AI VIETNAM
All-in-One Course
(TA Session)

AI VIET NAM
@aivietnam.edu.vn

Object Detection Part I

Exercise

Minh-Duc Bui
Phuc-Thinh Nguyen

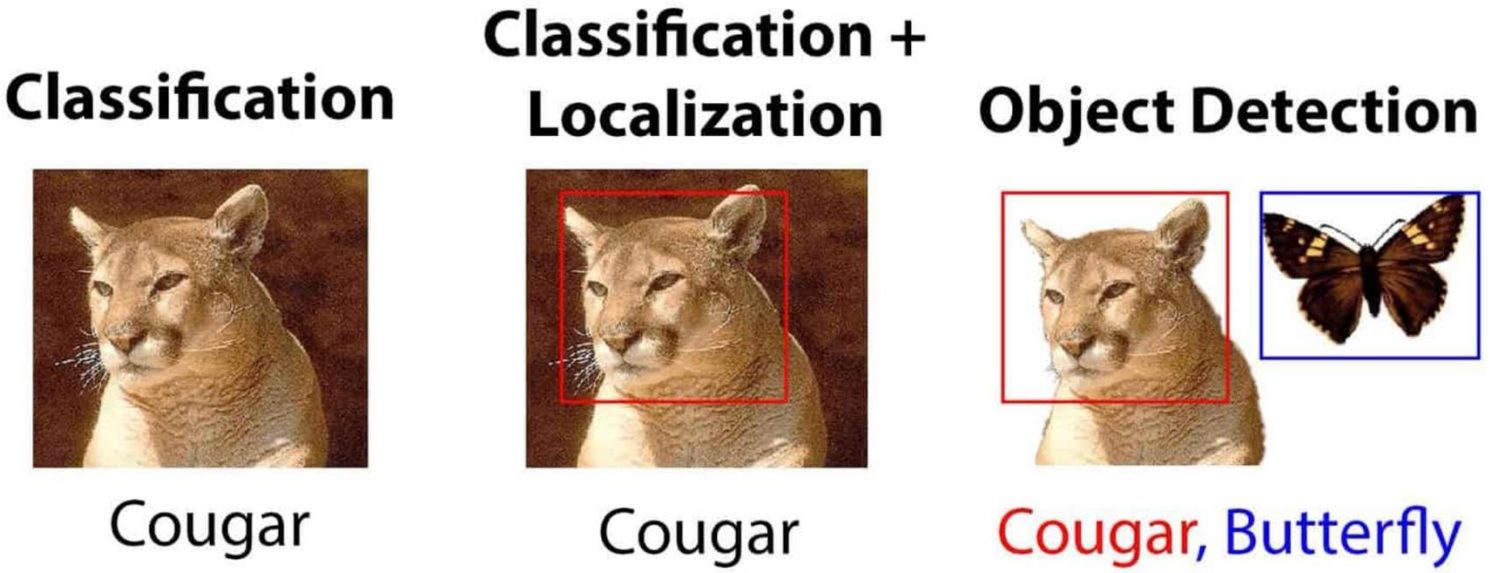
Year 2025

Agenda

- Introduction
- Classification
- Classification + Localization
- Basic YOLOv1
- Question

Introduction

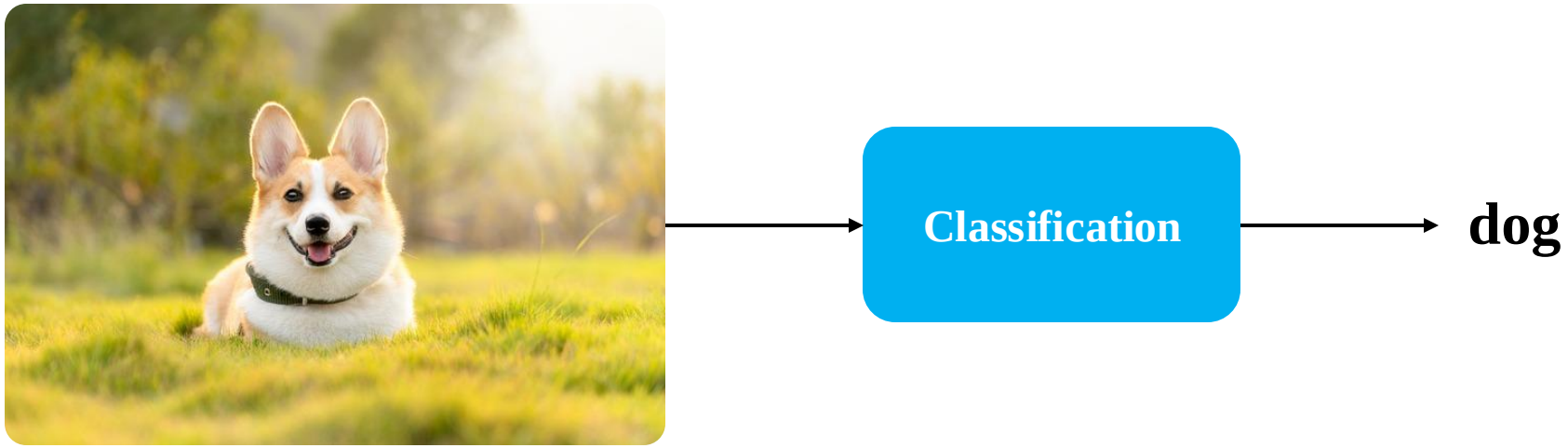
❖ Classification, Localization, and Object Detection



Classification

Classification

❖ Introduction



Classification

❖ Dataset

LARXEL · UPDATED 5 YEARS AGO

94 | New Notebook | Download

Dog and Cat Detection

Where is the good boy? (3686 images of cat and dogs with bounding boxes)

Data Card | Code (21) | Discussion (0) | Suggestions (0)

About Dataset

About this Data

In reality, the rise of AI is due to us looking for a way to hone our capacity to show love to our pets.

The computer itself evolved so that it could house more pictures of cattos and doggos and big data is largely a result of us posting more and more pictures of our pets in the interwebs.

With this dataset, containing 3686 images with bounding boxes for 2 classes (doggos, cattos), you can finally apply your skills to a relevant and timeless problem of the human race: find the good boyes so that you can give them treats and pet them.

Usability
8.75

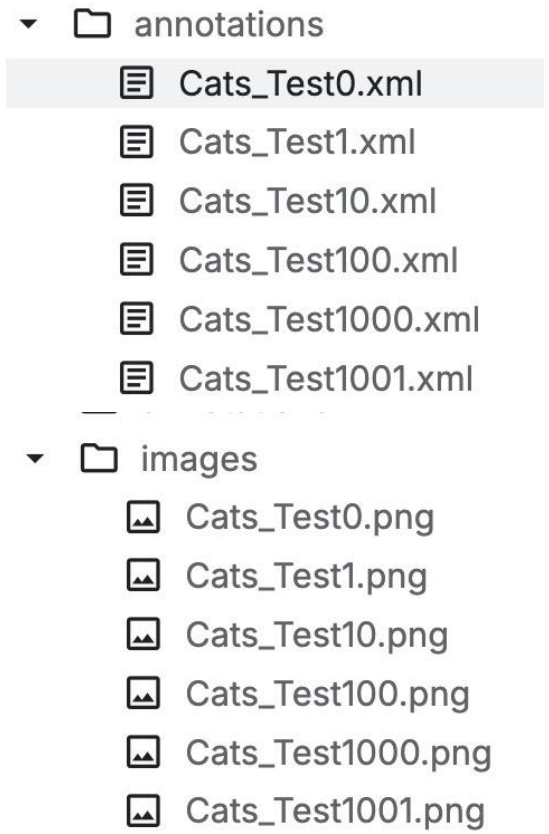
License
CC BY-SA 4.0

Expected update frequency
Never

Tags

Arts and Entertainment | Image | Computer Vision

❖ Dataset



Cats_Test0.xml (559 B)

About this file

Add Suggestion

This file does not have a description yet.

```
<annotation>
    <folder>images</folder>
    <filename>Cats_Test0.png</filename>
    <size>
        <width>233</width>
        <height>350</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>cat</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <occluded>0</occluded>
        <difficult>0</difficult>
        <bndbox>
            <xmin>83</xmin>
            <ymin>29</ymin>
            <xmax>197</xmax>
            <ymax>142</ymax>
        </bndbox>
    </object>
</annotation>
```

Data Explorer

Version 1 (1.1 GB)

- annotations
  - Cats_Test0.xml
  - Cats_Test1.xml
  - Cats_Test10.xml
  - Cats_Test100.xml
  - Cats_Test1000.xml
  - Cats_Test1001.xml
  - Cats_Test1002.xml
  - Cats_Test1003.xml
  - Cats_Test1004.xml
  - Cats_Test1005.xml
  - Cats_Test1006.xml
  - Cats_Test1007.xml
  - Cats_Test1008.xml
  - Cats_Test1009.xml
  - Cats_Test101.xml
  - Cats_Test1010.xml
  - Cats_Test1011.xml
  - Cats_Test1012.xml
  - Cats_Test1013.xml
  - Cats_Test1014.xml
  - Cats_Test1015.xml
  - Cats_Test1016.xml
  - Cats_Test1017.xml
  - Cats_Test1018.xml
  - Cats_Test1019.xml
  - Cats_Test102.xml
  - Cats_Test1020.xml

❖ Dataset

```python
# Data directory
annotations_dir = os.path.join(data_dir, 'annotations')
image_dir = os.path.join(data_dir, 'images')

# Get list of image files and create a dummy dataframe to split the data
image_files = [f for f in os.listdir(image_dir) if os.path.isfile(os.path.join(image_dir, f))]
df = pd.DataFrame({'image_name': image_files})

df.head()

```

✓ 0.0s

	image_name
0	Cats_Test3363.png
1	Cats_Test478.png
2	Cats_Test1574.png
3	Cats_Test1212.png
4	Cats_Test3405.png

❖ Dataset

```python
# Split data
train_df, val_df = train_test_split(df, test_size=0.2, random_state=42)

# Transforms
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Datasets
train_dataset = ImageDataset(annotations_dir, image_dir, transform=transform)
val_dataset = ImageDataset(annotations_dir, image_dir, transform=transform)

# Filter datasets based on train_df and val_df
train_dataset.image_files = [f for f in train_dataset.image_files if f in train_df['image_name'].values]
val_dataset.image_files = [f for f in val_dataset.image_files if f in val_df['image_name'].values]

# Dataloaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

