yolo_preprocess

October 18, 2024

#Purpose

This notebook demonstrates the differences between the source implementation and a branch implementation of a preprocessing function, explains why the branch implementation throws an error, and provides a recommended fix. We will walk through how each implementation handles object detection targets, highlighting key differences in how data is structured and processed.

[5]:
```
# Input data mimicking the format used by the source implementation

# 'input_targets_source_implementation' is a tensor where each row corresponds␣
↪to
# a detected object in an image. The first column represents the image index,
# the second column represents the object label, and the remaining four columns
# represent the bounding box coordinates in [x_center, y_center, width, height]␣
↪format.

# Here, we are creating data for 4 unique images. Three of the images have one␣
↪object each.
# One image (Image 2) contains two objects, but there are only three unique␣
↪labels across
# all the images. The batch size will be greater than the number of unique␣
↪labels.

import torch

# Creating the input data for the source implementation
# Format: [image_index, label, x_center, y_center, width, height]
targets_source_implementation = torch.tensor([
    [0, 1, 0.1, 0.2, 0.3, 0.4],  # Image 0, Object 1 (Label 1)
    [1, 2, 0.5, 0.6, 0.7, 0.8],  # Image 1, Object 1 (Label 2)
    [2, 1, 0.3, 0.3, 0.4, 0.4],  # Image 2, Object 1 (Label 1)
    [2, 3, 0.7, 0.7, 0.2, 0.2],  # Image 2, Object 2 (Label 3)
    [3, 2, 0.4, 0.5, 0.6, 0.7],  # Image 3, Object 1 (Label 2)
])

# Explanation:
# - Image 0 contains 1 object (Label 1) with bounding box [0.1, 0.2, 0.3, 0.4].
# - Image 1 contains 1 object (Label 2) with bounding box [0.5, 0.6, 0.7, 0.8].
# - Image 2 contains 2 objects:
#     - Object 1: Label 1 with bounding box [0.3, 0.3, 0.4, 0.4].
#     - Object 2: Label 3 with bounding box [0.7, 0.7, 0.2, 0.2].
# - Image 3 contains 1 object (Label 2) with bounding box [0.4, 0.5, 0.6, 0.7].
# - There are 4 images but only 3 unique labels: {1, 2, 3}.
```

[3]:
```
# Implementation of the source code preprocess function
def source_preprocess(targets, batch_size, scale_tensor):
    if targets.shape[0] == 0:
        out = torch.zeros(batch_size, 0, 5, device='cpu')
    else:
        i = targets[:, 0]  # image index
        _, counts = i.unique(return_counts=True)
        out = torch.zeros(batch_size, counts.max(), 5, device='cpu')
        for j in range(batch_size):
            matches = i == j
            n = matches.sum()
            if n:
                out[j, :n] = targets[matches, 1:]
        out[..., 1:5] = xywh2xyxy(out[..., 1:5])
    return out

# Dummy bounding box conversion function
def xywh2xyxy(boxes):
    # This is a placeholder function to simulate the bounding box conversion.
    return boxes  # In reality, it does more.
```

