

# EXPLANATION

## IMAGE SEGMENTATION WITH UNET

Somoeurn Virakden

University of Nankai  
MSE: Subject AIoT

---

## 1 Introduction

The repository provides a PyTorch implementation of the U-Net model for semantic image segmentation. It includes instructions for training, prediction, and usage with or without Docker. The model was trained on the Carvana dataset, and a pretrained model is available. Key features include mixed-precision support for faster training, integration with Weights & Biases for tracking progress, and easy customization for different datasets. You can train the model using provided scripts, or use it for predicting segmentation masks on input images.

## 2 Overview of the Code Structure

The UNet implementation is divided into several key components:

### 2.1 Main Script

1. `train.py`: Script for training the model.
2. `predict.py`: Script for running inference (making predictions) on new images.

### 2.2 Model Definition

1. `unet_model.py`: U-Net architecture definition.
2. `unet_parts.py`: Dataset class for loading images and masks.

### 2.3 Data Sets

1. `imgs`: This folder contains the training set images in .jpg format.
2. `masks`: This folder contains the training set masks in .gif format.

### 2.4 Data Loading

- Directory containing data loading scripts and dataset-specific configurations.

### 2.5 Utils

- Contains utility functions like data loading, transformations, and evaluation metrics.

### 3 Model Definition

The core of the code resides in the `unet_model.py` file, where the U-Net model is implemented. Below is a breakdown of the different sections of the `unet_model.py` file.

#### 3.1 U-Net Architecture

The UNet architecture is implemented as a class `UNet`, which inherits from `torch.nn.Module`. The network consists of two main parts:

- **Encoder (Contracting Path):** This part captures context by downsampling the input.
- **Decoder (Expansive Path):** This part recovers the spatial resolution and performs segmentation.

```

1  class UNet(nn.Module):
2  def __init__(self, n_channels, n_classes, bilinear=False):
3      super(UNet, self).__init__()
4      self.n_channels = n_channels
5      self.n_classes = n_classes
6      self.bilinear = bilinear
7
8      self.inc = DoubleConv(n_channels, 64)
9      self.down1 = Down(64, 128)
10     self.down2 = Down(128, 256)
11     self.down3 = Down(256, 512)
12     factor = 2 if bilinear else 1
13     self.down4 = Down(512, 1024 // factor)
14     self.up1 = Up(1024, 512 // factor, bilinear)
15     self.up2 = Up(512, 256 // factor, bilinear)
16     self.up3 = Up(256, 128 // factor, bilinear)
17     self.up4 = Up(128, 64, bilinear)
18     self.outc = OutConv(64, n_classes)
19
20 def forward(self, x):
21     # Use gradient checkpointing here
22     x1 = checkpoint.checkpoint(self.inc, x)
23     x2 = checkpoint.checkpoint(self.down1, x1)
24     x3 = checkpoint.checkpoint(self.down2, x2)
25     x4 = checkpoint.checkpoint(self.down3, x3)
26     x5 = checkpoint.checkpoint(self.down4, x4)
27     x = checkpoint.checkpoint(self.up1, x5, x4)
28     x = checkpoint.checkpoint(self.up2, x, x3)
29     x = checkpoint.checkpoint(self.up3, x, x2)
30     x = checkpoint.checkpoint(self.up4, x, x1)
31     logits = self.outc(x)
32     return logits
33
34 def fuse_checkpointing(self):
35     # This method is no longer needed with the current implementation
36     # Keeping it in case you want to apply it later manually if required
37     pass
38

```

### 3.2 Explanation of Parameters in the Model

- **in\_channels**: This parameter represents the number of input channels. For a grayscale image, it would be 1, and for a color image, it would be 3 (RGB).
- **out\_channels**: This parameter defines the number of output channels. For binary segmentation, this would be 1, and for multi-class segmentation, it would be the number of classes.

In the `__init__` method of the UNet class, the encoder, middle, and decoder layers are defined using `nn.Conv2d` for convolution and `nn.MaxPool2d` for downsampling (pooling). The activations are ReLU, and the output of the final layer is a segmentation mask.

