# Lab 4: Semantic Segmentation final-report

CHI YEONG HEO[1],
[1]School of Computer Science and Engineering, Pusan National University, Busan 46241 Republic of Korea

## 1. Introduction

This report presents an analysis of the implementation of Fully Convolutional Networks for Semantic Segmentation[4], training on Pascal Visual Object Classes (VOC) dataset[1].

## 2. Methodology

### 2.1. Dataset

The PASCAL Visual Object Classes (VOC) dataset[1] is an extensively utilized benchmark in computer vision research, particularly in the domains of object detection, image classification, and segmentation. This dataset was initially introduced as part of the PASCAL VOC challenges, which were conducted annually from 2005 to 2012 to drive advancements in visual object recognition. It contains images annotated with bounding boxes and class labels across multiple object categories, providing a rich and diverse training resource for model development.

For this implementation, the training dataset comprises 10,589 images sourced from both the VOC 2007 to 2011 datasets. The dataset encompasses 21 object classes, categorized as follows:

- Background: background
- Person: person
- Animal: bird, cat, cow, dog, horse, sheep
- Vehicle: aeroplane, bicycle, boat, bus, car, motorbike, train
- Indoor: bottle, chair, dining table, potted plant, sofa, tv/monitor

### 2.2. Data Augmentation

Data augmentation is a technique employed to increase the effective size and variability of a training dataset by applying random transformations to the input data. This approach enhances the model's generalization capabilities by exposing it to a broader range of representations of the original images, which helps mitigate overfitting and improves performance on unseen data.

For this implementation, the original image dimensions (1024 $\times$ 1024 pixels) are maintained without resizing. Pixel values are normalized to the range [0, 1] prior to applying data augmentation.

#### 2.2.1. Random Scaling

For this implementation, the images are first resized to 321 $\times$ 321, and then scaled by a random factor (between 0.5 and 1.3). This scaling variability enhances the model's robustness to changes in object size.

#### 2.2.2. Random Horizontal Flipping

Random Horizontal Flipping is a technique that horizontally mirrors the images with a predefined probability (0.5 in this implementation). By introducing variations in the orientation of objects, this method helps the model become less biased toward any specific orientation. As a result, the model is encouraged to learn more generalized and abstract representations of features, improving its overall performance on unseen images.

### 2.3. Normalization

Normalization is a preprocessing technique that adjusts the pixel values of the images to have a mean of zero, based on the mean computed from the training dataset. This statistic are then applied consistently to both the training and test datasets to prevent data leakage. For this implementation, channel-wise means of (104.008, 116.669, 122.675) is used.

The normalization process offers two key advantages:

(1) *Faster Convergence*: By ensuring the input data lies within a standardized range, normalization aids in achieving faster and more stable convergence during the model training phase.

(2) *Improved Generalization*: Standardizing the input distribution reduces the model's sensitivity to variations in pixel intensity, allowing it to focus on more critical and meaningful patterns within the data, thereby improving its ability to generalize beyond the training set.

### 2.4. Adam Optimizer

The Adam (Adaptive Moment Estimation) [3] optimizer is a widely-used algorithm in machine learning, particularly for deep learning applications, that improves upon Stochastic Gradient Descent (SGD) by combining ideas from momentum and adaptive learning rates. Adam aims to minimize a given loss function by iteratively updating model parameters. It uses the first moment (mean) and second moment (variance) of past gradients to stabilize and accelerate convergence, making it especially suited for complex optimization landscapes.

Adam computes an individual learning rate for each parameter based on the gradients' mean and variance, making it effective at dealing with noisy and sparse gradients. Adam also includes bias correction for the moment estimates, which is critical in early iterations. The PyTorch library's Adam implementation allows further customization with parameters like weight decay and an optional AMSGrad variant to ensure convergence.