```

✓ 1.1s

❖ Dataset

```python
# Dataset Class
class ImageDataset(Dataset):
    def __init__(self, annotations_dir, image_dir, transform=None):
        self.annotations_dir = annotations_dir
        self.image_dir = image_dir
        self.transform = transform
        self.image_files = self.filter_images_with_multiple_objects()

    def filter_images_with_multiple_objects(self):
        valid_image_files = []
        for f in os.listdir(self.image_dir):
            if os.path.isfile(os.path.join(self.image_dir, f)):
                img_name = f
                annotation_name = os.path.splitext(img_name)[0] + ".xml"
                annotation_path = os.path.join(self.annotations_dir, annotation_name)

                if self.count_objects_in_annotation(annotation_path) <= 1:
                    valid_image_files.append(img_name)
                else:
                    print(
                        f"Image {img_name} has multiple objects and will be excluded from the dataset"
                    )
        return valid_image_files

    def count_objects_in_annotation(self, annotation_path):
        try:
            tree = ET.parse(annotation_path)
            root = tree.getroot()
            count = 0
            for obj in root.findall("object"):
                count += 1
            return count
        except FileNotFoundError:
            return 0

    def __len__(self):
        return len(self.image_files)
```

❖ Dataset

```python
def __getitem__(self, idx):
    # Image path
    img_name = self.image_files[idx]
    img_path = os.path.join(self.image_dir, img_name)

    # Load image
    image = Image.open(img_path).convert("RGB")

    # Annotation path
    annotation_name = os.path.splitext(img_name)[0] + ".xml"
    annotation_path = os.path.join(self.annotations_dir, annotation_name)

    # Parse annotation
    label = self.parse_annotation(annotation_path)

    if self.transform:
        image = self.transform(image)

    return image, label

def parse_annotation(self, annotation_path):
    tree = ET.parse(annotation_path)
    root = tree.getroot()

    label = None
    for obj in root.findall("object"):
        name = obj.find("name").text
        if (
            label is None
        ):  # Take the first label for now. We are working with 1 label per image
            label = name

    # Convert label to numerical representation (0 for cat, 1 for dog)
    label_num = 0 if label == "cat" else 1 if label == "dog" else -1

    return label_num
```

❖ Model

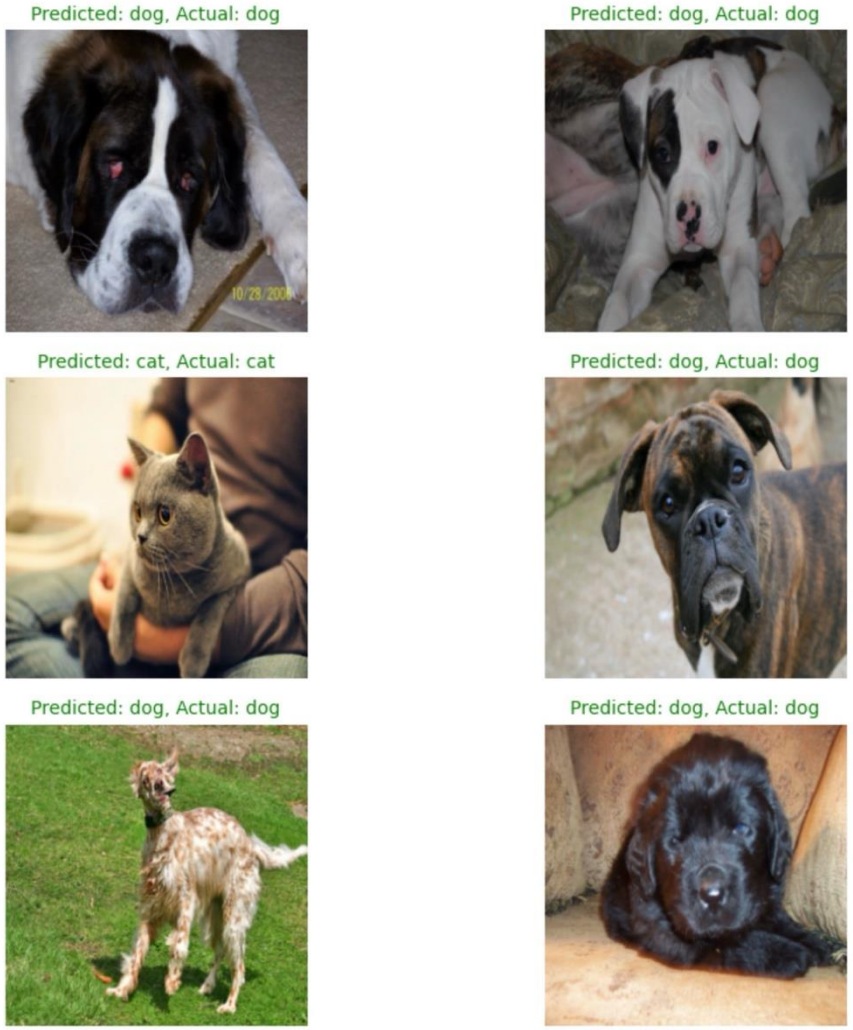
```python
# Model
model = models.resnet18(weights=ResNet18_Weights.DEFAULT)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 2)  # 2 classes: cat and dog

# Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

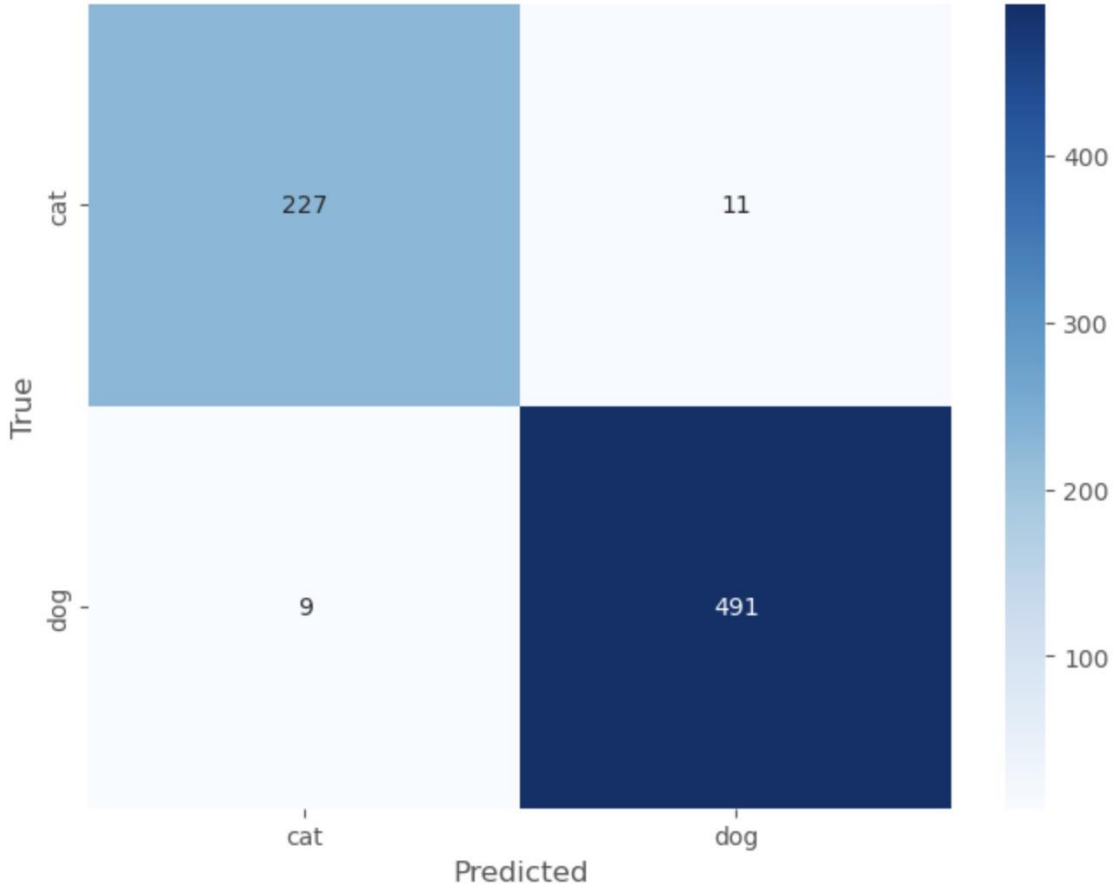
❖ Training

```python
# Training Loop
num_epochs = 10
for epoch in tqdm.tqdm(range(num_epochs), desc="Epochs"):
    model.train()
    for batch_idx, (data, targets) in enumerate(tqdm.tqdm(train_loader, desc="Training", leave=False)):
        data = data.to(device)
        targets = targets.to(device)

        scores = model(data)
        loss = criterion(scores, targets)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for data, targets in tqdm.tqdm(val_loader, desc="Validation", leave=False):
            data = data.to(device)
            targets = targets.to(device)
            scores = model(data)
            _, predictions = scores.max(1)
            correct += (predictions == targets).sum()
            total += targets.size(0)

        print(f'Epoch {epoch+1}/{num_epochs}, Validation Accuracy: {float(correct)/float(total)*100:.2f}%')

```

