AI Hiragana Detector

Group 1:

Group Members: Matthew Phillips, Demitrius Webb, Gage Leinenger

Abstract

For our project, we wanted to tackle the problem of how an AI agent can detect specific Japanese hiragana and predict with high accuracy which character is being displayed to it. We hypothesized that by implementing an AI agent with both OpenCV for detection capabilities as well as a CNN for information processing and prediction that our AI agent could succeed in predicting Japanese hiragana with high efficiency. This report summarizes our project process and results in developing that AI Agent which uses sensors to detect, analyze, and predict specific Japanese hiragana that are present.

Key findings include:

- Our model can predict each character with at least 49% prediction confidence
- Our model on average predicts any given character with 92.2% prediction confidence
- Our model predicts over 50% of the dataset with 95% or better confidence
- Our model predicts over 78% of the dataset with 90% or better confidence

Table of Contents

1 Introduction

The basis of our project is rooted in the ability for an AI agent to both accurately detect images presented to it, as well as make accurate predictions given inputs through sensors. This report will examine our methodology, dataset, project scope, and findings of our project and how our agent performed these tasks.

1.1 **Methodology**

For our AI agent to be able to accomplish our goals, we realized it needed to have 2 specific capabilities. First, our AI agent needs to be able to receive information, specifically images, from its environment. Secondly, our AI agent needs a way to process the information it is receiving and make a prediction based on some training dataset.

For the first capability, we found through research that OpenCV implementation in Python would be our best option. OpenCV, also known as Open-Source Computer Vision Library, is a powerful open-source library in Python used for computer vision and image processing tasks. It provides tools to read, write, and manipulate images and videos, perform operations like object detection, face recognition, edge detection, and feature extraction. We used these features to capture a still frame from a video and format it properly to be recognizable and readable by our AI agent.

For the second capability, we found through research that using Pytorch implementations would be our best option. Pytorch allows us to set up a simple CNN to detect features in a 28x28 MNIST style image. MNIST style of image has the character in the foreground and a black background. This style image makes a simple CNN more than enough to handle the detection of handwritten characters if the input is like the training data. Implementation will be discussed in the Methods section.

1.2 **Dataset**

The dataset that our AI agent has access to is a custom-generated dataset based on 12 different font packages. The dataset applies different filters and modifiers to the image to apply variation to the images. Specifically, we apply slight pixel randomization, vertical and horizontal translation, and gaussian blur. These three modifiers allow each training image in the dataset to be slight variations of the initial image while still maintaining defining characteristics for our model to train from. The dataset consists of 6000 images per individual character, with a 80/20 ratio of training data and test data. This means our total dataset consists of 276,000 images of hiragana characters.

1.3 **Project Scope**

For this project, our AI agent should be trained on a custom dataset of 12 font packages and be able to build a strong neural network to base its predictions on. Then, it will be exposed to individual hiragana characters and make predictions based on its training as to which hiragana is being displayed.

1.4 **Limitations**

For our AI agent, we had originally planned to look at the Kuzushiji-Kanji dataset and have it analyze Kanji characters. However, we discovered two flaws with this original plan. First, we quickly discovered that our lack of knowledge of different kanji as well as the overwhelming. Secondly, the number of kanji characters within the dataset would prove challenging with our limited time and computational resources. The dataset consisted of 3,832 individual kanji characters and over 140,000 images. Though the total number of images was less than the Kuzushiji-49 dataset, we found that many of the 3832 kanji had only 1 image of training data available. For these reasons we decided that to create the most efficient and correct AI agent with our time, we would try to use the Kuzushiji-49 dataset and focus our efforts on hiragana specifically.

However, this led us to a new complication when further exploring the Kuzushiji-49 dataset. A closer analysis of the dataset revealed that the training images were based on an older style of Japanese writing, a sort of cursive, that was popular before the Meiji restoration of 1868. The images to train from in this dataset did not in any way represent the style of writing we wanted our model to handle and recognize, leading to early iterations having extremely low prediction certainty. Because of this, we shifted to our final dataset and the method we used to generate a dataset for our model.

