Just Write Better Code.

Are you using Nvidia GPUs from Volta, Turing, Ampere, Lovelace, Hopper, or Blackwell generation? Do you feel your GPU is not up to the speed for your training code then congratulations you might be entitled for more performance!!!!!! Call 1-800-NVDA-GPU for more.   
  
Let’s be real, the biggest problem with folks building neural networks isn’t that their hardware’s outdated—it’s that their code is trash. People get so caught up in writing what they think is optimized code, passing every unit, system, and integration test like they’ve won the lottery, but in reality, they’re leaving massive performance on the table. System design and optimization aren’t just side quests; they’re the damn main quest. So no, it’s not your hardware that’s holding you back, it’s your half-baked code. Time to face the music and stop pretending otherwise.

Oh, so you’re telling me your GPU is hitting 100% utilization while your training code runs? That’s cute. But seriously, do you even know the specs of your GPU? Yeah, sure, it’s got CUDA cores—everyone loves to throw that around like it’s the golden ticket to more performance. But let me ask you this: have you even heard of Tensor Cores? What’s that? You haven’t? Buddy, there’s more under the hood of your NVIDIA GPU than just CUDA cores, and if you’re not tapping into those Tensor Cores, you’re basically leaving money on the table. So, before you start bragging about your 100% utilization, maybe take a minute to learn what your hardware’s really capable of.

Alright, let’s dive into Tensor Cores like we’re cracking open a cold one. So, you’ve heard of CUDA Cores—those bad boys are like the bread and butter of GPU computing. But Tensor Cores? They’re the spicy secret sauce that takes your neural network training from 'meh' to 'hell yeah!'

Tensor Cores are specialized hardware units inside your NVIDIA GPU designed to handle matrix math at warp speed. And when I say matrix math, I’m talking about the heavy lifting that happens in deep learning, like multiplying and adding up a bunch of numbers at once. Normally, this stuff eats up a ton of compute power, but Tensor Cores? They chew through it like it’s nothing.

Here’s the kicker: Tensor Cores are built to work with mixed-precision training, meaning they juggle 16-bit floating-point numbers (half precision) and 32-bit floating-point numbers (full precision) without breaking a sweat. Why does this matter? Because it speeds up training while keeping your model accurate as hell. It’s like getting the best of both worlds—speed and precision, all in one go.

So, while your CUDA Cores are great for general-purpose tasks, Tensor Cores are like the nitrous boost that pushes your GPU into overdrive for deep learning tasks. If you’re not using them, you’re basically driving a sports car in first gear and wondering why you’re not winning any races. Get with the program and start using those Tensor Cores, because they’re the real MVPs in the world of AI.

Alright, listen up, kids. This is what most of you think is a top-tier, squeaky-clean, “optimized” training loop:

def train(data\_loader, model, criterion, optimizer, scheduler=None, early\_stop=None):

learning\_rate\_tracker = {}

epoch\_correct = 0

running\_loss = 0.0

model.train()

for i, (images, labels) in tqdm(enumerate(data\_loader), total=len(data\_loader), desc="Training"):

learning\_rate\_tracker[i] = optimizer.param\_groups[0]['lr']

images = images.to(device)

labels = labels.to(device)

optimizer.zero\_grad()

outputs = model(images)

loss = criterion(outputs, labels)

running\_loss += loss.item()

predicted = torch.max(outputs.data, 1)[1]

epoch\_correct += (predicted == labels).sum().item()

if early\_stop and i == early\_stop:

break

loss.backward()

optimizer.step()

if scheduler:

scheduler.step()

return epoch\_correct, running\_loss, learning\_rate\_tracker

But let me break it to you: **this is why only your mom thinks you’re doing a good job**. This code is the reason AI is giving climate change a run for its money. It's gonna hit a performance plateau faster than you can say "backpropagation." Don’t be this guy.

Why? Because this code is stuck in the past, leaving your shiny Tensor Cores twiddling their thumbs, and your GPU’s potential is just collecting dust. You think you’re optimized, but you’re really just cruising in the slow lane, burning more power than you need, and taking forever to get where you’re going. **So, wake up, stop being basic, and start coding like you actually know what you’re doing.**

Exactly! You see, making your code go from "meh" to "hell yeah" isn’t rocket science—it’s literally just three extra lines. Here’s what you should be doing:  
  
def train\_tensor(data\_loader, model, criterion, optimizer, scheduler=None, early\_stop=None):

learning\_rate\_tracker = {}

epoch\_correct = 0

running\_loss = 0.0

model.train()

scaler = torch.cuda.amp.GradScaler() # Initialize the gradient scaler

for i, (images, labels) in tqdm(enumerate(data\_loader), total=len(data\_loader), desc="Training"):

learning\_rate\_tracker[i] = optimizer.param\_groups[0]['lr']

images = images.to(device)

labels = labels.to(device)

optimizer.zero\_grad()

with torch.cuda.amp.autocast(): # Enable mixed precision

outputs = model(images)

loss = criterion(outputs, labels)

running\_loss += loss.item()

predicted = torch.max(outputs.data, 1)[1]

epoch\_correct += (predicted == labels).sum().item()

if early\_stop and i == early\_stop:

break

scaler.scale(loss).backward() # Scale the loss and call backward()

scaler.step(optimizer) # Step the optimizer

scaler.update() # Update the scaler

if scheduler:

scheduler.step()

return epoch\_correct, running\_loss, learning\_rate\_tracker

See? Writing good code isn’t some mythical beast you have to slay. It’s just about spending a little extra time, doing a bit of research, and adding a few lines of code. You’re living in 2024—abstractions and libraries have done all the heavy lifting for you. So, instead of waiting around for her text that’s never coming, put in the effort to level up your code. Stop being a lazy twat and start writing code that actually makes your GPU earn its keep. Bob’s your uncle, and now you’re ready to roll.

You might be thinking, ‘Alright, those are some big words, but where are the receipts?’ Well, here’s the deal. I’ve got the numbers to back it up. Don’t believe me? Go ahead and run the code yourself—it’s up on GitHub for you to try.

I’m running an RTX 4060, and when I was training my ResNet architecture on CIFAR10 using just CUDA, I was clocking in at around 28 iterations per second on average. The accuracy during training? Plateaued at about 97%. But once I flipped the switch to mixed precision, it was like hitting the turbo button—45 iterations per second, and my training accuracy jumped to 99%.

So, there you have it. You can’t just write ‘good’ code and call it a day. If you’re not optimizing with the right tools, you’re leaving performance on the table. Get some help, do it better, and start cranking out code that actually makes your GPU sweat.