```
Epoch 1/10, Validation Accuracy: 93.50%

Epoch 2/10, Validation Accuracy: 95.12%

Epoch 3/10, Validation Accuracy: 89.43%

Epoch 4/10, Validation Accuracy: 81.84%

Epoch 5/10, Validation Accuracy: 94.58%

Epoch 6/10, Validation Accuracy: 92.01%

Epoch 7/10, Validation Accuracy: 92.14%

Epoch 8/10, Validation Accuracy: 93.77%

Epoch 9/10, Validation Accuracy: 89.97%

Epoch 10/10, Validation Accuracy: 87.53%
```

❖ Inference

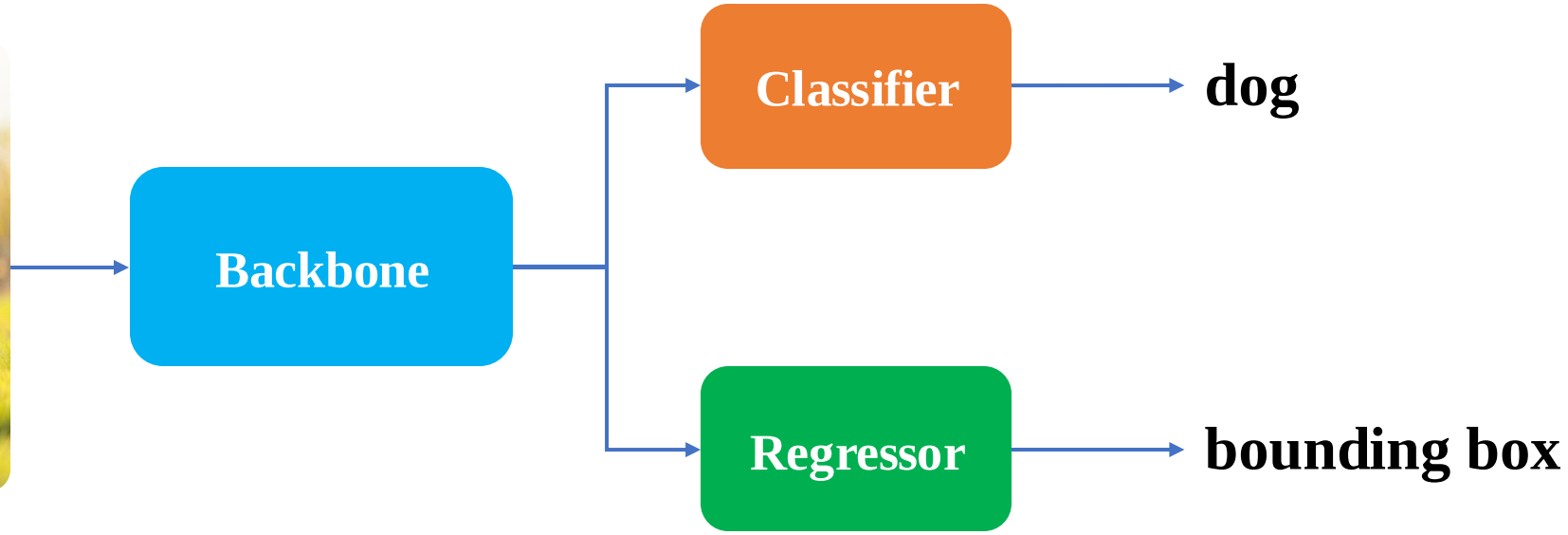

Predicted: dog, Actual: dog
Predicted: dog, Actual: dog
Predicted: cat, Actual: cat
Predicted: dog, Actual: dog
Predicted: dog, Actual: dog
Predicted: dog, Actual: dog


227
11
9
491
cat
dog
True
cat
dog
Predicted
400
300
200
100

Classification + Localization

Classification + Localization

❖ Introduction



Classification + Localization

❖ Dataset

```python
def __getitem__(self, idx):
    # Image path
    img_name = self.image_files[idx]
    img_path = os.path.join(self.image_dir, img_name)

    # Load image
    image = Image.open(img_path).convert("RGB")

    # Annotation path
    annotation_name = os.path.splitext(img_name)[0] + ".xml"
    annotation_path = os.path.join(self.annotations_dir, annotation_name)

    # Parse annotation
    label, bbox = self.parse_annotation(annotation_path)  # Get both label and bbox

    if self.transform:
        image = self.transform(image)

    return image, label, bbox
```

Classification + Localization

❖ Dataset

```xml
<annotation>
    <folder>images</folder>
    <filename>Cats_Test0.png</filename>
    <size>
        <width>233</width>
        <height>350</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>cat</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <occluded>0</occluded>
        <difficult>0</difficult>
        <bndbox>
            <xmin>83</xmin>
            <ymin>29</ymin>
            <xmax>197</xmax>
            <ymax>142</ymax>
        </bndbox>
    </object>
</annotation>
```

```python
def parse_annotation(self, annotation_path):
    tree = ET.parse(annotation_path)
    root = tree.getroot()

    # Get image size for normalization
    image_width = int(root.find('size/width').text)
    image_height = int(root.find('size/height').text)

    label = None
    bbox = None
    for obj in root.findall('object'):
        name = obj.find('name').text
        if label is None:  # Take the first label
            label = name
            # Get bounding box coordinates
            xmin = int(obj.find('bndbox/xmin').text)
            ymin = int(obj.find('bndbox/ymin').text)
            xmax = int(obj.find('bndbox/xmax').text)
            ymax = int(obj.find('bndbox/ymax').text)

            # Normalize bbox coordinates to [0, 1]
            bbox = [
                xmin / image_width,
                ymin / image_height,
                xmax / image_width,
                ymax / image_height,
            ]

    # Convert label to numerical representation (0 for cat, 1 for dog)
    label_num = 0 if label == 'cat' else 1 if label == 'dog' else -1

    return label_num, torch.tensor(bbox, dtype=torch.float32)
```

Classification + Localization

❖ Model

```python
# Model with Two Heads
class TwoHeadedModel(nn.Module):
    def __init__(self, num_classes=2):
        super(TwoHeadedModel, self).__init__()
        self.base_model = models.resnet18(weights=ResNet18_Weights.DEFAULT)
        self.num_ftrs = self.base_model.fc.in_features

        # Remove the original fully connected layer
        self.base_model.fc = nn.Identity()

        # Classification head
        self.classifier = nn.Linear(self.num_ftrs, num_classes)

        # Bounding box regression head
        self.regressor = nn.Linear(self.num_ftrs, 4)

    def forward(self, x):
        x = self.base_model(x)
        class_logits = self.classifier(x)
        bbox_coords = torch.sigmoid(self.regressor(x))
        return class_logits, bbox_coords

```

```python
# Model
model = TwoHeadedModel()

# Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Loss and Optimizer
criterion_class = nn.CrossEntropyLoss()
criterion_bbox = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

```

Classification + Localization

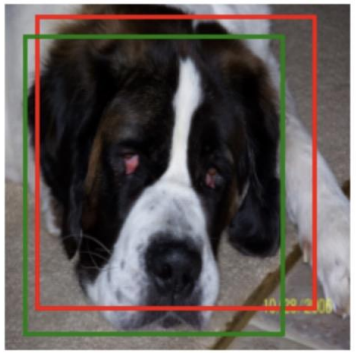
❖ Training

```python
# Training Loop
num_epochs = 10
for epoch in tqdm.tqdm(range(num_epochs), desc="Epochs"):
    model.train()
    for batch_idx, (data, targets, bboxes) in enumerate(tqdm.tqdm(train_loader, desc="Training", leave=False)):
        data = data.to(device)
        targets = targets.to(device)
        bboxes = bboxes.to(device)

        scores, pred_bboxes = model(data)
        loss_class = criterion_class(scores, targets)
        loss_bbox = criterion_bbox(pred_bboxes, bboxes)
        loss = loss_class + loss_bbox  # Combine losses

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        total_loss_bbox = 0
        total_samples = 0
        for data, targets, bboxes in tqdm.tqdm(val_loader, desc="Validation", leave=False):
            data = data.to(device)
            targets = targets.to(device)
            bboxes = bboxes.to(device)

            scores, pred_bboxes = model(data)
            _, predictions = scores.max(1)
            correct += (predictions == targets).sum()
            total += targets.size(0)

            # Calculate bbox loss for monitoring (optional)
            total_loss_bbox += criterion_bbox(pred_bboxes, bboxes).item() * data.size(0)
            total_samples += data.size(0)

        avg_loss_bbox = total_loss_bbox / total_samples

        print(f'Epoch {epoch+1}/{num_epochs}, Validation Accuracy: {float(correct)/float(total)*100:.2f}%, '
              f'Avg. Bbox Loss: {avg_loss_bbox:.4f}')
```

Classification + Localization

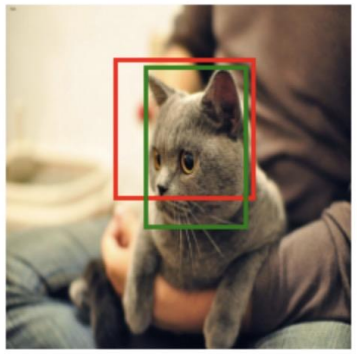
❖ Inference

Predicted: dog, Actual: dog



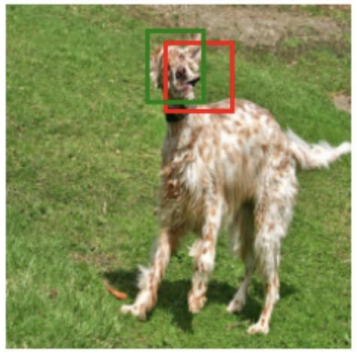
Predicted: dog, Actual: dog

Predicted: cat, Actual: cat



Predicted: dog, Actual: dog

Predicted: cat, Actual: dog

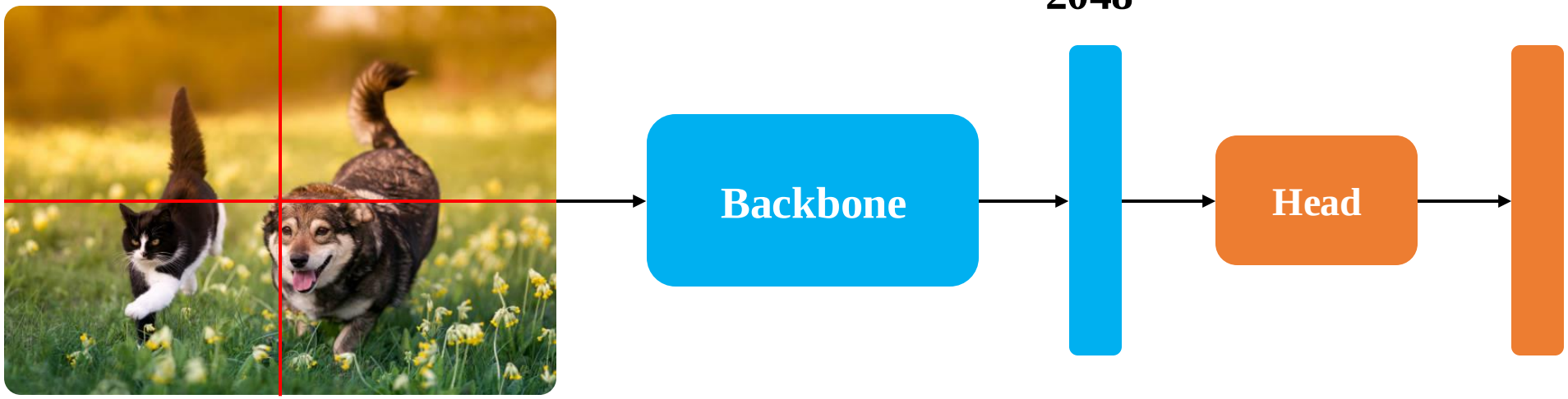


Predicted: dog, Actual: dog

Classification (>1 object) + Localization

Classification (>1 object) + Localization

❖ Introduction



n_cell * (n_class + bbox)

4 * (2 + 4)

Classification (>1 object) + Localization

❖ Dataset

```python
class MyDataset(Dataset):
    def __init__(self, annotations_dir, image_dir, transform=None):
        self.annotations_dir = annotations_dir
        self.image_dir = image_dir
        self.transform = transform
        self.image_files = self.filter_images_with_multiple_objects()

    def filter_images_with_multiple_objects(self):
        valid_image_files = []
        for f in os.listdir(self.image_dir):
            if os.path.isfile(os.path.join(self.image_dir, f)):
                img_name = f
                annotation_name = os.path.splitext(img_name)[0] + ".xml"
                annotation_path = os.path.join(self.annotations_dir, annotation_name)

                if self.count_objects_in_annotation(annotation_path) == 1:
                    valid_image_files.append(img_name)
        return valid_image_files

    def count_objects_in_annotation(self, annotation_path):
        try:
            tree = ET.parse(annotation_path)
            root = tree.getroot()
            count = 0
            for obj in root.findall("object"):
                count += 1
            return count
        except FileNotFoundError:
            return 0
```