The final limitation we found was with the google translate functionality of the agent. The agent struggles to decipher specific beginnings and ends of characters, which leads to a misconception of what characters are actually being presented to the agent for translation. For example, if there was a word with 2 individual hiragana characters, but the first character was separated into two major contours, the agent would sometimes decipher the combination as 3 distinct hiragana. This led to the agent making predictions on segments of hiragana and registering them in the translator as its highest prediction for that segment. Though not always an issue, it proved challenging to deal with when specific words were inputted for testing.

2 Methods

2.1 PEAS Overview

Our agents design revolves around these PEAS items:

Performance:

An accuracy measure with the confidence of predicting a written hiragana character.

Environment:

Standard paper and markers to keep things in the environment simplified for detection.

Actuators:

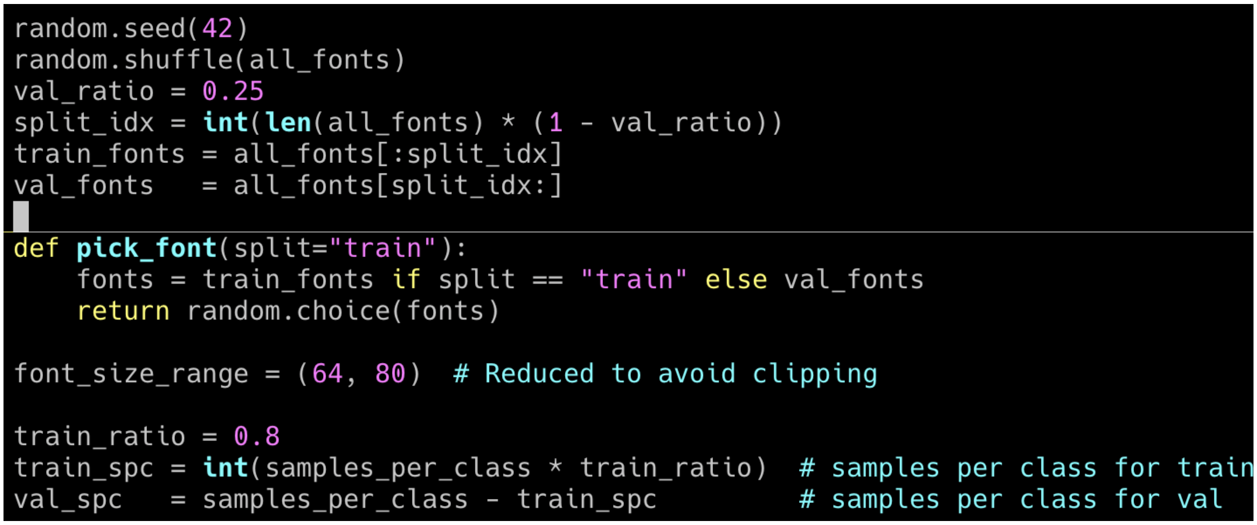
This includes CNN itself for prediction along with transformers to alter the images in a way that makes them look like handwriting. Google translate falls under this category to handle translations.

Sensors:

Any standard camera is to be used.