[6]:
```
# Testing the source_preprocess function using the␣
 ↪input_targets_source_implementation

# Set up parameters for the test
batch_size = 4  # We have 4 images, so the batch size is set to 4 (greater than␣
 ↪the number of unique labels)
scale_tensor = torch.tensor([640, 480, 640, 480])

# Run the preprocess function
output = source_preprocess(targets_source_implementation, batch_size,␣
 ↪scale_tensor)

# Print the output tensor
print("Output of source_preprocess function:\n", output)

# Explanation:
# The output tensor has the shape [batch_size, max_objects_per_image, 5].
# In this case:
# - The first dimension corresponds to the 4 unique images in the batch.
# - The second dimension represents the maximum number of objects in any image␣
↪(in this case, 2 objects for Image 2).
# - The third dimension holds 5 values for each object:
#     - The first value is the object label.
#     - The next four values are the bounding box coordinates in [x_min, y_min,␣
↪x_max, y_max] format (after conversion).

# Detailed breakdown of the output:
for img_idx in range(batch_size):
    print(f"\nImage {img_idx} contains the following objects:")
    for obj_idx in range(output.shape[1]):
        obj_data = output[img_idx, obj_idx]
        if obj_data[0] != 0:  # Skip empty entries (padded with zeros)
            print(f"  Object {obj_idx + 1}: Label = {obj_data[0].item()}, "
                  f"Bounding Box = {obj_data[1:].tolist()}")
        else:
            print(f"  Object {obj_idx + 1}: No object (empty row)")
```

```
Output of source_preprocess function:
 tensor([[[1.0000, 0.1000, 0.2000, 0.3000, 0.4000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

        [[2.0000, 0.5000, 0.6000, 0.7000, 0.8000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

        [[1.0000, 0.3000, 0.3000, 0.4000, 0.4000],
         [3.0000, 0.7000, 0.7000, 0.2000, 0.2000]],

        [[2.0000, 0.4000, 0.5000, 0.6000, 0.7000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]])

Image 0 contains the following objects:
  Object 1: Label = 1.0, Bounding Box = [0.10000000149011612,
0.20000000298023224, 0.30000001192092896, 0.4000000059604645]
  Object 2: No object (empty row)

Image 1 contains the following objects:
  Object 1: Label = 2.0, Bounding Box = [0.5, 0.6000000238418579,
0.699999988079071, 0.800000011920929]
  Object 2: No object (empty row)

Image 2 contains the following objects:
  Object 1: Label = 1.0, Bounding Box = [0.30000001192092896,
0.30000001192092896, 0.4000000059604645, 0.4000000059604645]
  Object 2: Label = 3.0, Bounding Box = [0.699999988079071, 0.699999988079071,
0.20000000298023224, 0.20000000298023224]
```

```
Image 3 contains the following objects:
  Object 1: Label = 2.0, Bounding Box = [0.4000000059604645, 0.5,
0.6000000238418579, 0.699999988079071]
  Object 2: No object (empty row)
```

1 Creating the Branch Implementation Targets

In this section, we create the `targets_branch_implementation`, which represents the target data as a list of dictionaries. Each dictionary corresponds to an object in an image, containing the image index, object label, and bounding box. This structure is different from the source implementation, but it stores the same information in a more flexible format.

[20]:
```
# Creating 'targets_branch_implementation' as a list of dictionaries

# This format is different from the source implementation because it stores the␣
↪data as a list of dictionaries.
# Each dictionary corresponds to a detected object in an image and contains␣
↪fields for:
# - "image": Identifies the image index (similar to the first column in the␣
↪source format).
# - "labels": The label of the object detected in the image (scalar).
# - "boxes": The bounding box coordinates for the object in the image.

# I made the assumption that this structure contains an "image" field because␣
↪the model needs to associate
# each label and bounding box with a specific image, especially when multiple␣
↪images and multiple objects
# per image are involved. Without this, it would be difficult to match labels␣
↪and boxes to their corresponding images.

targets_branch_implementation = [
    {"image": torch.tensor([0]), "labels": torch.tensor([1]), "boxes": torch.
↪tensor([[0.1, 0.2, 0.3, 0.4]])},  # Image 0, 1 object
    {"image": torch.tensor([1]), "labels": torch.tensor([2]), "boxes": torch.
↪tensor([[0.5, 0.6, 0.7, 0.8]])},  # Image 1, 1 object
    {"image": torch.tensor([2]), "labels": torch.tensor([1]), "boxes": torch.
↪tensor([[0.3, 0.3, 0.4, 0.4]])},  # Image 2, Object 1
    {"image": torch.tensor([2]), "labels": torch.tensor([3]), "boxes": torch.
↪tensor([[0.7, 0.7, 0.2, 0.2]])},  # Image 2, Object 2
    {"image": torch.tensor([3]), "labels": torch.tensor([2]), "boxes": torch.
↪tensor([[0.4, 0.5, 0.6, 0.7]])},  # Image 3, 1 object
]

# Explanation:
# - Each entry now corresponds to a single object, even if multiple objects are␣
↪in the same image.
```

```
# - The structure is a list of dictionaries where:
#     - "image": A scalar tensor identifying the image index.
#     - "labels": A scalar tensor representing the label for each object.
#     - "boxes": A tensor of bounding box coordinates for the object.
# - For Image 2, we have split the objects into two separate rows: one for␣
  ↪Label 1 and one for Label 3.
# - This adjustment ensures that 'labels' is always a scalar, aligning with the␣
  ↪expected data structure.
```