```python
    def parse_annotation(self, annotation_path):
        tree = ET.parse(annotation_path)
        root = tree.getroot()

        # Get image size for normalization
        image_width = int(root.find("size/width").text)
        image_height = int(root.find("size/height").text)

        label = None
        bbox = None
        for obj in root.findall("object"):
            name = obj.find("name").text
            if label is None:  # Take the first label
                label = name
                # Get bounding box coordinates
                xmin = int(obj.find("bndbox/xmin").text)
                ymin = int(obj.find("bndbox/ymin").text)
                xmax = int(obj.find("bndbox/xmax").text)
                ymax = int(obj.find("bndbox/ymax").text)

                # Normalize bbox coordinates to [0, 1]
                bbox = [
                    xmin / image_width,
                    ymin / image_height,
                    xmax / image_width,
                    ymax / image_height,
                ]

        # Convert label to numerical representation (0 for cat, 1 for dog)
        label_num = 0 if label == "cat" else 1 if label == "dog" else -1

        return label_num, torch.tensor(bbox, dtype=torch.float32)

    def __len__(self):
        return len(self.image_files)
```

Classification (>1 object) + Localization


```python
def __getitem__(self, idx):
    img1_file = self.image_files[idx]
    img1_path = os.path.join(self.image_dir, img1_file)

    annotation_name = os.path.splitext(img1_file)[0] + ".xml"
    img1_annotations = self.parse_annotation(
        os.path.join(self.annotations_dir, annotation_name)
    )

    idx2 = random.randint(0, len(self.image_files) - 1)
    img2_file = self.image_files[idx2]
    img2_path = os.path.join(self.image_dir, img2_file)

    annotation_name = os.path.splitext(img2_file)[0] + ".xml"
    img2_annotations = self.parse_annotation(
        os.path.join(self.annotations_dir, annotation_name)
    )

    img1 = Image.open(img1_path).convert("RGB")
    img2 = Image.open(img2_path).convert("RGB")

    # Horizontal merge
    merged_image = Image.new(
        "RGB", (img1.width + img2.width, max(img1.height, img2.height))
    )
    merged_image.paste(img1, (0, 0))
    merged_image.paste(img2, (img1.width, 0))
    merged_w = img1.width + img2.width
    merged_h = max(img1.height, img2.height)

    merged_annotations = []

    # No change for objects from img1, already normalized
    merged_annotations.append(
        {"bbox": img1_annotations[1].tolist(), "label": img1_annotations[0]}
    )

    # Adjust bbox coordinates for objects from img2 AND normalize
    new_bbox = [
        (img2_annotations[1][0] * img2.width + img1.width) / merged_w,  # Normalize xmin
        img2_annotations[1][1] * img2.height / merged_h,                # Normalize ymin
        (img2_annotations[1][2] * img2.width + img1.width) / merged_w,  # Normalize xmax
        img2_annotations[1][3] * img2.height / merged_h,                # Normalize ymax
    ]

    merged_annotations.append({"bbox": new_bbox, "label": img2_annotations[0]})

    # Convert merged image to tensor
    if self.transform:
        merged_image = self.transform(merged_image)
    else:
        merged_image = transforms.ToTensor()(merged_image)

    # Convert annotations to 1D tensors, with shape (4,) for bbox and (1,) for label
    annotations = torch.zeros((len(merged_annotations), 5))
    for i, ann in enumerate(merged_annotations):
        annotations[i] = torch.cat((torch.tensor(ann["bbox"]), torch.tensor([ann["label"]])))

    return merged_image, annotations
```

Classification (>1 object) + Localization

❖ Model

```python
class SimpleYOLO(nn.Module):
    def __init__(self, num_classes):
        super(SimpleYOLO, self).__init__()
        self.backbone = models.resnet50(weights=ResNet50_Weights.DEFAULT)
        self.num_classes = num_classes

        # Remove the final classification layer of ResNet
        self.backbone = nn.Sequential(*list(self.backbone.children())[:-2])

        # Add the YOLO head
        self.fcs = nn.Linear(
            2048, 2 * 2 * (4 + self.num_classes)
        )  # 2 is for the number of grid cell

    def forward(self, x):
        # x shape: (batch_size, C, H, W)
        features = self.backbone(x)
        features = F.adaptive_avg_pool2d(features, (1, 1)) # shape: (batch_size, 2048, 1, 1)
        features = features.view(features.size(0), -1) # shape: (batch_size, 2048)
        features = self.fcs(features)

        return features

```

❖ Loss

```python
def calculate_loss(output, targets, device, num_classes):
    mse_loss = nn.MSELoss()
    ce_loss = nn.CrossEntropyLoss()

    batch_size = output.shape[0]
    total_loss = 0

    output = output.view(batch_size, 2, 2, 4 + num_classes)  # Reshape to (batch_size, grid_y, grid_x, 4 + num_classes)

    for i in range(batch_size):  # Iterate through each image in the batch
        for j in range(len(targets[i])): # Iterate through objects in the image

            # Determine which grid cell the object's center falls into
            # Assuming bbox coordinates are normalized to [0, 1]
            bbox_center_x = (targets[i][j][0] + targets[i][j][2]) / 2
            bbox_center_y = (targets[i][j][1] + targets[i][j][3]) / 2

            grid_x = int(bbox_center_x * 2)  # Multiply by number of grid cells (2 in this case)
            grid_y = int(bbox_center_y * 2)

            # 1. Classification Loss for the responsible grid cell
            # Convert label to one-hot encoding only for this example
            # One hot encoding
            label_one_hot = torch.zeros(num_classes, device=device)
            label_one_hot[int(targets[i][j][4])] = 1

            # Classification loss (using CrossEntropyLoss)
            classification_loss = ce_loss(output[i, grid_y, grid_x, 4:], label_one_hot)

            # 2. Regression Loss for the responsible grid cell
            bbox_target = targets[i][j][:4].to(device)
            regression_loss = mse_loss(output[i, grid_y, grid_x, :4], bbox_target)

            # 3. No Object Loss (for other grid cells)
            no_obj_loss = 0
            for other_grid_y in range(2):
                for other_grid_x in range(2):
                    if other_grid_y != grid_y or other_grid_x != grid_x:
                        # MSE loss for predicting no object (all zeros)
                        no_obj_loss += mse_loss(output[i, other_grid_y, other_grid_x, :4], torch.zeros(4, device=device))

            total_loss += classification_loss + regression_loss + no_obj_loss

    return total_loss / batch_size  # Average loss over the batch

```

Classification (>1 object) + Localization

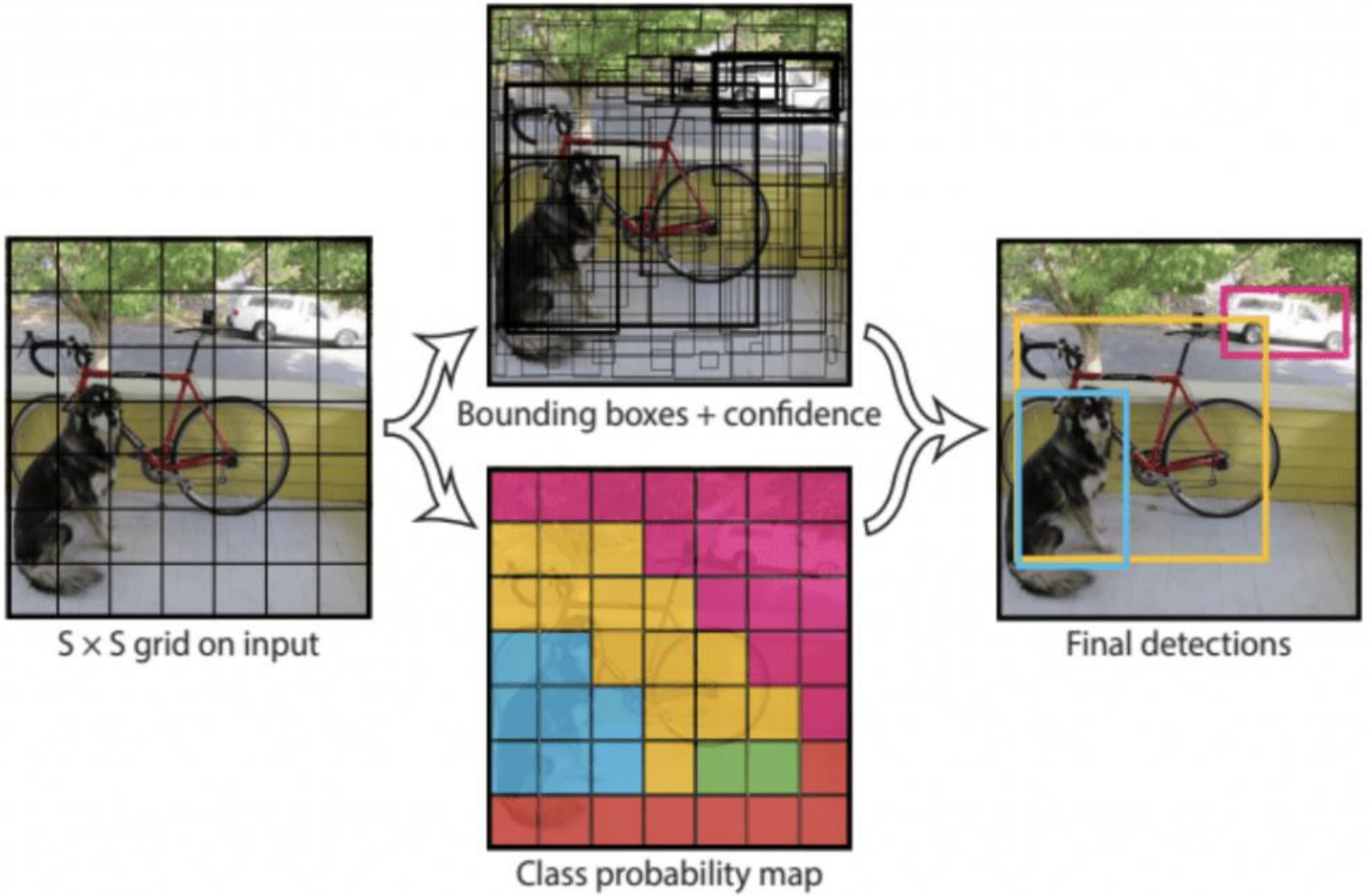