**Algorithm 1:** Adam Optimizer with AMSGrad Variant

**Input:** $\gamma$ (learning rate), $\beta_1, \beta_2$ (moment coefficients), $\theta_0$ (initial parameters), $f(\theta)$ (objective function), $\lambda$ (weight decay), *amsgrad* flag, *maximize* flag

**Initialize:** $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\hat{v}_0 \leftarrow 0$ (max second moment for AMSGrad)

**for** $t = 1$ *to* ... **do**

  **if** *maximize* **then**

    $g_t \leftarrow -\nabla_\theta f(\theta_{t-1})$

  **else**

    $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$

  **if** $\lambda \neq 0$ **then**

    $g_t \leftarrow g_t + \lambda \theta_{t-1}$

  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$ // Update biased first moment

  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ // Update biased second moment

  $\hat{m}_t \leftarrow m_t / (1-\beta_1^t)$

  $\hat{v}_t \leftarrow v_t / (1-\beta_2^t)$

  **if** *amsgrad* **then**

    $\hat{v}_t^{\max} \leftarrow \max(\hat{v}_{t-1}^{\max}, \hat{v}_t)$

    $\theta_t \leftarrow \theta_{t-1} - \gamma \hat{m}_t / (\sqrt{\hat{v}_t^{\max}} + \varepsilon)$

  **else**

    $\theta_t \leftarrow \theta_{t-1} - \gamma \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$

**Return** $\theta_t$

### 2.5. Batch Normalization

Batch Normalization was introduced in [2] to address the challenge of internal covariate shift in the training of Deep Neural Networks. Internal covariate shift refers to the variation in the distribution of each layer's inputs caused by updates to the parameters of preceding layers during training. This phenomenon complicates the training process by necessitating reduced learning rates and meticulous parameter initialization. Additionally, it poses difficulties in training models that employ saturating nonlinearities.

Batch Normalization alleviates the internal covariate shift by standardizing the inputs to each layer. Specifically, it normalizes each individual feature based on its mean and variance calculated over the mini-batch. Subsequently, a linear transformation using scaling and shifting parameters, denoted as $\gamma$ and $\beta$, respectively, is applied. This ensures the capability of the transformation to represent the identity function. Should the original input (prior to normalization) be optimal, these parameters are expected to approximate the identity function.

The Batch Normalization offers two key advantages:

(1) *Enabling Higher Learning Rates*: By normalizing input distributions throughout the network, it prevents small changes to the parameters from amplifying into larger and suboptimal changes in activations in gradients. Batch Normalization also makes training more resilient to the parameter scale.

(2) *Regularizing the model*: A training example is seen in conjunction with other examples as a form of statistics, and the training network no longer producing deterministic values for a given training example.

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1, \ldots, x_m\}$;
Parameters to be learned: $\gamma, \beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \varepsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

**Algorithm 2:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.

### 2.6. Rectified Linear Unit (ReLU)

The Rectified Linear Unit (ReLU) is an extensively adopted activation function within the domain of neural networks, particularly noted for its simplicity and efficacy in deep learning models. Formally, ReLU is defined by the piecewise linear function $f(x) = \max(0, x)$, where $x$ denotes the input to a given neuron. Consequently, the ReLU function yields an output of zero for any negative input, while preserving the input value for non-negative entries as a linear identity.

### 2.7. Dropout

Dropout[5] is a regularization technique used in neural networks to prevent overfitting by randomly "dropping out" a subset of neurons during each training iteration. Dropout addresses the tendency of large, complex networks to overfit by forcing the network to learn redundant representations

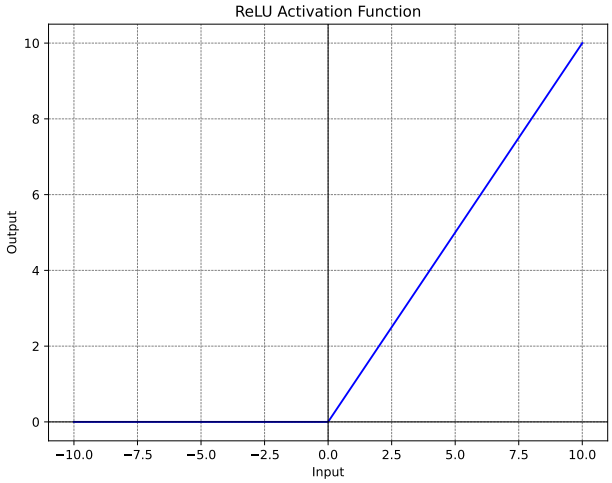


Figure 1: Rectified Linear Unit

of data. This is achieved by randomly setting a fraction $p$ (dropout rate) of neuron activations to zero during each forward pass in the training phase. As a result, neurons are less dependent on any one feature or other neurons, promoting robustness and improving generalization.

During training, dropout is applied to each layer independently, temporarily removing randomly selected neurons and their connections. This random omission of neurons creates an ensemble of slightly different networks with shared weights, which effectively reduces the likelihood of overfitting by discouraging the network from relying too heavily on any particular feature or pathway.