### 3.3 Explanation of Parameters in the Model

1. **Convolutional Layers (`nn.Conv2d`)**: These layers apply convolutions to the input. The `kernel_size=3` means a 3x3 filter is used, and `padding=1` ensures that the spatial dimensions of the input are preserved after convolution.
2. **Max Pooling Layer (`nn.MaxPool2d`)**: This reduces the spatial resolution of the feature maps, which helps the network focus on higher-level features.
3. **ReLU Activation (`nn.ReLU`)**: This introduces non-linearity into the model, enabling it to learn complex patterns.
4. **Output Layer**: The final convolutional layer produces the segmentation mask. The number of output channels corresponds to the number of classes to segment.

## 4 Training Script

This script is responsible for training the U-Net model on a set of images and corresponding masks. The code follows a step-by-step approach to initialize the model, configure the dataset, and handle the training process. Below is a detailed breakdown of the script.

This script trains a U-Net model for image segmentation, tracks the training process using WandB, and saves checkpoints periodically. It incorporates techniques like gradient clipping and learning rate scheduling to optimize training. The parameters can be customized via command-line arguments for flexibility in experimentation.

## 4.1 train\_model Function:

```

1  def train_model(
2      model,
3      device,
4      epochs: int = 5,
5      batch_size: int = 1,
6      learning_rate: float = 1e-5,
7      val_percent: float = 0.1,
8      save_checkpoint: bool = True,
9      img_scale: float = 0.5,
10     amp: bool = False,
11     weight_decay: float = 1e-8,
12     momentum: float = 0.999,
13     gradient_clipping: float = 1.0,
14 ):

```

- **model**: The U-Net model.
- **device**: The device on which the model will run (cpu or mps).
- **epochs**: Number of epochs to train the model.
- **batch\_size**: Number of samples in each batch.
- **learning\_rate**: Learning rate for the optimizer.
- **val\_percent**: The percentage of the dataset to use for validation.
- **save\_checkpoint**: Whether or not to save model checkpoints during training.
- **img\_scale**: Scale factor for resizing the images.
- **amp**: Whether or not to use mixed precision training (AMP).
- **weight\_decay**: Weight decay for regularization.
- **momentum**: Momentum for the optimizer.
- **gradient\_clipping**: Maximum gradient value for clipping.

## 4.2 Dataset Initialization and Splitting

```

1  try:
2      dataset = CarvanaDataset(dir_img, dir_mask, img_scale)
3  except (AssertionError, RuntimeError, IndexError):
4      dataset = BasicDataset(dir_img, dir_mask, img_scale)
5
6      # Use only 10% of the dataset
7      subset_size = int(len(dataset) * 0.1)
8      dataset, _ = random_split(
9          dataset,
10         [subset_size, len(dataset) - subset_size],
11         generator=torch.Generator().manual_seed(0),
12     )

```

- This tries to load the dataset using the CarvanaDataset class. If that fails, it falls back to BasicDataset.

```
1 subset_size = int(len(dataset) * 0.1)
2 dataset, _ = random_split(
3     dataset,
4     [subset_size, len(dataset) - subset_size],
5     generator=torch.Generator().manual_seed(0),
6 )
```

- A subset of 10% of the dataset is used to speed up the training process. This is useful for initial experimentation or when working with large datasets.

```
1 n_val = int(len(dataset) * val_percent)
2 n_train = len(dataset) - n_val
3 train_set, val_set = random_split(
4     dataset, [n_train, n_val], generator=torch.Generator().manual_seed(0)
5 )
6
```

- The dataset is split into training and validation sets. The validation set is determined by the val\_percent argument (10% by default).