```python
def train_model(
    model, train_loader, val_loader, optimizer, num_epochs, device, num_classes
):
    best_val_accuracy = 0.0
    train_losses = []
    val_losses = []
    train_accuracies = []
    val_accuracies = []

    for epoch in tqdm.tqdm(range(num_epochs), desc="Epochs"):
        model.train()
        running_loss = 0.0

        for images, targets in tqdm.tqdm(train_loader, desc="Batches", leave=False):
            images = images.to(device)

            optimizer.zero_grad()
            output = model(images)

            total_loss = calculate_loss(output, targets, device, num_classes)

            total_loss.backward()
            optimizer.step()
            running_loss += total_loss.item()

        epoch_loss = running_loss / len(train_loader)
        train_losses.append(epoch_loss)

        # Validation
        val_loss, val_accuracy = evaluate_model(model, val_loader, device, num_classes)
        val_losses.append(val_loss)
        val_accuracies.append(val_accuracy)

        # Save the best model
        if val_accuracy > best_val_accuracy:
            best_val_accuracy = val_accuracy
            torch.save(model.state_dict(), "best_model.pth")

    return train_losses, val_losses, train_accuracies, val_accuracies

```

```python
def evaluate_model(model, data_loader, device, num_classes):
    model.eval()
    running_loss = 0.0
    all_predictions = []
    all_targets = []

    with torch.no_grad():
        for images, targets in tqdm.tqdm(data_loader, desc="Validation", leave=False):
            images = images.to(device)

            output = model(images)

            total_loss = calculate_loss(output, targets, device, num_classes)
            running_loss += total_loss.item()

            # Reshape output to (batch_size, grid_y, grid_x, 4 + num_classes)
            output = output.view(images.shape[0], 2, 2, 4 + num_classes)

            # Collect predictions and targets for mAP calculation
            for batch_idx in range(images.shape[0]):
                for target in targets[batch_idx]:
                    # Determine responsible grid cell
                    bbox_center_x = (target[0] + target[2]) / 2
                    bbox_center_y = (target[1] + target[3]) / 2
                    grid_x = int(bbox_center_x * 2)
                    grid_y = int(bbox_center_y * 2)

                    # Class prediction (index of max probability)
                    prediction = output[batch_idx, grid_y, grid_x, 4:].argmax().item()
                    all_predictions.append(prediction)

                    all_targets.append(target[4].item())

    val_loss = running_loss / len(data_loader)

    # Convert lists to tensors for PyTorch's metric functions
    all_predictions = torch.tensor(all_predictions, device=device)
    all_targets = torch.tensor(all_targets, device=device)

    # Calculate accuracy
    val_accuracy = (all_predictions == all_targets).float().mean()

    return val_loss, val_accuracy.item()

```

Basic YOLOv1

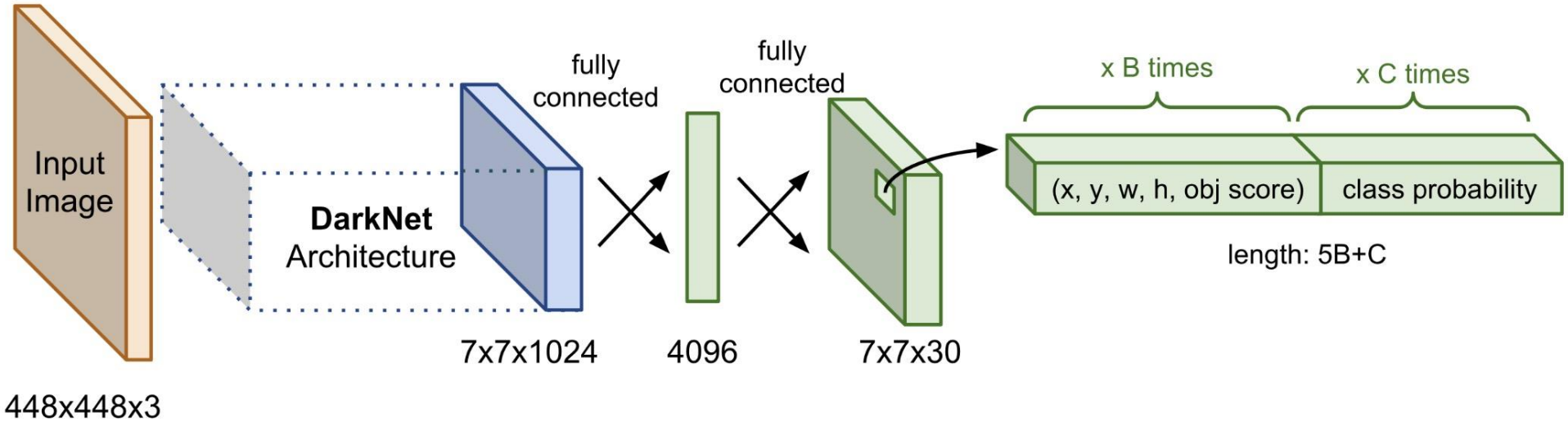
Basic YOLO

❖ Motivation



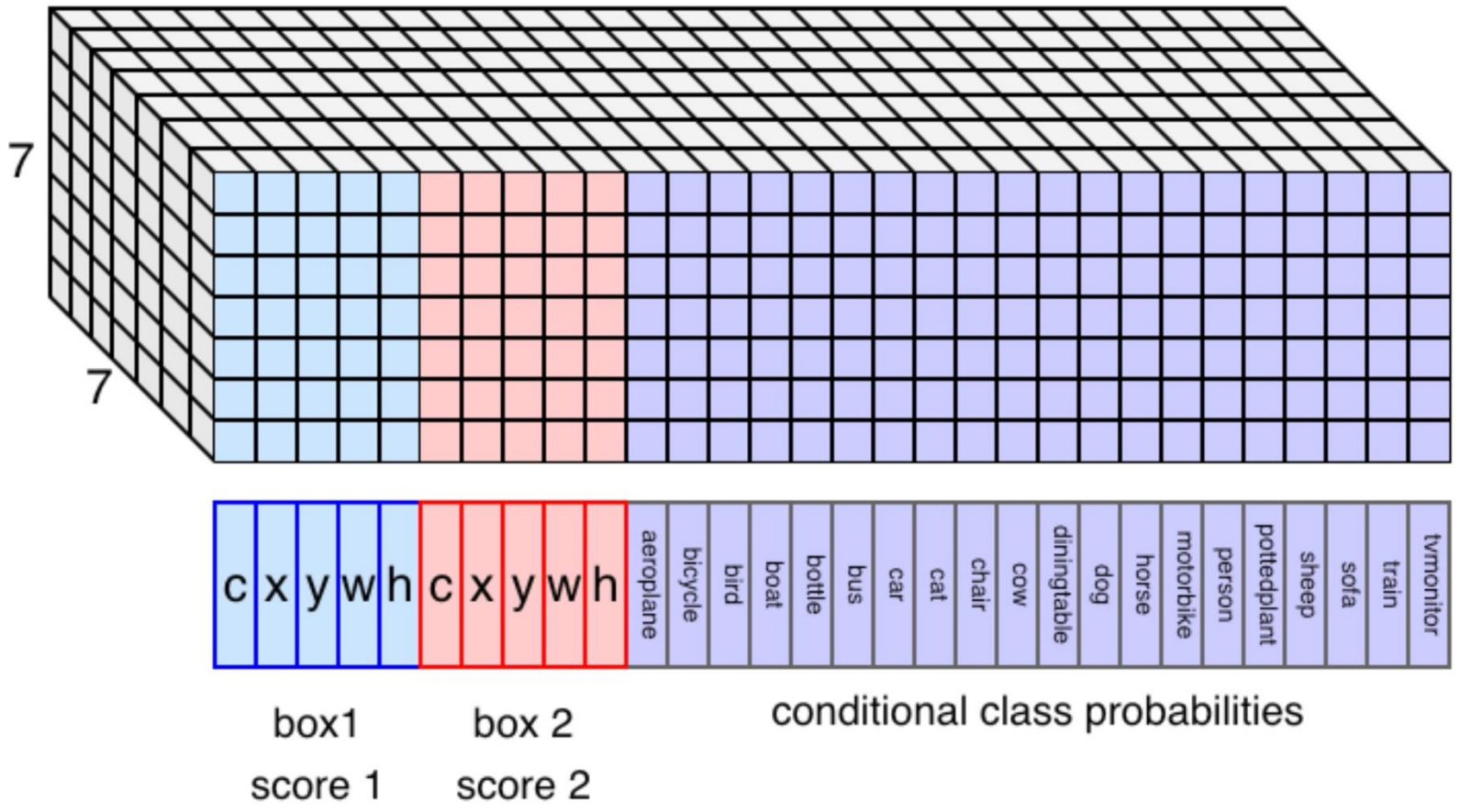
Basic YOLO

❖ Model



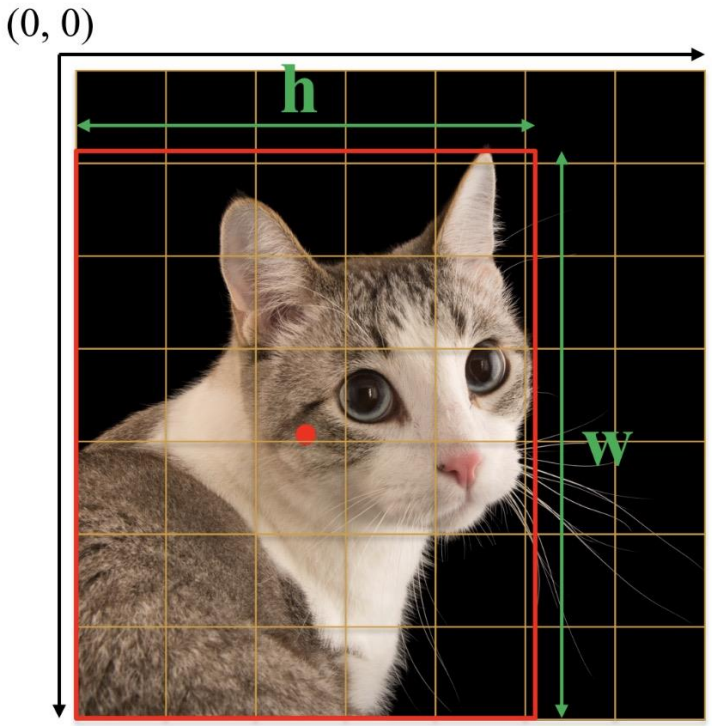
Basic YOLO

❖ Output

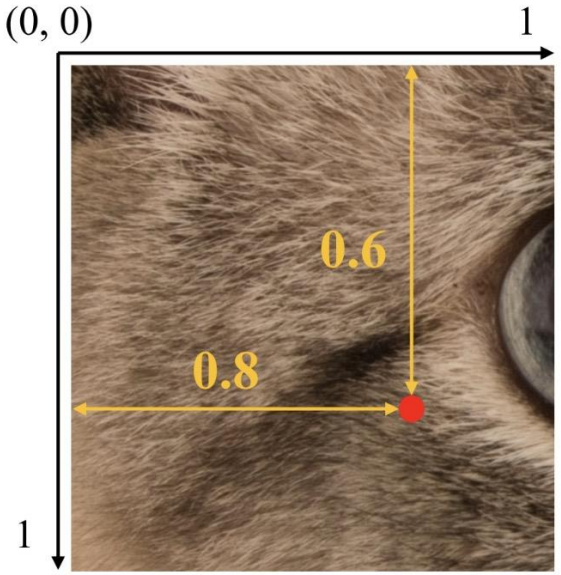


Basic YOLO

❖ Bbox



rescale



x	y	w	h
0.8	0.6	5.1	6.2

❖ Dataset