At test time, dropout is turned off, allowing all neurons to participate in the final output. The network's weights are scaled down by the dropout rate to account for the reduced activations used during training, making the predictions consistent with those seen in the training phase.

**Algorithm 2:** Dropout Forward Pass

**Input:** $x$ (input layer activations), $p$ (dropout rate)
**Generate:** $r \sim \mathrm{Bernoulli}(1-p)$ // Random binary mask based on $p$
**Apply Mask:** $\tilde{x} = r \cdot x$ // Drop random activations by element-wise multiplication
**Scale:** $\tilde{x} = \tilde{x}/(1-p)$ // Scale remaining activations
**Return:** $\tilde{x}$

## 3. Implementation Details

### 3.1. FCN-8s Model Architecture

The FCN8s (Fully Convolutional Network) architecture is designed for semantic segmentation tasks, where the goal is to classify each pixel in an image into one of the predefined classes.

#### 3.1.1. Backbone: VGG16 Features

The model utilizes the feature extraction layers from a pre-trained VGG16 network, up to a certain depth. The `self.features` attribute is initialized with `models.vgg16(pretrained=True).features`, retaining the convolutional layers of VGG16. This part acts as the encoder of the network, capturing spatial hierarchies from the input images. The original convolutional configurations of VGG-16 are presented in Table 1. For this implementation, configuration "D" is employed, as it is the default setting of the 'vgg16' module in the torchvision library.

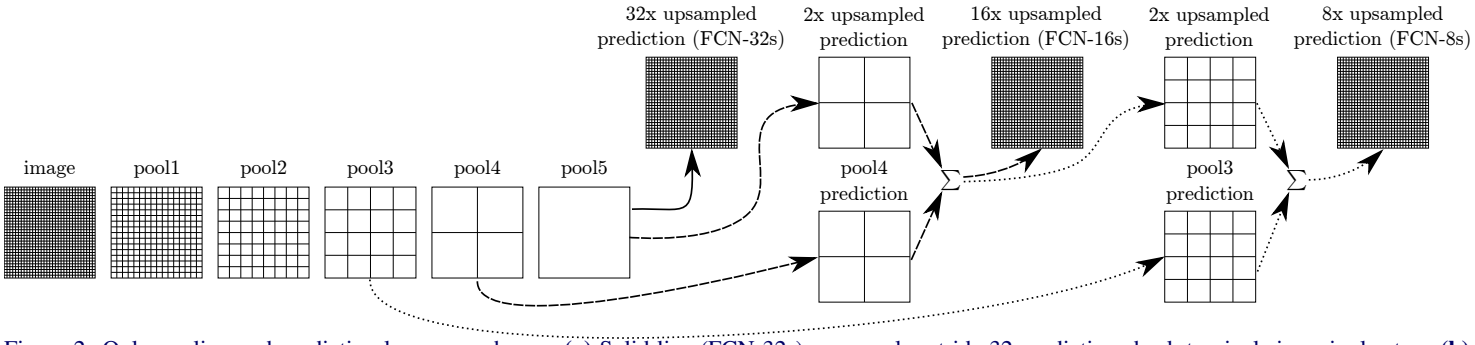


Figure 2: Only pooling and prediction layers are shown. **(a)** Solid line (FCN-32s): upsamples stride 32 predictions back to pixels in a single step. **(b)** Dashed line (FCN-16s): Combining predictions from both the final layer and the `pool4` layer, at stride 16, lets the net predict finer details, while retaining high-level semantic information. **(c)** Dotted line (FCN-8s): Additional predictions from `pool3`, at stride 8, provide further precision.

Since the edges of feature maps are not processed well, they are removed during forward pass. To implement this, the first convolutional layer has a padding (100, 100), enlarging the size of intermediate feature maps. Later in skip connection step, the feature maps are cropped and the border are discarded.

#### 3.1.2. Fully Connected Layers (as Convolutional Layers)

The architecture replaces the fully connected layers from VGG16 with convolutional layers to make the network fully convolutional. The **fc6** layer is a convolutional layer with 4096 output channels and a kernel size of 7, followed by a ReLU activation and dropout. Similarly, **fc7** is another convolutional layer with 4096 output channels and a kernel size of 1, followed by a ReLU activation and dropout. These layers function similarly to fully connected layers by using large kernel sizes and are crucial for dense labeling.