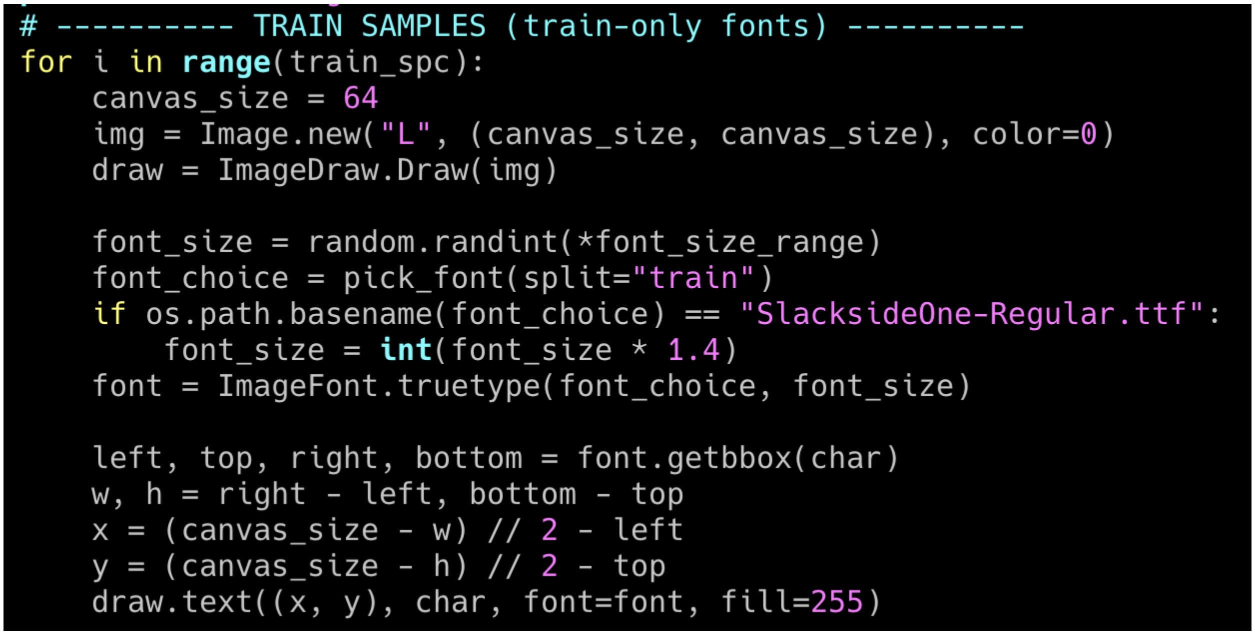
2.2 Generating the Dataset

The Kuzushiji datasets are images taken of old scrolls. The characters are written up and down and in cursive, and this wasn't acceptable for how the characters are written by us and other students. We took 12 different font styles and generated images based on those styles. We also distorted the images to add some variations to the images, to match handwritten hiragana characters. Fonts we used are open for personal use and are the following styles: ipagothic, KleeOne-Regular, NotoSansJP-Regular, SlacksideOne-Regular, TekitouPoem, ZenKurenaido-Regular, Kaorigel-Z9YK, Yomogi-Regular, HinaMincho-Regular, GL-CurulMinamoto, AoyagiSosekiFont2, and KouzanMouhituFontOTF. Links to the fonts are located in the references section. First, we shuffle the fonts to then break them into which ones to use for training data and which to use for testing data. Once a font is selected, the size of the font is then selected at random from 64 to 80 to then draw onto the image. The image starts with a black background and then the character gets centered and drawn on top in white.



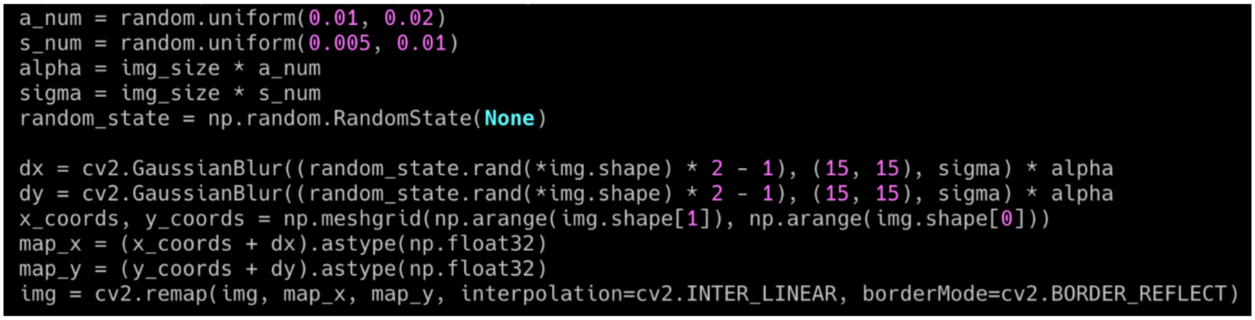
```
random.seed(42)
random.shuffle(all_fonts)
val_ratio = 0.25
split_idx = int(len(all_fonts) * (1 - val_ratio))
train_fonts = all_fonts[:split_idx]
val_fonts   = all_fonts[split_idx:]

def pick_font(split="train"):
    fonts = train_fonts if split == "train" else val_fonts
    return random.choice(fonts)

font_size_range = (64, 80)  # Reduced to avoid clipping

train_ratio = 0.8
train_spc = int(samples_per_class * train_ratio)  # samples per class for train
val_spc   = samples_per_class - train_spc         # samples per class for val
```

The above shows the randomization of the font selection and below shows how the character is centered and drawn onto the image initially.