```python
class CustomVOCDataset(torchvision.datasets.VOCDetection):
    def init_config_yolo(self, class_mapping, S=7, B=2, C=20, custom_transforms=None):
        # Initialize YOLO-specific configuration parameters.
        self.S = S  # Grid size S x S
        self.B = B  # Number of bounding boxes
        self.C = C  # Number of classes
        self.class_mapping = class_mapping  # Mapping of class names to class indices
        self.custom_transforms = custom_transforms

```

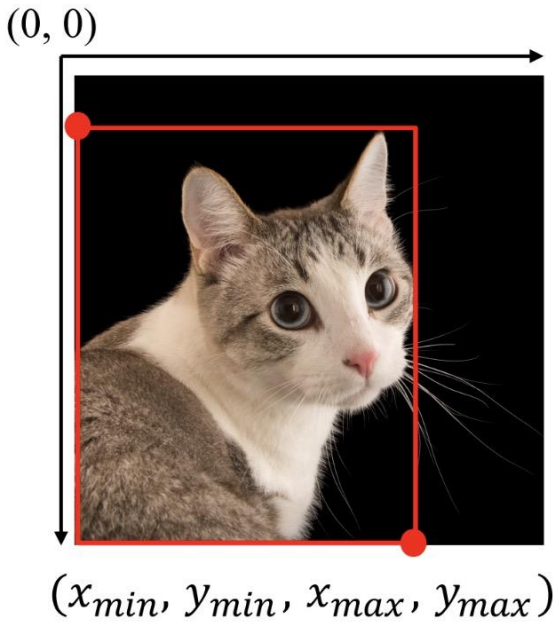
❖ Dataset

```python
    def __getitem__(self, index):
        # Get an image and its target (annotations) from the VOC dataset.
        image, target = super(CustomVOCDataset, self).__getitem__(index)
        img_width, img_height = image.size

        # Convert target annotations to YOLO format bounding boxes.
        boxes = convert_to_yolo_format(target, img_width, img_height, self.class_mapping)

        just_boxes = boxes[:,1:]
        labels = boxes[:,0]

        # Transform the image and bounding boxes
        if self.custom_transforms:
            sample = {
            'image': np.array(image),
            'bboxes': just_boxes,
            'labels': labels
            }

            sample = self.custom_transforms(**sample)
            image = sample['image']
            boxes = sample['bboxes']
            labels = sample['labels']
```

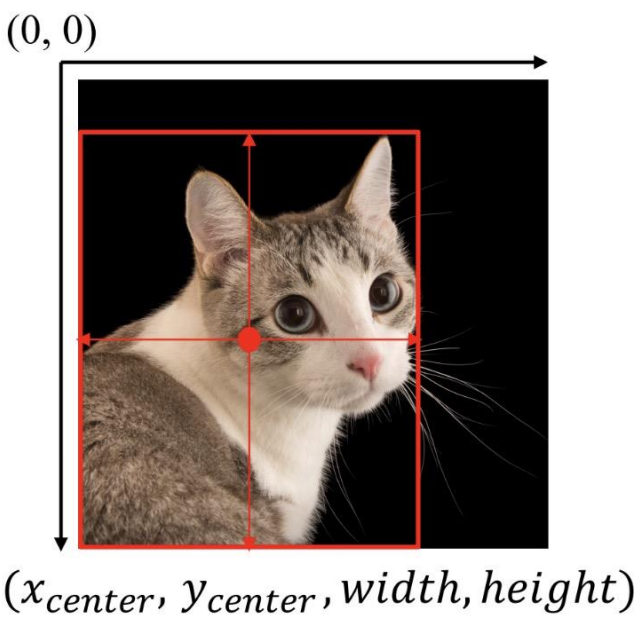
Basic YOLO

❖ Dataset

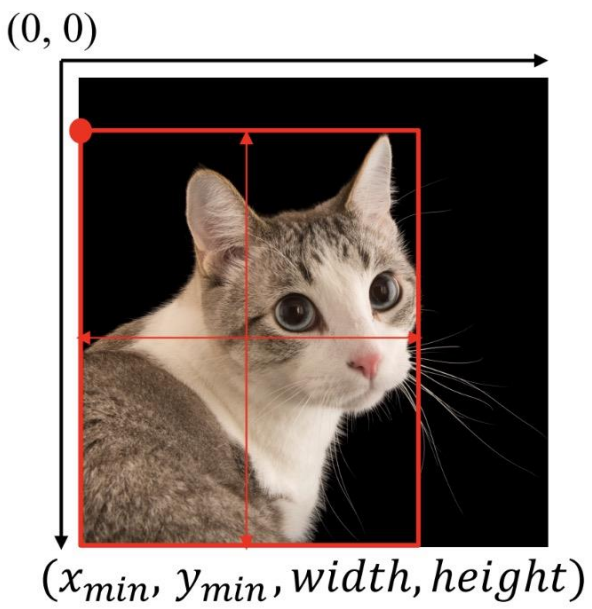
VOC



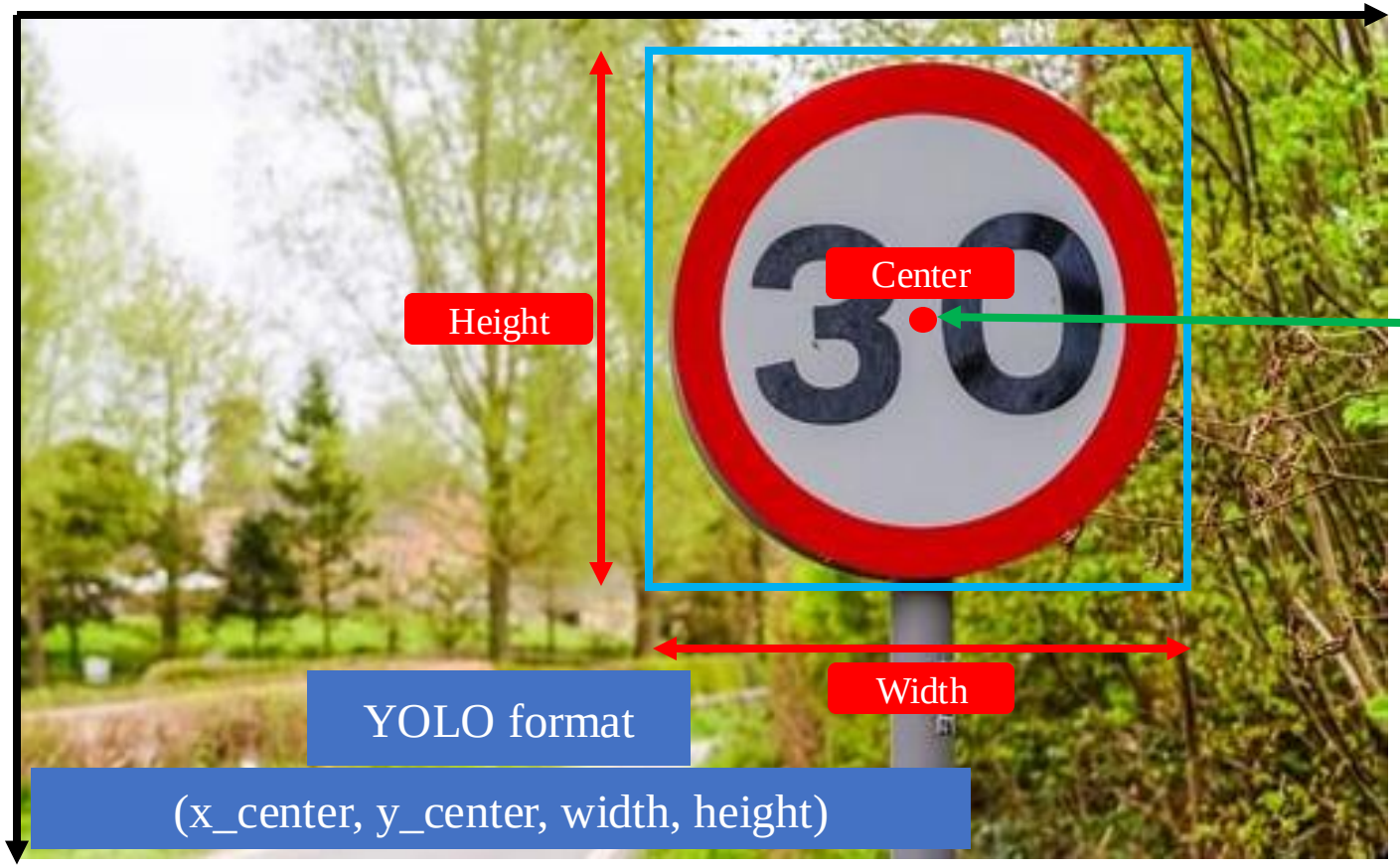
YOLO



COCO



❖ Dataset



Format of YOLO

AI VIET NAM
@aivietnam.edu.vn

❖ Dataset

Detailed Explanation

1. **Input Values**:

```
x, y = box coordinates # normalized coordinates (0-1) representing center of object
self.S = 7 # grid size (7×7 grid)
```

2. **Grid Cell Calculation**:

```
i, j = int(self.S * y), int(self.S * x)
```

- `self.S * x` : Scales the normalized x-coordinate (0-1) to grid coordinates (0-7)
- `self.S * y` : Scales the normalized y-coordinate (0-1) to grid coordinates (0-7)
- `int()` : Floors the values to get the grid cell indices

Example:

- If x = 0.3, y = 0.6 and S = 7:
  - j = int(7 *0.3) = int(2.1) = 2 (column index)*
  - i = int(7 *0.6) = int(4.2) = 4 (row index)*

