**First issue:** Requirements were not met. Had to create a requirements file for this to be fixed.

**Understanding and Fixing the "tensor does not require grad" Error**

**The Problem**

The error you encountered is a common issue in PyTorch:

Copy

RuntimeError: element 0 of tensors does not require grad and does not have a grad\_fn

This error occurs when PyTorch tries to backpropagate through a computational graph, but somewhere in the graph there's a tensor that doesn't have gradient tracking enabled.

**Root Causes in Your Case**

After analyzing your code, I identified several issues in the loss function implementation:

1. **Edge case handling**: The original triplet loss implementation had problems when there were no valid positive or negative pairs for some samples in a batch.
2. **Gradient disconnection**: When handling special cases (like no valid triplets), the code was returning tensors that weren't properly connected to the computation graph.
3. **Masking operations**: The way the masks were applied with operations like min() and max() sometimes led to gradient disconnection.

**Key Fixes**

In the updated loss\_functions.py file, I made these changes:

1. **Improved distance calculation**: Changed the pairwise distance computation to maintain gradient flow throughout.
2. **Proper masking**: Replaced the problematic masking approach with masked\_fill() to handle invalid values.
3. **Gradient preservation for edge cases**: When there are no valid triplets, instead of returning a new zero tensor (which would disconnect the gradient), I now return:

python

Copy

return torch.sum(triplet\_loss) \* 0

This maintains connection to the computation graph while still returning zero.

1. **Handling empty valid sets**: Added checks for when there are no valid samples and proper fallbacks.
2. **Valid sample tracking**: Added logic to track which samples have valid positives and negatives.

**For Deep Metric Learning Configuration**

Understanding these parameters from the README:

* **BACKBONE**: The CNN architecture used as the feature extractor. Options are:
  + resnet18: Smaller, faster model (18 layers)
  + resnet34: Medium-sized model (34 layers)
  + resnet50: Larger model with more capacity (50 layers)
* **EMBEDDING\_SIZE**: Determines the dimensionality of your embedding space. Higher values (e.g., 256, 512) provide more capacity but require more computation and might overfit on smaller datasets. Lower values (e.g., 64, 128) are more efficient but may have limited representational power.
* **LOSS\_TYPE**: The loss function for training:
  + triplet: Pushes anchor closer to positive and further from negative examples (requires margin)
  + contrastive: Works on pairs, bringing similar items together and pushing dissimilar items apart (requires margin)
  + arcface: Classification-based approach that adds angular margin to improve discrimination
* **MARGIN**: For triplet and contrastive losses, this is the minimum distance enforced between positive and negative pairs. Typically values between 0.2-1.0 work well. Higher margin means more stringent separation.
* **MINING\_TYPE**: Strategies for selecting triplets in triplet loss:
  + batch\_all: Uses all valid triplets in a batch (can be inefficient)
  + batch\_hard: Only uses the hardest positive and hardest negative for each anchor (efficient, helps with faster convergence)
  + batch\_semi\_hard: Uses negatives that are harder than positives but still within margin (balanced approach)

The code is designed to be modular, so you can easily swap different components:

* Try different backbones
* Change loss functions
* Adjust hyperparameters like embedding size and margin
* Implement custom mining strategies

New error:

2025-04-08 19:24:36.555531: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable TF\_ENABLE\_ONEDNN\_OPTS=0.

These messages aren't actually errors, but rather informational messages from TensorFlow about oneDNN optimizations. Since your project primarily uses PyTorch, these messages are likely coming from a dependency that uses TensorFlow under the hood (such as UMAP for visualization).

To fix it, I used a silence\_tensorflow.py

"""

Add this at the very beginning of main.py to silence TensorFlow warnings and messages.

"""

import os

import logging

import warnings

# Silence TensorFlow messages

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '3' # 0=DEBUG, 1=INFO, 2=WARNING, 3=ERROR

os.environ['TF\_ENABLE\_ONEDNN\_OPTS'] = '0' # Disable oneDNN custom operations

# Silence other logging

warnings.filterwarnings('ignore')

logging.getLogger('tensorflow').setLevel(logging.ERROR)

Error 2:

Epochs coming out as 0 train and val losses....?  
  
 warnings.warn(msg)

2025-04-08 19:31:35,590 - INFO - Starting training for 30 epochs

2025-04-08 19:33:01,474 - INFO - Epoch 1/30 - Train Loss: 0.0054, Val Loss: 0.0000

2025-04-08 19:33:01,587 - INFO - Checkpoint saved to outputs\models\checkpoint\_epoch\_1.pth

2025-04-08 19:33:01,696 - INFO - New best model saved with validation loss: 0.0000

2025-04-08 19:34:29,735 - INFO - Epoch 2/30 - Train Loss: 0.0000, Val Loss: 0.0000

2025-04-08 19:34:29,868 - INFO - Checkpoint saved to outputs\models\checkpoint\_epoch\_2.pth

2025-04-08 19:35:56,291 - INFO - Epoch 3/30 - Train Loss: 0.0000, Val Loss: 0.0000

2025-04-08 19:35:56,406 - INFO - Checkpoint saved to outputs\models\checkpoint\_epoch\_3.pth

Training: 0%| | 0/79 [00:00<?, ?it/s]

I've identified and fixed several issues that were causing your training to report 0.0000 losses:

**Issues Fixed:**

1. **Model initialization**: Updated model.py to use the correct weights parameter instead of the deprecated 'pretrained' parameter.
2. **Triplet loss implementation**: Completely rewrote the triplet loss implementation to:
   * Handle edge cases properly when no valid triplets are found
   * Ensure the loss stays connected to the computational graph
   * Use more stable distance calculations
   * Fix bugs in the mining strategies
3. **Data handling**: Enhanced the data module to:
   * Ensure labels are tensor-typed
   * Add error handling for corrupted images
   * Improve few-shot dataset creation

Issue:

You're right, the current implementation overrides your config file values with the parser defaults, which isn't what you want. Here's an updated version of main.py that respects the config file values first:

main.py (Fixed)

Code

""" Main script for Deep Metric Learning on the Oxford-IIIT Pet Dataset """ # Add these lines at the very beginning import os # Silence TensorFlow messages os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '3' # 0=DEBUG, 1=INFO, 2=WARNING, 3=ERROR os.environ['TF

The problem is that the current implementation always overrides the config file values with parser defaults, even when those arguments aren't explicitly provided on the command line.

I've fixed this by:

1. **Setting all argparse defaults to None** - This way, we can detect which arguments were explicitly passed by the user vs. which weren't specified.
2. **Only updating config values when an argument is not None** - This ensures that values from the config file are preserved unless explicitly overridden via command line.
3. **Special handling for 'store\_true' flags** - Since the --pretrained flag is a special case (action='store\_true'), I've added separate handling for it.
4. **Fallback for mode selection** - Added a fallback to use the config's MODE value if present.

With this updated implementation, your config file values will be properly respected as the primary source of configuration, with command-line arguments only overriding specific values when explicitly provided.

This gives you the best of both worlds:

* You can maintain your preferred settings in the config.py file
* You can quickly override specific parameters via command line when needed for experimentation

For example, if you want to test a different embedding size without changing your config.py, you can simply run:

python main.py --embedding\_size 256

And only that parameter will be overridden, while all other parameters will be taken from your config file.

Issue:   
  
You're right - there's an issue with how the pretrained flag is being handled. Since this is an action='store\_true' argument, it works differently from other arguments.

Here's a fix for the main.py file to correctly handle the pretrained flag:

main.py (Pretrained Flag Fix)

Code

""" Main script for Deep Metric Learning on the Oxford-IIIT Pet Dataset """ # Add these lines at the very beginning import os # Silence TensorFlow messages os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '3' # 0=DEBUG, 1=INFO, 2=WARNING, 3=ERROR os.environ['TF

I see the issue now. The problem is with how boolean flags like --pretrained are handled in argparse.

**The Problem**

For boolean flags, argparse uses action='store\_true', which:

1. Sets the default to False
2. When the flag is provided, sets the value to True

However, in our current implementation, we're using None as a detector for "was this argument explicitly provided?", but this doesn't work for boolean flags because they're always either True or False.

**The Solution**

I've fixed this in the updated code by:

1. Using a more sophisticated approach for the pretrained flag:

python

*# Changed to store\_true/store\_false with no default for pretrained*

parser.add\_argument('--pretrained', action='store\_true', dest='pretrained', help='Use pretrained backbone')

parser.add\_argument('--no-pretrained', action='store\_false', dest='pretrained', help='Do not use pretrained backbone')

parser.set\_defaults(pretrained=None) *# Set default to None to detect if it was specified*

This creates two flags:

* + --pretrained to set it to True
  + --no-pretrained to set it to False
  + Default is None, which allows us to detect if neither was provided

1. Now we can properly detect if the pretrained flag was explicitly set or if we should use the config value.

With this change, the PRETRAINED = True value from your config file will be respected unless you explicitly override it with either --pretrained or --no-pretrained on the command line.

Replace your current main.py with this updated version and your pretrained weights should be properly used!

Training run:

**Training Run Summary - CV Metric Learning on Oxford-IIIT Pet Dataset**

**Configuration Details**

* **Model**: ResNet18 backbone (pretrained)
* **Embedding Size**: 128
* **Batch Size**: 64
* **Learning Rate**: 0.0001
* **Loss Type**: triplet with margin 0.3
* **Mining Type**: batch\_hard
* **Dataset Split**: Train: 5,032, Val: 1,259, Test: 699

**Training Progress**

| **Epoch** | **Train Loss** | **Val Loss** | **Time** |
| --- | --- | --- | --- |
| 1/3 | 0.3229 | 0.2897\* | ~5.9 min |
| 2/3 | 0.2713 | 0.3018 | ~6.0 min |
| 3/3 | 0.2720 | 0.3257 | ~5.9 min |

\*Best model saved at epoch 1 (lowest validation loss)

**Total Training Time**: 17.85 minutes

**Evaluation Results**

**Verification Task**

* **ROC AUC**: 0.9795
* **Equal Error Rate (EER)**: 0.0729
* **EER Threshold**: 0.3607

**Retrieval Performance**

* **Precision@1**: 0.8398 | **Recall@1**: 0.8398
* **Precision@5**: 0.8369 | **Recall@5**: 0.9456
* **Precision@10**: 0.8215 | **Recall@10**: 0.9700

**Few-Shot Learning**

* Note: Only 2 classes found instead of 5 for few-shot learning
* **2-way 5-shot Accuracy**: 1.0000 ± 0.0000 (100 tasks)

**Key Observations**

* Model achieved best validation performance in epoch 1
* Strong verification performance with 0.98 ROC AUC
* Excellent retrieval metrics, especially recall@5 (0.95) and recall@10 (0.97)
* Perfect accuracy on limited few-shot tasks