the new graidents are added to existing gradients stored in .grad . it does not replace the old gradient automatically .
optimizer.zero_grad() clears the old gradients before computing new ones . if we forget it then it keeps up adding to the previous one only and becomes mathematically incorrect .

.view() works on contiguous tensor while .reshape() works on both contiguous and non contiguous tensors.
.reshape () is safer to use as fallback.

pytorch will give an error since both the tensors must be on same device .
tensors do not automatically inherit models device . they must explicitly be moved to same device .

```python
import torch
images = torch.rand(4, 28, 28)
def image_mask(images):
    masked_images = torch.where(images < 0.5,
                                torch.tensor(0.0),
                                torch.tensor(1.0))
    return masked_images
output = image_mask(images)
print(output)
```

```python
import torch
x = torch.tensor(4.0, requires_grad=True)
y = x**3 + 2*x
y.backward()
print("PyTorch gradient:", x.grad.item())
manual_grad = 3*(4**2) + 2
print("Manual gradient:", manual_grad)
if x.grad.item() != manual_grad:
    raise ValueError("Gradient mismatch!")
else:
    print("Gradient check passed.")
```

```python
import torch
from torch.utils.data import Dataset
class NumberDataset(Dataset):
    def __init__(self, numbers):
        self.numbers = numbers

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

    def __getitem__(self, index):
        x = torch.tensor(self.numbers[index], dtype=torch.float32)
        y = torch.tensor(self.numbers[index] * 2, dtype=torch.float32)
        return x, y
numbers = [1, 2, 3, 4]
dataset = NumberDataset(numbers)
print("Dataset length:", len(dataset))
print("First item:", dataset[0])
```

```python
import torch
import torch.nn as nn
class SimpleClassifier(nn.Module):
    def __init__(self):
        super(SimpleClassifier, self).__init__()
        self.fc1 = nn.Linear(10, 5)
        self.fc2 = nn.Linear(5, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x
model = SimpleClassifier()
x = torch.randn(1, 10)
output = model(x)
print(output)
```

```python
import torch
import torch.nn as nn
import torch.optim as optim
class SimpleClassifier(nn.Module):
    def __init__(self):
        super(SimpleClassifier, self).__init__()
        self.fc1 = nn.Linear(10, 5)
        self.fc2 = nn.Linear(5, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x
def train_step(model, inputs, targets, optimizer, criterion):
    predictions = model(inputs)
    loss = criterion(predictions, targets)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    return loss.item()
model = SimpleClassifier()
inputs = torch.randn(4, 10)
targets = torch.rand(4, 1)
optimizer = optim.SGD(model.parameters(), lr=0.01)
criterion = nn.BCELoss()
loss_value = train_step(model, inputs, targets, optimizer, criterion)
print("Loss:", loss_value)
```