```python
    # Create an empty label matrix for YOLO ground truth.
    label_matrix = torch.zeros((self.S, self.S, self.C + 5 * self.B))

    boxes = torch.tensor(boxes, dtype=torch.float32)
    labels = torch.tensor(labels, dtype=torch.float32)
    image  = torch.as_tensor(image, dtype=torch.float32)

    # Iterate through each bounding box in YOLO format.
    for box, class_label in zip(boxes, labels):
        x, y, width, height = box.tolist()
        class_label = int(class_label)

        # Calculate the grid cell (i, j) that this box belongs to.
        i, j = int(self.S * y), int(self.S * x)
        x_cell, y_cell = self.S * x - j, self.S * y - i

        # Calculate the width and height of the box relative to the grid cell.
        width_cell, height_cell = (
            width * self.S,
            height * self.S,
        )

        # If no object has been found in this specific cell (i, j) before:
        if label_matrix[i, j, 20] == 0:
            # Mark that an object exists in this cell.
            label_matrix[i, j, 20] = 1

            # Store the box coordinates as an offset from the cell boundaries.
            box_coordinates = torch.tensor(
                [x_cell, y_cell, width_cell, height_cell]
            )

            # Set the box coordinates in the label matrix.
            label_matrix[i, j, 21:25] = box_coordinates

            # Set the one-hot encoding for the class label.
            label_matrix[i, j, class_label] = 1

    return image, label_matrix
```

❖ Dataset

3. **Cell-Relative Coordinates**:

```
x_cell, y_cell = self.S * x - j, self.S * y - i
```

This calculates the object's position relative to the grid cell's top-left corner:

- `self.S * x - j` : Horizontal offset within the cell (0-1)
- `self.S * y - i` : Vertical offset within the cell (0-1)

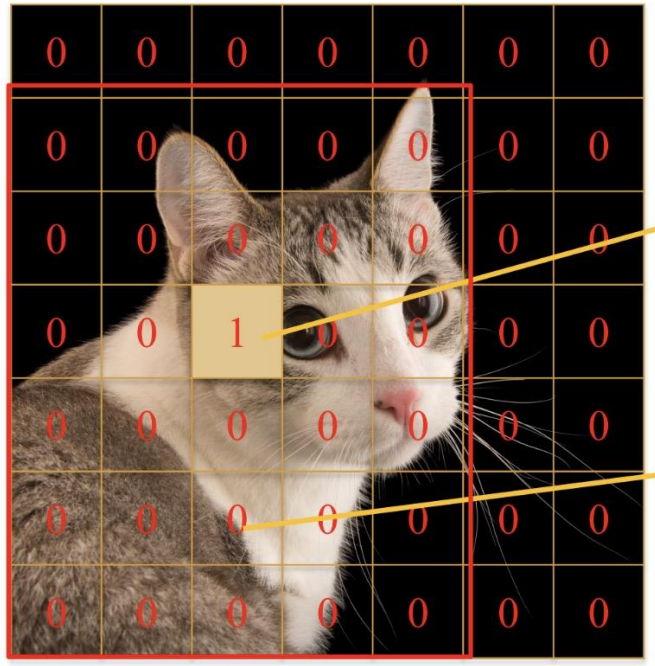
Example continued:

- x_cell = *7 0.3 - 2 = 2.1 - 2 = 0.1*
- y_cell = *7 0.6 - 4 = 4.2 - 4 = 0.2*

```
34      # Create an empty label matrix for YOLO ground truth.
35      label_matrix = torch.zeros((self.S, self.S, self.C + 5 * self.B))
36
37      boxes = torch.tensor(boxes, dtype=torch.float32)
38      labels = torch.tensor(labels, dtype=torch.float32)
39      image  = torch.as_tensor(image, dtype=torch.float32)
40
41      # Iterate through each bounding box in YOLO format.
42      for box, class_label in zip(boxes, labels):
43          x, y, width, height = box.tolist()
44          class_label = int(class_label)
45
46          # Calculate the grid cell (i, j) that this box belongs to.
47          i, j = int(self.S * y), int(self.S * x)
48          x_cell, y_cell = self.S * x - j, self.S * y - i
49
50          # Calculate the width and height of the box relative to the grid cell.
51          width_cell, height_cell = (
52              width * self.S,
53              height * self.S,
54          )
55
56          # If no object has been found in this specific cell (i, j) before:
57          if label_matrix[i, j, 20] == 0:
58              # Mark that an object exists in this cell.
59              label_matrix[i, j, 20] = 1
60
61              # Store the box coordinates as an offset from the cell boundaries.
62              box_coordinates = torch.tensor(
63                  [x_cell, y_cell, width_cell, height_cell]
64              )
65
66              # Set the box coordinates in the label matrix.
67              label_matrix[i, j, 21:25] = box_coordinates
68
69              # Set the one-hot encoding for the class label.
70              label_matrix[i, j, class_label] = 1
71
72      return image, label_matrix
```

Basic YOLO

❖ Target for training



For example, we have 2 classes: *cat* and *dog*.

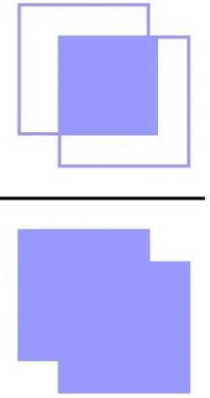
cat	dog		bounding box (x, y, w, h)			
1	0	1	0.8	0.6	5.1	6.2
		conf				

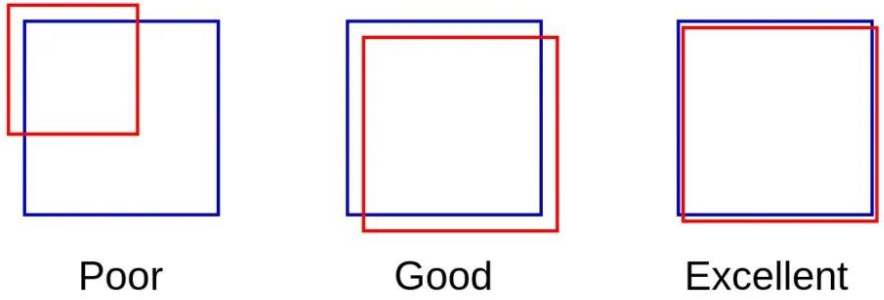
0	0	0	0	0	0	0

For each object, **only 1 grid cell** is responsible for the prediction.

❖ IoU

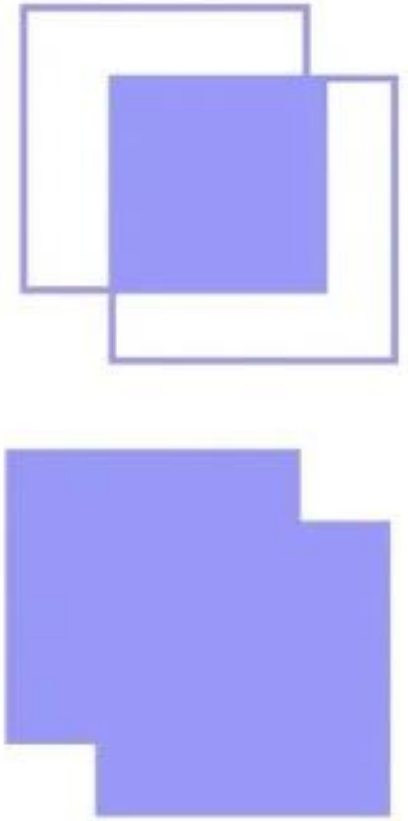
$$IoU = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$





❖ IoU

```python
def intersection_over_union(boxes_preds, boxes_labels, box_format="midpoint"):
    """…
    # Check if the box format is "midpoint"
    if box_format == "midpoint":…
    # Check if the box format is "corners"
    if box_format == "corners":…
    # Calculate coordinates of the intersection rectangle
    x1 = torch.max(box1_x1, box2_x1)
    y1 = torch.max(box1_y1, box2_y1)
    x2 = torch.min(box1_x2, box2_x2)
    y2 = torch.min(box1_y2, box2_y2)
    # Compute the area of the intersection rectangle, clamp(0) to handle cases where they do not overlap
    intersection = (x2 - x1).clamp(0) * (y2 - y1).clamp(0)
    # Calculate the areas of the predicted and ground truth boxes
    box1_area = abs((box1_x2 - box1_x1) * (box1_y2 - box1_y1))
    box2_area = abs((box2_x2 - box2_x1) * (box2_y2 - box2_y1))
    # Calculate the Intersection over Union, adding a small epsilon to avoid division by zero
    return intersection / (box1_area + box2_area - intersection + 1e-6)
```



$$IoU = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

❖ Non-Maximum Suppression (NMS)


```python
def non_max_suppression(bboxes, iou_threshold, confidence_threshold, box_format="corners"):
    """
    Perform Non-Maximum Suppression (NMS) to filter overlapping bounding boxes.

    Args:
        bboxes (list): List of bounding boxes, each containing [class_id, confidence, x1, y1, x2, y2]
        iou_threshold (float): IoU threshold to determine box overlap
        confidence_threshold (float): Minimum confidence score to keep a box
        box_format (str): Format of box coordinates - "corners" or "midpoint"

    Returns:
        list: Filtered list of bounding boxes after NMS
    """
    if not isinstance(bboxes, list):
        raise TypeError("bboxes must be a list")

    # Keep only high confidence predictions
    confident_boxes = [box for box in bboxes if box[1] > confidence_threshold]

    # Sort by confidence score (highest first)
    confident_boxes.sort(key=lambda x: x[1], reverse=True)

    kept_boxes = []

    while confident_boxes:
        # Take box with highest confidence
        best_box = confident_boxes.pop(0)

        # Keep remaining boxes that either:
        # 1. Are from a different class OR
        # 2. Don't overlap significantly with the best box
        confident_boxes = [
            box for box in confident_boxes
            if (box[0] != best_box[0]) or  # Different class
               (intersection_over_union(
                   torch.tensor(best_box[2:]),
                   torch.tensor(box[2:]),
                   box_format=box_format
               ) < iou_threshold)  # Low overlap
        ]

        kept_boxes.append(best_box)

    return kept_boxes
```

❖ Non-Maximum Suppression (NMS)

NMS Process:

1. **Initial Filtering** (confidence > 0.5):
   - All boxes pass (0.9, 0.8, 0.7, 0.85 > 0.5)

2. **Sorting by confidence**:

