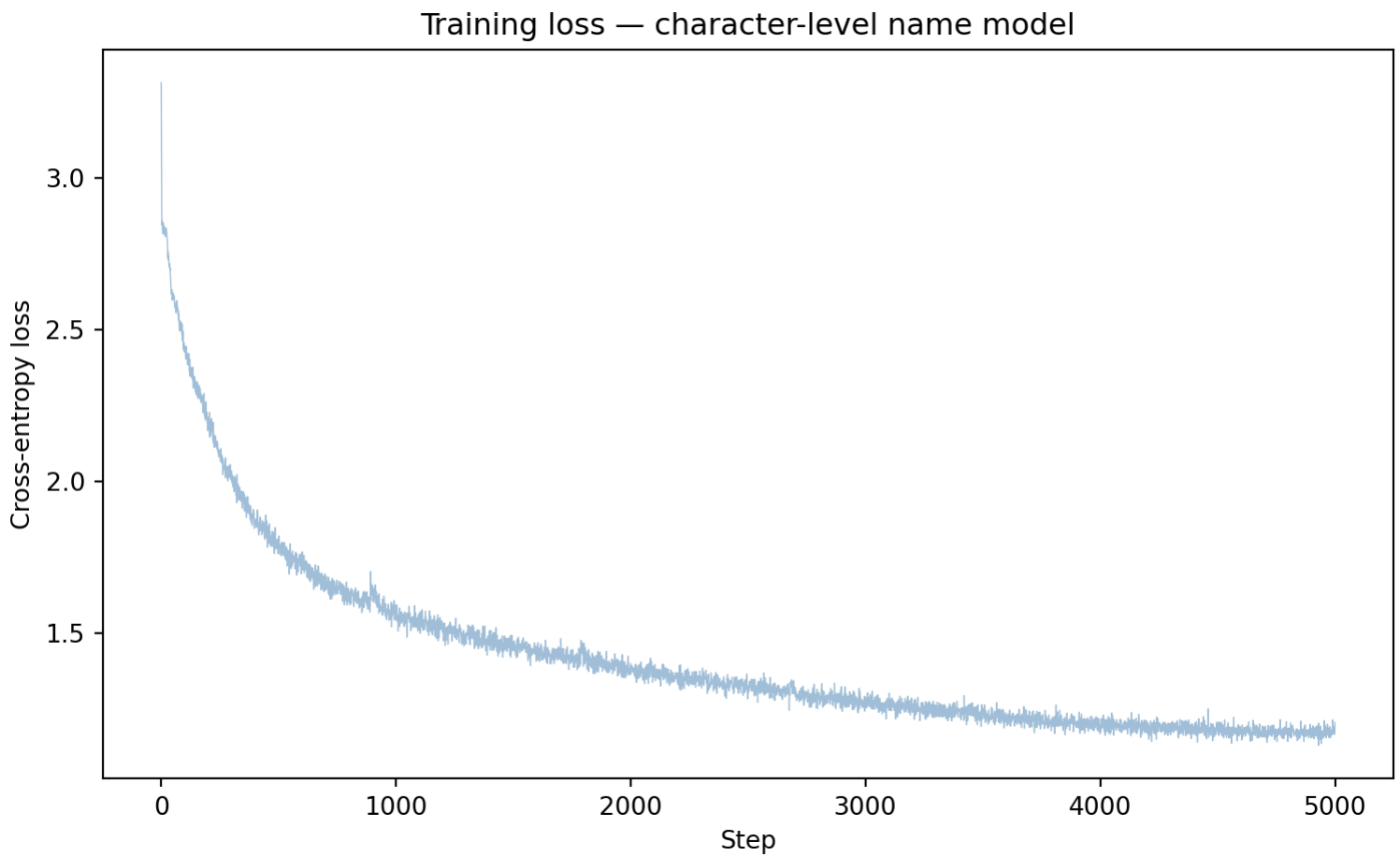
```
1 model.train()
```

```
GPT(
  (transformer): ModuleDict(
    (wte): Embedding(27, 128)
    (wpe): Embedding(32, 128)
    (drop): Dropout(p=0.1, inplace=False)
    (h): ModuleList(
      (0-3): 4 x Block(
        (ln1): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
        (attn): CausalSelfAttention(
          (c_attn): Linear(in_features=128, out_features=384, bias=True)
          (c_proj): Linear(in_features=128, out_features=128, bias=True)
          (attn_drop): Dropout(p=0.1, inplace=False)
          (resid_drop): Dropout(p=0.1, inplace=False)
        )
        (ln2): LayerNorm((128,), eps=1e-05, elementwise_affine=True)
        (mlp): Sequential(
          (0): Linear(in_features=128, out_features=512, bias=True)
          (1): GELU(approximate='none')
          (2): Linear(in_features=512, out_features=128, bias=True)
          (3): Dropout(p=0.1, inplace=False)
        )
      )
```

```python
losses  = []
data_iter = iter(loader)

for step in range(5000):
    try:
        x, y = next(data_iter)
    except StopIteration:
        data_iter = iter(loader)
        x, y = next(data_iter)

    x, y = x.to(device), y.to(device)
    logits, loss = model(x, y)

    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    optimizer.step()
    scheduler.step()
    losses.append(loss.item())
```