[27]:
```
# Original branch preprocess implementation (this will throw an error)
def branch_preprocess(targets, batch_size, scale_tensor):
    """Preprocesses targets before computing loss."""
    if len(targets) == 0:
        out = torch.zeros(batch_size, 0, 5, device='cpu')
    else:
        labels = []
        for label in targets:
            labels.append(label["labels"])

        _, counts = torch.tensor(labels).unique(return_counts=True)

        # The tensor is size 5 because of (class, x, y, x, y)
        out = torch.zeros(batch_size, counts.max(), 5, device='cpu')

        for batch in range(batch_size):
                out[batch, : counts[batch], 0] = targets[batch]["labels"]
                out[batch, : counts[batch], 1:5] = targets[batch]["boxes"]
        out[..., 1:5] = xywh2xyxy(out[..., 1:5])

    return out


# Test with branch implementation (this will throw an error)
try:
    batch_size = 4  # Batch size is set to 4 for this test
    scale_tensor = torch.tensor([640, 480, 640, 480])  # Dummy scaling factors
    output_branch = branch_preprocess(targets_branch_implementation,␣
  ↪batch_size, scale_tensor)

    # Print the output if no error occurs (though an error is expected)
    print("Output of branch_preprocess function:\n", output_branch)
except Exception as e:
    # Print the error message
    print(f"Error occurred: {e}")

# Explanation:
```

```
# This implementation throws an error because of the line `out[batch, :
↪counts[batch]]`.
# The issue arises when the batch size is greater than the number of unique␣
↪labels in the data.
# Specifically, the variable `counts` contains the count of unique labels, and␣
↪its length
# (i.e., the number of unique labels, `len(counts)`) is less than the batch␣
↪size.
# When the loop tries to access `counts[batch]` for batches that don't have a␣
↪corresponding
# unique label, it causes an out-of-bounds error.
# This happens because there aren't enough unique labels to match the batch␣
↪size, leading
# to an attempt to index into `counts` for a batch index that doesn't exist.
```

```
Error occurred: index 3 is out of bounds for dimension 0 with size 3
```

2 Fixing the Branch Implementation: Converting Targets to Match Source Format

In this section, we implement the `preprocess_fixed_branch` function to fix the issues in the branch implementation. The logic behind the fix is to convert the target data from a list of dictionaries into a single tensor, matching the format of the source implementation. By doing so, we ensure that the data can be processed using the same logic as the source code. Each object in an image is represented by a row in the tensor, containing the image index, object label, and bounding box, which allows the function to handle multiple objects per image and avoid errors related to mismatched batch size and label counts.