```
bboxes = [
    [0, 0.9, 0.2, 0.3, 0.4, 0.5],    # Highest confidence
    [1, 0.85, 0.7, 0.7, 0.9, 0.9],
    [0, 0.8, 0.21, 0.31, 0.41, 0.51],
    [0, 0.7, 0.19, 0.29, 0.39, 0.49]
]
```

3. **First Iteration**:
   - Choose first box (0.9 confidence)
   - Compare with remaining boxes:
     - Box 2 (0.85): Different class (1 vs 0) → Keep
     - Box 3 (0.8): Same class, high IoU → Remove
     - Box 4 (0.7): Same class, high IoU → Remove

4. **Second Iteration**:
   - Only box with class 1 remains
   - Added to final list

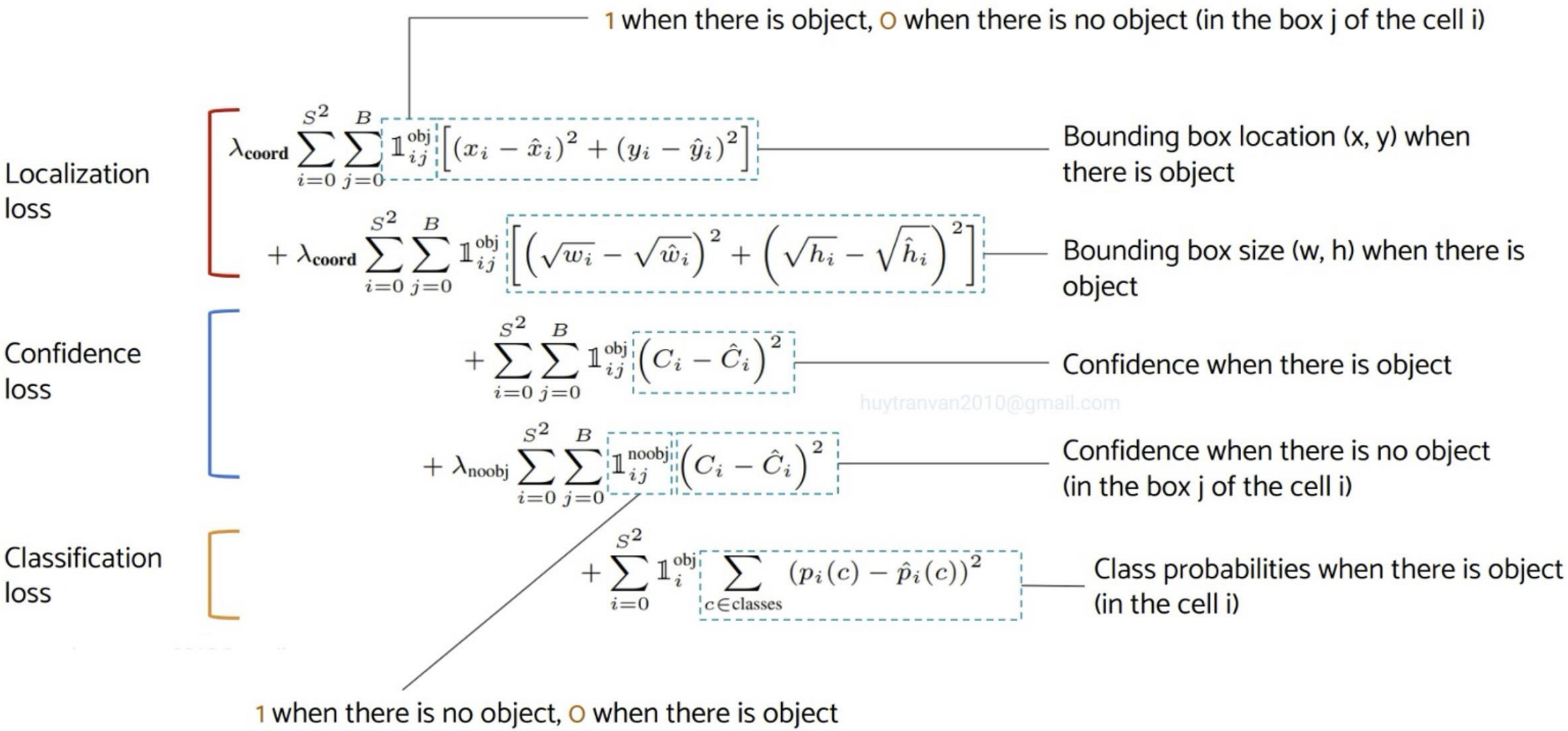
Final Output:

```
bboxes_after_nms = [
    [0, 0.9, 0.2, 0.3, 0.4, 0.5],    # Best box for class 0
    [1, 0.85, 0.7, 0.7, 0.9, 0.9]     # Best box for class 1
]
```

NMS Benefits:

- Removes duplicate detections
- Keeps highest confidence predictions
- Handles multiple classes independently
- Improves precision by reducing false positives

❖ Loss


1 when there is object, 0 when there is no object (in the box j of the cell i)
Localization loss
$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$
Bounding box location (x, y) when there is object
$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$
Bounding box size (w, h) when there is object
Confidence loss
$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$
Confidence when there is object
$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$
Confidence when there is no object (in the box j of the cell i)
Classification loss
$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2$
Class probabilities when there is object (in the cell i)
1 when there is no object, 0 when there is object

```python
class YoloLoss(nn.Module):
    def __init__(self, S=7, B=2, C=20):…

    def forward(self, predictions, target):
        # Reshape the predictions to the shape (BATCH_SIZE, S*S(C+B*5))
        predictions = predictions.reshape(-1, self.S, self.S, self.C + self.B * 5)

        # Calculate Intersection over Union (IoU) for the two predicted bounding boxes
        # with the target bounding box.
        iou_b1 = intersection_over_union(predictions[..., 21:25], target[..., 21:25])
        iou_b2 = intersection_over_union(predictions[..., 26:30], target[..., 21:25])
        ious = torch.cat([iou_b1.unsqueeze(0), iou_b2.unsqueeze(0)], dim=0)

        # Get the box with the highest IoU among the two predictions.
        # Note that bestbox will have an index of 0 or 1, indicating which box is better.
        iou_maxes, bestbox = torch.max(ious, dim=0)
        exists_box = target[..., 20].unsqueeze(3)  # This represents Iobj_i in the paper

        # ======================== #
        #   FOR BOX COORDINATES    #
        # ======================== #

        # Set the boxes with no objects to zero. Choose one of the two predictions
        # based on the bestbox index calculated earlier.
        box_predictions = exists_box * (
            (
                bestbox * predictions[..., 26:30]
                + (1 - bestbox) * predictions[..., 21:25]
            )
        )
        box_targets = exists_box * target[..., 21:25]

        # Take the square root of width and height to ensure positive values.
        box_predictions[..., 2:4] = torch.sign(box_predictions[..., 2:4]) * torch.sqrt(
            torch.abs(box_predictions[..., 2:4] + 1e-6)
        )
        box_targets[..., 2:4] = torch.sqrt(box_targets[..., 2:4])

        box_loss = self.mse(
            torch.flatten(box_predictions, end_dim=-2),
            torch.flatten(box_targets, end_dim=-2),
        )

        # ====================== #
        #   FOR OBJECT LOSS    #
        # ====================== #
        # pred_box represents the confidence score of the box with the highest IoU.
        pred_box = (
            bestbox * predictions[..., 25:26] + (1 - bestbox) * predictions[..., 20:21]
        )
        object_loss = self.mse(
            torch.flatten(exists_box * pred_box),
            torch.flatten(exists_box * target[..., 20:21]),
        )

        # ======================== #
        #   FOR NO OBJECT LOSS    #
        # ======================== #
        no_object_loss = self.mse(
            torch.flatten((1 - exists_box) * predictions[..., 20:21], start_dim=1),
            torch.flatten((1 - exists_box) * target[..., 20:21], start_dim=1),
        )
        no_object_loss += self.mse(
            torch.flatten((1 - exists_box) * predictions[..., 25:26], start_dim=1),
            torch.flatten((1 - exists_box) * target[..., 20:21], start_dim=1)
        )

        # ================== #
        #   FOR CLASS LOSS   #
        # ================== #
        class_loss = self.mse(
            torch.flatten(exists_box * predictions[..., :20], end_dim=-2,),
            torch.flatten(exists_box * target[..., :20], end_dim=-2,),
        )
        # Calculate the final loss by combining the above components.
        loss = (
            self.lambda_coord * box_loss  # First term
            + object_loss  # Second term
            + self.lambda_noobj * no_object_loss  # Third term
            + class_loss  # Fourth term
        )

        return loss
```

AI VIET NAM
@aivietnam.edu.vn

❖ Model

```python
class Yolov1(nn.Module):
    def __init__(self, in_channels=3, **kwargs):
        """
        Initializes the YOLOv1 model with ResNet34 as the backbone.

        Args:
            in_channels (int): Number of input channels. Default is 3 for RGB images.
            **kwargs: Additional keyword arguments such as split_size, num_boxes, num_classes.
        """
        super(Yolov1, self).__init__()
        self.split_size = kwargs.get("split_size", 7)  # S
        self.num_boxes = kwargs.get("num_boxes", 2)  # B
        self.num_classes = kwargs.get("num_classes", 20)  # C

        # Initialize ResNet34 backbone from timm
        # `features_only=True` returns a list of feature maps from specified stages
        # `out_indices=[4]` corresponds to the last layer's output
        self.backbone = timm.create_model(
            "resnet34",
            pretrained=True,
            features_only=True,
            out_indices=[4],
            in_chans=in_channels,
        )

        # Adaptive pooling to ensure the spatial dimensions match (S x S)
        self.pool = nn.AdaptiveAvgPool2d((self.split_size, self.split_size))

        # Fully connected layers
        self.fcs = self._create_fcs()

    @torch.autocast(device_type="cuda", dtype=torch.float16)
    def forward(self, x):
        """
        Forward pass of the YOLOv1 model.

        Args:
            x (torch.Tensor): Input tensor of shape (N, C, H, W).

        Returns:
            torch.Tensor: Output tensor containing bounding box predictions.
        """
        # Extract features using ResNet34 backbone
        features = self.backbone(x)[
            0
        ]  # timm returns a list; take the first (and only) element

        # Apply adaptive pooling to match spatial dimensions
        pooled_features = self.pool(features)

        # Pass through fully connected layers
        output = self.fcs(pooled_features)

        return output

    def _create_fcs(self):
        """
        Creates the fully connected layers for YOLOv1.

        Returns:
            nn.Sequential: Sequential container of fully connected layers.
        """
        S, B, C = self.split_size, self.num_boxes, self.num_classes

        # Calculate the input features for the first linear layer
        # ResNet34's last feature map has 512 channels
        # After pooling to (S x S), the total features are 512 * S * S
        input_features = 512 * S * S

        return nn.Sequential(
            nn.Flatten(),
            nn.Linear(input_features, 4096),
            nn.Dropout(0.0),
            nn.LeakyReLU(0.1),
            nn.Linear(4096, S * S * (C + B * 5)),
        )

```


Thank you!