Training Loss

Actual | Smoothed



Generating Names

```python
def sample_names(model, dataset, n=10, temperature=0.8, top_k=10):
    newline_idx = dataset.stoi['\n']
    start = torch.tensor([[newline_idx]], dtype=torch.long, device=device)
    names_out = []
    for _ in range(n):
        out = model.generate(start, max_new_tokens=20, temperature=temperature, top_k=top_k)
        tokens = out[0].tolist()
        # collect characters up to the next newline (after the seed)
        name = ''
        for t in tokens[1:]:
            c = dataset.itos[t]
            if c == '\n':
                break
            name += c
        if name:
            names_out.append(name)
    return names_out

sample_names(model, dataset)
```

```
['jahmari', 'kohler', 'montrell', 'edith', 'arleanne', 'derian', 'makala', 'josef', 'elianah', 'ka
```

Temperature & Sampling

`temperature=0.5` (more conservative):

```
1 sample_names(model, dataset, n=10, temperature=0.5)
```

```
['kenish', 'abdulrahman', 'danaya', 'kingston', 'adalei', 'alijah', 'solomi', 'ameena', 'maryelynn
```

`temperature=1.0` (default):

```
1 sample_names(model, dataset, n=10, temperature=1.0)
```

```
['asheal', 'lexander', 'caiman', 'scotland', 'adelia', 'milles', 'kyaira', 'loisie', 'darsha', 'se
```

`temperature=1.5` (more creative / noisy):

```
1 sample_names(model, dataset, n=10, temperature=1.5)
```

```
['jakoby', 'story', 'reyn', 'sunair', 'milealah', 'dmayah', 'jiles', 'elazar', 'dollan', 'emmalyn'
```

Lower temperature sharpens the distribution (more likely completions); higher temperature flattens it (more diverse but

Some “state-of-the-art” models

Hugging Face

This is an online community and platform for sharing machine learning models (architectures and weights), data, and related artifacts. They also maintain a number of packages and related training materials that help with building, training, and deploying ML models.

Some notable resources,

- `transformers` - APIs and tools to easily download and train state-of-the-art (pretrained) transformer based models
- `diffusers` - provides pretrained vision and audio diffusion models, and serves as a modular toolbox for inference and training

Stable Diffusion

```
from huggingface_hub import snapshot_download, login
```

```
model_id = "/data/stable-diffusion-2-1"
```

```
snapshot_download(
    repo_id="sd2-community/stable-diffusion-2-1",
    local_dir=model_id,
    ignore_patterns=["*.ckpt", "*.safetensors.index.json"],
    token="..." # or use login() ahead of time
)
```

```
import torch
from diffusers import StableDiffusionPipeline, DPMSolverMultistepScheduler

pipe = StableDiffusionPipeline.from_pretrained(model_id, torch_dtype=torch.float16)
```

```
pipe.scheduler = DPMSolverMultistepScheduler.from_config(pipe.scheduler.config)
pipe = pipe.to("cuda")
```

`sd2-community/stable-diffusion-2-1` is a gated repo — you must accept the model license on huggingface.co and be

Inference

Code | Images

```
prompt = "a picture of thomas bayes with a cat on his lap"
generator = [torch.Generator(device="cuda").manual_seed(i) for i in range(6)]
fit = pipe(prompt, generator=generator, num_inference_steps=20, num_images_per_prompt=6)
```

```
fit.images
```

```
[<PIL.Image.Image image mode=RGB size=768x768 at 0x7F2CEDB5FA10>, <PIL.Image.Image image mode=RGB
```

```
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(10, 6), layout="constrained")

for i, ax in enumerate([ax for row in axes for ax in row]):
    ax.set_axis_off()
    p = ax.imshow(fit.images[i])

plt.show()
```

Customizing prompts

Code | Images

```
prompt = "a picture of thomas bayes with a cat on his lap"
prompts = [
  prompt + t for t in
  ["in the style of a japanese wood block print",
   "as a hipster with facial hair and glasses",
   "as a simpsons character, cartoon, yellow",
   "in the style of a vincent van gogh painting",
   "in the style of a picasso painting",
   "with flowery wall paper"
  ]
]
```

```
generator = [torch.Generator(device="cuda").manual_seed(i) for i in range(6)]
fit = pipe(prompts, generator=generator, num_inference_steps=20, num_images_per_prompt=1)
```

Increasing inference steps

```
generator = [torch.Generator(device="cuda").manual_seed(i) for i in range(6)]
fit = pipe(prompts, generator=generator, num_inference_steps=50, num_images_per_prompt=1)
```

A more current model

This model is larger than the available GPU memory - so we adjust the weight types to make it fit.

```
from diffusers import BitsAndBytesConfig, SD3Transformer2DModel
from diffusers import StableDiffusion3Pipeline
import torch

model_id = "/data/stable-diffusion-3-5-medium"

nf4_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)
model_nf4 = SD3Transformer2DModel.from_pretrained(
    model_id,
    subfolder="transformer",
    quantization_config=nf4_config,
    torch_dtype=torch.bfloat16
)
pipe = StableDiffusion3Pipeline.from_pretrained(
    model_id,
    transformer=model_nf4,
    torch_dtype=torch.bfloat16
)
```

```
pipe.enable_model_cpu_offload()
```

Images

```
generator = [torch.Generator(device="cuda").manual_seed(i) for i in range(6)]
fit = pipe(prompts, generator=generator, num_inference_steps=30, num_images_per_prompt=1)
```