### 4.3 Data Loaders

```
1 loader_args = dict(
2     batch_size=batch_size, num_workers=os.cpu_count(), pin_memory=False
3 ) # Pin memory set to False for CPU
4 train_loader = DataLoader(train_set, shuffle=True, **loader_args)
5 val_loader = DataLoader(val_set, shuffle=False, drop_last=True, **loader_args)
6
```

- The DataLoader is used to load the training and validation sets in batches. The shuffle=True argument ensures that the training data is shuffled during each epoch.

## 4.4 WandB Experiment Initialization

```
1 experiment = wandb.init(project="U-Net", resume="allow", anonymous="must")
2 experiment.config.update(
3     dict(
4         epochs=epochs,
5         batch_size=batch_size,
6         learning_rate=learning_rate,
7         val_percent=val_percent,
8         save_checkpoint=save_checkpoint,
9         img_scale=img_scale,
10        amp=amp,
11    )
12 )
```

- WandB is initialized to track the training process and log metrics (e.g., loss, validation score) throughout the training.

## 4.5 Optimizer, Loss, and Scheduler Setup

```
1 optimizer = optim.RMSprop(
2     model.parameters(),
3     lr=learning_rate,
4     weight_decay=weight_decay,
5     momentum=momentum,
6     foreach=True,
7 )
8 scheduler = optim.lr_scheduler.ReduceLROnPlateau(
9     optimizer, "max", patience=5
10 ) # goal: maximize Dice score
11 grad_scaler = None # AMP not supported on MacBook M3 (No CUDA)
12 criterion = nn.CrossEntropyLoss() if model.n_classes > 1 else nn.BCEWithLogitsLoss()
13 global_step = 0
14
```

- RMSprop is used as the optimizer, along with ReduceLROnPlateau scheduler to adjust the learning rate if the validation Dice score plateaus.

- CrossEntropyLoss or BCEWithLogitsLoss is chosen based on the number of classes in the model.

## 4.6 Training Loop

```
1  for epoch in range(1, epochs + 1):
2      model.train()
3      epoch_loss = 0
4      with tqdm(total=n_train, desc=f"Epoch {epoch}/{epochs}", unit="img") as pbar:
5          for batch in train_loader:
6              images, true_masks = batch["image"], batch["mask"]
7
```


- The training loop iterates over the dataset for the specified number of epochs.
- For each batch in the training set.
  - The images and masks are transferred to the selected device (CPU or MPS).
  - A forward pass is done on the model to predict the masks.
  - The loss is calculated using the chosen loss function, which is a combination of CrossEntropyLoss (or BCEWithLogitsLoss) and dice\_loss (for better segmentation accuracy).
  - The model's parameters are updated using backpropagation.

## 4.7 Checkpoint Saving

```
1  if save_checkpoint:
2      Path(dir_checkpoint).mkdir(parents=True, exist_ok=True)
3      state_dict = model.state_dict()
4      # state_dict["mask_values"] = dataset.mask_values
5      state_dict["mask_values"] = dataset.dataset.mask_values
6      torch.save(state_dict, str(dir_checkpoint / f"checkpoint_epoch{epoch}.pth"))
7      logging.info(f"Checkpoint {epoch} saved!")
8
```

- If the save\_checkpoint flag is True, the model's state dictionary (weights and other parameters) is saved to the checkpoint directory after each epoch.

## 4.8 Model Setup and Training Execution



```
1  model = UNet(n_channels=3, n_classes=args.classes, bilinear=args.bilinear)
2  model = model.to(memory_format=torch.channels_last)
3
4  train_model(
5      model=model,
6      epochs=args.epochs,
7      batch_size=args.batch_size,
8      learning_rate=args.lr,
9      device=device,
10     img_scale=args.scale,
11     val_percent=args.val / 100,
12     amp=args.amp,
13 )
14
```

- The U-Net model is initialized with the specified number of input channels, output classes, and upscaling method (bilinear or transposed convolution).
- The model is moved to the correct device (CPU or MPS) and training is initiated by calling `train_model()`.