[35]:
```
# Modified branch preprocess function that processes 'targets' into the tensor␣
↪shape of the original implementation
def preprocess_fixed_branch(targets, batch_size, scale_tensor):
    """Preprocesses targets and converts them into the tensor format expected␣
↪by the original implementation."""

    # Step 1: Convert the list of dictionaries (targets) into a single tensor,␣
↪similar to the source implementation
    all_targets = []

    for target in targets:
        image_idx = target["image"]  # Extract the image index
        labels = target["labels"]  # Scalar label
        boxes = target["boxes"]  # Bounding box

        # Stack the image index, label, and bounding box into a tensor for each␣
↪object
        img_target = torch.cat([
            image_idx.unsqueeze(0).float(),  # Image index
            labels.unsqueeze(0).float(),                          # Labels␣
↪(convert to 2D for concatenation)
            boxes.float()                                         # Bounding box
        ], dim=1)

        all_targets.append(img_target)

    # Step 2: Concatenate all individual object tensors into one large tensor,␣
↪like the source implementation
    all_targets = torch.cat(all_targets, dim=0)  # Shape: (total_objects, 6)
    print("You can see now that the shape of targets after modification matches␣
↪the shape of targest in the source implementation")
    print(all_targets)

    # Step 3: Now we can process 'all_targets' just like in the original source␣
↪implementation
    if all_targets.shape[0] == 0:
        out = torch.zeros(batch_size, 0, 5, device='cpu')
    else:
        i = all_targets[:, 0]  # Image index
        _, counts = i.unique(return_counts=True)
        out = torch.zeros(batch_size, counts.max(), 5, device='cpu')

        for j in range(batch_size):
            matches = i == j  # Find the targets for the current image (j)
            n = matches.sum()  # Number of objects in this image
            if n:
                out[j, :n] = all_targets[matches, 1:]  # Set labels and␣
↪bounding boxes for this image

        out[..., 1:5] = xywh2xyxy(out[..., 1:5])

    return out

# Dummy bounding box conversion function (unchanged)
def xywh2xyxy(boxes):
    # This is a placeholder function to simulate the bounding box conversion.
    return boxes  # In reality, you'd convert xywh format to xyxy format here.

# Running the test with the fixed branch implementation
try:
    batch_size = 4  # Batch size of 4 images
    scale_tensor = torch.tensor([640, 480, 640, 480])  # Dummy scaling factors
    output_fixed_branch =␣
↪preprocess_fixed_branch(targets_branch_implementation, batch_size,␣
↪scale_tensor)
```

```
    # Print the output
    print("Output of preprocess_fixed_branch function:\n", output_fixed_branch)
    print("You can now see that the output matches the output of the source␣
 ↪implementation on comparable inputs")
except Exception as e:
    # Print the error if something goes wrong
    print(f"Error occurred: {e}")
```

```
You can see now that the shape of targets after modification matches the shape
of targest in the source implementation
tensor([[0.0000, 1.0000, 0.1000, 0.2000, 0.3000, 0.4000],
        [1.0000, 2.0000, 0.5000, 0.6000, 0.7000, 0.8000],
        [2.0000, 1.0000, 0.3000, 0.3000, 0.4000, 0.4000],
        [2.0000, 3.0000, 0.7000, 0.7000, 0.2000, 0.2000],
        [3.0000, 2.0000, 0.4000, 0.5000, 0.6000, 0.7000]])
Output of preprocess_fixed_branch function:
 tensor([[[1.0000, 0.1000, 0.2000, 0.3000, 0.4000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

        [[2.0000, 0.5000, 0.6000, 0.7000, 0.8000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

        [[1.0000, 0.3000, 0.3000, 0.4000, 0.4000],
         [3.0000, 0.7000, 0.7000, 0.2000, 0.2000]],

        [[2.0000, 0.4000, 0.5000, 0.6000, 0.7000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]])
You can now see that the output matches the output of the source implementation
on comparable inputs
```

[33]:
```
import sys
import os
from contextlib import redirect_stdout

with open(os.devnull, 'w') as devnull:
    with redirect_stdout(devnull):
        output_fixed_branch =␣
 ↪preprocess_fixed_branch(targets_branch_implementation, batch_size,␣
 ↪scale_tensor)

print(output_fixed_branch)
```

```
tensor([[[1.0000, 0.1000, 0.2000, 0.3000, 0.4000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

        [[2.0000, 0.5000, 0.6000, 0.7000, 0.8000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

        [[1.0000, 0.3000, 0.3000, 0.4000, 0.4000],
         [3.0000, 0.7000, 0.7000, 0.2000, 0.2000]],

        [[2.0000, 0.4000, 0.5000, 0.6000, 0.7000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]])
```

[34]:
```
output_source_implementation = source_preprocess(targets_source_implementation,␣
  ↪batch_size, scale_tensor)
print(output_source_implementation)
```

```
tensor([[[1.0000, 0.1000, 0.2000, 0.3000, 0.4000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

         [[2.0000, 0.5000, 0.6000, 0.7000, 0.8000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]],

         [[1.0000, 0.3000, 0.3000, 0.4000, 0.4000],
          [3.0000, 0.7000, 0.7000, 0.2000, 0.2000]],

         [[2.0000, 0.4000, 0.5000, 0.6000, 0.7000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]]])
```