#### 3.1.3. Prediction Layers

The architecture involves three prediction layers that correspond to different scales in the feature hierarchy. **Predict1** converts the deep features from the `fc7` layer to class scores using a $1 \times 1$ convolution. **Predict2** converts the spatial features from an earlier stage (layer with index 23) using a $1 \times 1$ convolution. **Predict3** operates on an even earlier feature map (layer with index 16) to produce class scores. These layers transform intermediate feature maps into score maps for each class.

#### 3.1.4. Upsampling through Deconvolutional Layers

To restore the output to its original spatial dimensions, the model utilizes a series of transposed convolutional layers. The first transposed convolutional layer, **Deconv1**, enhances the spatial resolution of the score map derived from `predict1` by a factor of two. Subsequently, **Deconv2** further refines the spatial resolution by upsampling the merged score map from both `predict1` and `predict2`. The final layer, **Deconv3**, executes an additional upsampling by a factor of eight. During these operations, larger score maps are appropriately cropped to align their dimensions with smaller score maps, enabling the addition operation. Due to the dimensional disparity between the output of **Deconv3** and the input image, the score map is also center-cropped to match the input size. Upon completion of this process, the model is capable of producing segmentation logit maps that conform to the original input dimensions.

#### 3.1.5. Skip Connections and Fusion

The model amalgamates detailed information from multiple layers (`pr1`, `pr2`, `pr3`) through the implementation of skip connections. This hierarchical fusion is instrumental in capturing both low-level and high-level semantic information. The incorporation of skip connections allows for a weighted integration, wherein predictions from earlier layers are added to the upsampled, coarser representations using small scaling factors. This approach enhances the stability and accuracy of the segmentation process.

### 3.2. Cross-entropy loss

Cross-entropy loss is a widely employed objective function in classification tasks, particularly in the training of models for tasks such as semantic segmentation. It is utilized to measure the dissimilarity between the predicted class probabilities and the true class distribution, thus guiding the optimization of model parameters during training.

For a given input image, the Fully Convolutional Network 8s (FCN8s) generates a score map of dimensions $(C, W, H)$, where $C$ denotes the number of classes, and $W$ and $H$ are the spatial dimensions of the output. The predicted probability of each pixel belonging to a class $c$ is obtained by applying the softmax function across the class scores:

$$p_{i,j,c} = \frac{\exp(s_{i,j,c})}{\sum_{c'=1}^{C} \exp(s_{i,j,c'})}$$

Here, $s_{i,j,c}$ represents the score for class $c$ at pixel $(i, j)$, and $p_{i,j,c}$ signifies the probability that the pixel $(i, j)$ belongs to class $c$.

The cross-entropy loss for a single image with ground truth labels $y_{i,j}$ is defined as:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{c=1}^{C} \mathbf{1}(y_{i,j} = c) \cdot \log(p_{i,j,c})$$

where $N$ is the total number of pixels, and $\mathbf{1}(y_{i,j} = c)$ is an indicator function that equals 1 if the true class label for pixel $(i, j)$ is $c$, and 0 otherwise.

The optimization of the model parameters is carried out by minimizing the cross-entropy loss, thereby aligning the predicted class probabilities with the true class distributions. T

## 4. Results

### 4.1. Inference Visualization

To evaluate the qualitative performance of our trained model, I conducted a visual analysis of detection results on a subset of test images (Fig. 3. The model demonstrates its semantic segmentation capabilities by generating class-specific segmentation masks.

### 4.2. Test Set Performance

The model's quantitative performance was evaluated using the mean Intersection over Union (mIoU) metric, calculated for 1,449 test images.

The mIoU across these images is 0.4225, which reflects the model's overall semantic segmentation accuracy.

## 5. Conclusion

This report presented a detailed implementation and analysis of the Fully Convolutional Networks framework, focusing specifically on the FCN8s architecture. Through leveraging a pre-trained VGG16 model, FCN8s efficiently extracts hierarchical features from input images, which are crucial for effective semantic segmentation tasks. The transition from fully connected layers to convolutional ones, denoted as `fc6` and `fc7`, allows the model to be fully convolutional, thereby facilitating dense pixel predictions across images of varying sizes.


[1] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, January 2015.

[2] Sergey Ioffe. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[3] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[4] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation, 2015. URL https://arxiv.org/abs/1505.04366.

[5] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.


Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

| ConvNet Configuration | | | | | |
|---|---|---|---|---|---|
| A | A-LRN | B | C | D | E |
| 11 weight layers | 11 weight layers | 13 weight layers | 16 weight layers | 16 weight layers | 19 weight layers |
| input ($224 \times 224$ RGB image) | | | | | |
| conv3-64 | conv3-64<br>**LRN** | conv3-64<br>**conv3-64** | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 |
| maxpool | | | | | |
| conv3-128 | conv3-128 | conv3-128<br>**conv3-128** | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 |
| maxpool | | | | | |
| conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256<br>**conv1-256** | conv3-256<br>conv3-256<br>**conv3-256** | conv3-256<br>conv3-256<br>conv3-256<br>**conv3-256** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| FC-4096 | | | | | |
| FC-4096 | | | | | |
| FC-1000 | | | | | |
| soft-max | | | | | |

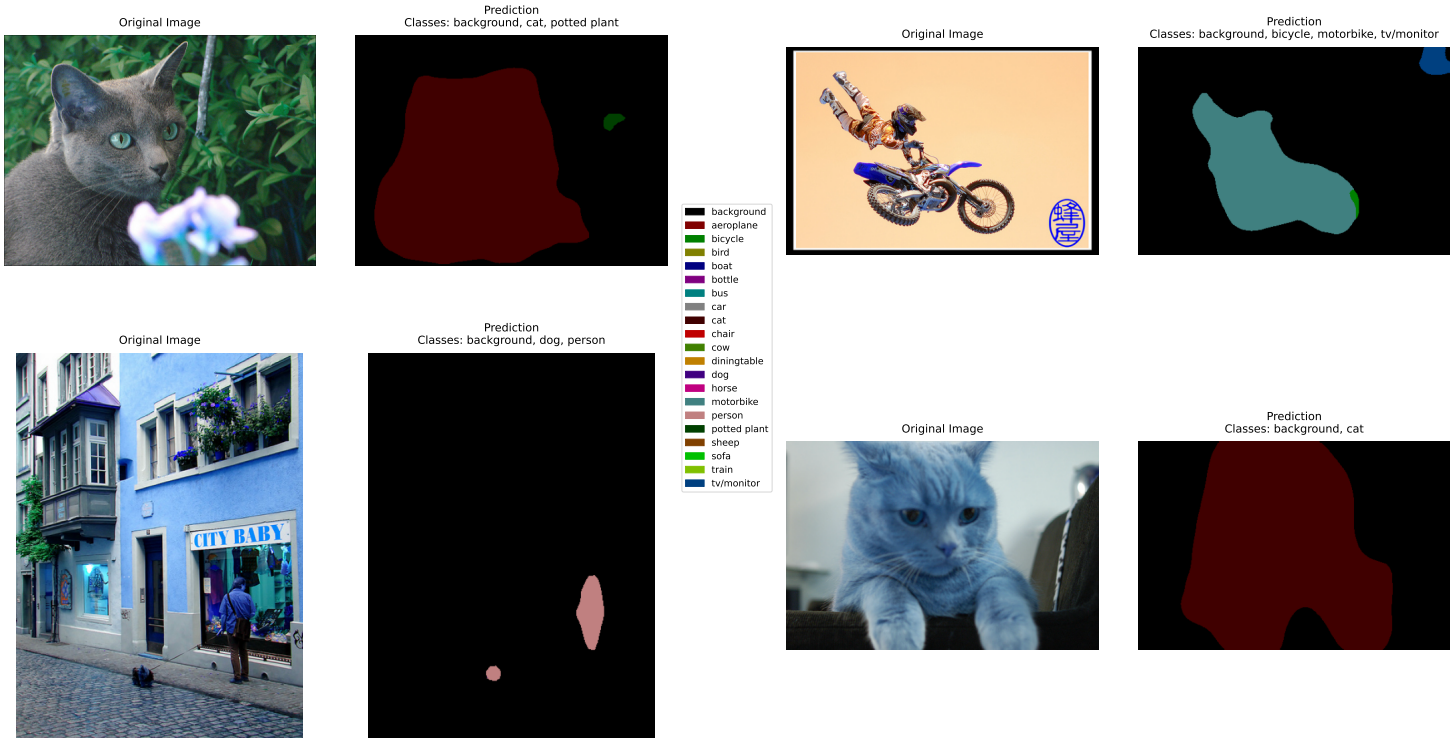


Figure 3: Model inference results on four test images. Each original image (left) is paired with its corresponding predicted segmentation map (right). The model accurately predicts class labels; however, further refinement is required to achieve sharper segmentation boundaries.