## 5 Prediction Script

The prediction process involves the following steps:

1. Argument Parsing: The script parses command-line arguments to determine input/output filenames, model file, and other parameters.
2. Model Setup:
  - Creates a UNet instance with the specified number of input channels and classes.
  - Sets up the device (CPU or MPS if available).
3. Prediction Loop: For each input image:
  - Loads the image.
  - Calls the `predict_img` function to generate a mask
  - Optionally saves the result
  - Optionally visualizes the result
4. Image Processing:



- Converts the image to a tensor suitable for the model input
- Converts the image to a tensor suitable for the model input
- Runs inference on the model
- Post-processes the output (upsampling and thresholding)

## 5.1 Key Functions

### 5.1.1 predict\_img Function

```
1 def predict_img(net, full_img, device, scale_factor=1, out_threshold=0.5):
2     net.eval()
3     img = torch.from_numpy(
4         BasicDataset.preprocess(None, full_img, scale_factor, is_mask=False)
5     )
6     img = img.unsqueeze(0)
7     img = img.to(device=device, dtype=torch.float32)
8
9     with torch.no_grad():
10         output = net(img).cpu() # CPU for now, we'll handle MPS in the device setting
11         output = F.interpolate(
12             output, (full_img.size[1], full_img.size[0]), mode="bilinear"
13         )
14         if net.n_classes > 1:
15             mask = output.argmax(dim=1)
16         else:
17             mask = torch.sigmoid(output) > out_threshold
18
19     return mask[0].long().squeeze().numpy()
```

This function performs the actual prediction on a single image:

1. Prepares the input image tensor
2. Runs inference on the model
3. Upsamples the output to match the original image size
4. Applies thresholding to convert probabilities to binary masks

### 5.1.2 get\_args Function

This function sets up the command-line interface for user input, allowing users to specify:

- Input and output filenames
- Model file

- Visualization option
- Threshold for mask generation
- Scale factor for resizing
- Bilinear interpolation option
- Number of classes

```
1 def get_args():
2     parser = argparse.ArgumentParser(description="Predict masks from input images")
3     parser.add_argument(
```

### 5.1.3 mask\_to\_image Function

This function converts the predicted mask to an image format:

- Determines the color mapping based on the number of classes
- Creates a colored image from the mask

```
1 def mask_to_image(mask: np.ndarray, mask_values):
2     if isinstance(mask_values[0], list):
3         out = np.zeros(
4             (mask.shape[-2], mask.shape[-1], len(mask_values[0])), dtype=np.uint8
5         )
6     elif mask_values == [0, 1]:
7         out = np.zeros((mask.shape[-2], mask.shape[-1]), dtype=bool)
8     else:
9         out = np.zeros((mask.shape[-2], mask.shape[-1]), dtype=np.uint8)
10
11     if mask.ndim == 3:
12         mask = np.argmax(mask, axis=0)
13
14     for i, v in enumerate(mask_values):
15         out[mask == i] = v
16
17     return Image.fromarray(out)
```

## 5.2 Prediction Workflow

1. Load the trained model from the specified file.
2. For each input image:
  - Open the image file
  - Call `predict_img` to generate a mask
  - Optionally save the result
  - Optionally visualize the result
3. Repeat for all input images.

## 6 Model Execution Flow

1. **Data Loading:** The input image is loaded using a custom dataset class, which performs any necessary transformations (e.g., resizing, normalization).
2. **Forward Pass:**
  - The input image `x` passes through the encoder, where its dimensions are reduced, and features are extracted.
  - The features go through the bottleneck, which processes them with the highest abstraction.
  - Finally, the decoder upsamples the feature maps and reconstructs the segmentation mask.
3. **Output:** The model generates a segmentation mask with pixel-level predictions.

The PyTorch U-Net implementation provides a comprehensive solution for semantic segmentation tasks, with a focus on high-quality image processing. It offers flexibility through various configuration options, supports automatic mixed precision training, and provides tools for both training and prediction phases of the model lifecycle. The code is structured to be modular and easy to understand, with clear separation of concerns between data loading, model definition, training loop, and visualization components.