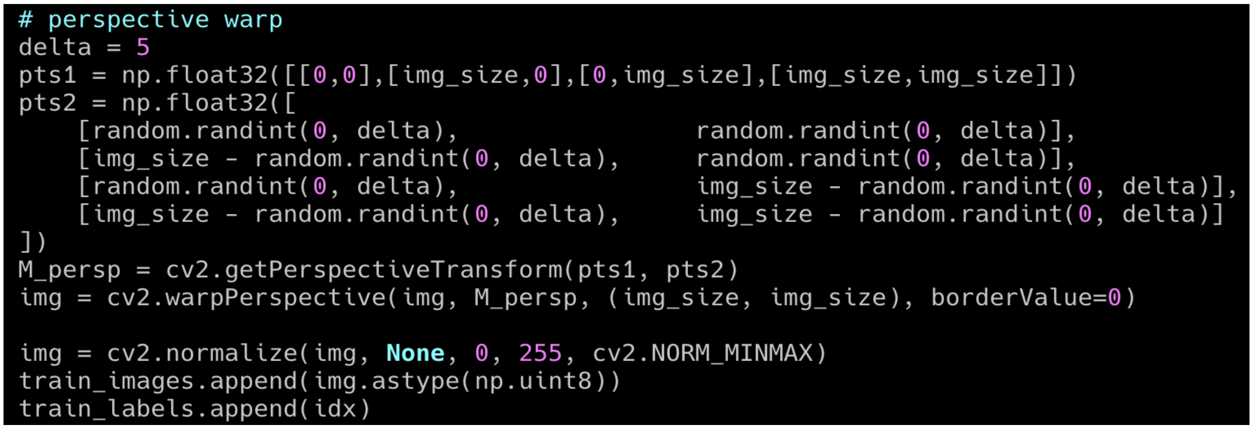
```
# ---------- TRAIN SAMPLES (train-only fonts) ----------
for i in range(train_spc):
    canvas_size = 64
    img = Image.new("L", (canvas_size, canvas_size), color=0)
    draw = ImageDraw.Draw(img)

    font_size = random.randint(*font_size_range)
    font_choice = pick_font(split="train")
    if os.path.basename(font_choice) == "SlacksideOne-Regular.ttf":
        font_size = int(font_size * 1.4)
    font = ImageFont.truetype(font_choice, font_size)

    left, top, right, bottom = font.getbbox(char)
    w, h = right - left, bottom - top
    x = (canvas_size - w) // 2 - left
    y = (canvas_size - h) // 2 - top
    draw.text((x, y), char, font=font, fill=255)
```

After this we apply some blur like so:



```
a_num = random.uniform(0.01, 0.02)
s_num = random.uniform(0.005, 0.01)
alpha = img_size * a_num
sigma = img_size * s_num
random_state = np.random.RandomState(None)

dx = cv2.GaussianBlur((random_state.rand(*img.shape) * 2 - 1), (15, 15), sigma) * alpha
dy = cv2.GaussianBlur((random_state.rand(*img.shape) * 2 - 1), (15, 15), sigma) * alpha
x_coords, y_coords = np.meshgrid(np.arange(img.shape[1]), np.arange(img.shape[0]))
map_x = (x_coords + dx).astype(np.float32)
map_y = (y_coords + dy).astype(np.float32)
img = cv2.remap(img, map_x, map_y, interpolation=cv2.INTER_LINEAR, borderMode=cv2.BORDER_REFLECT)
```

Warp the image and normalize the pixel values:

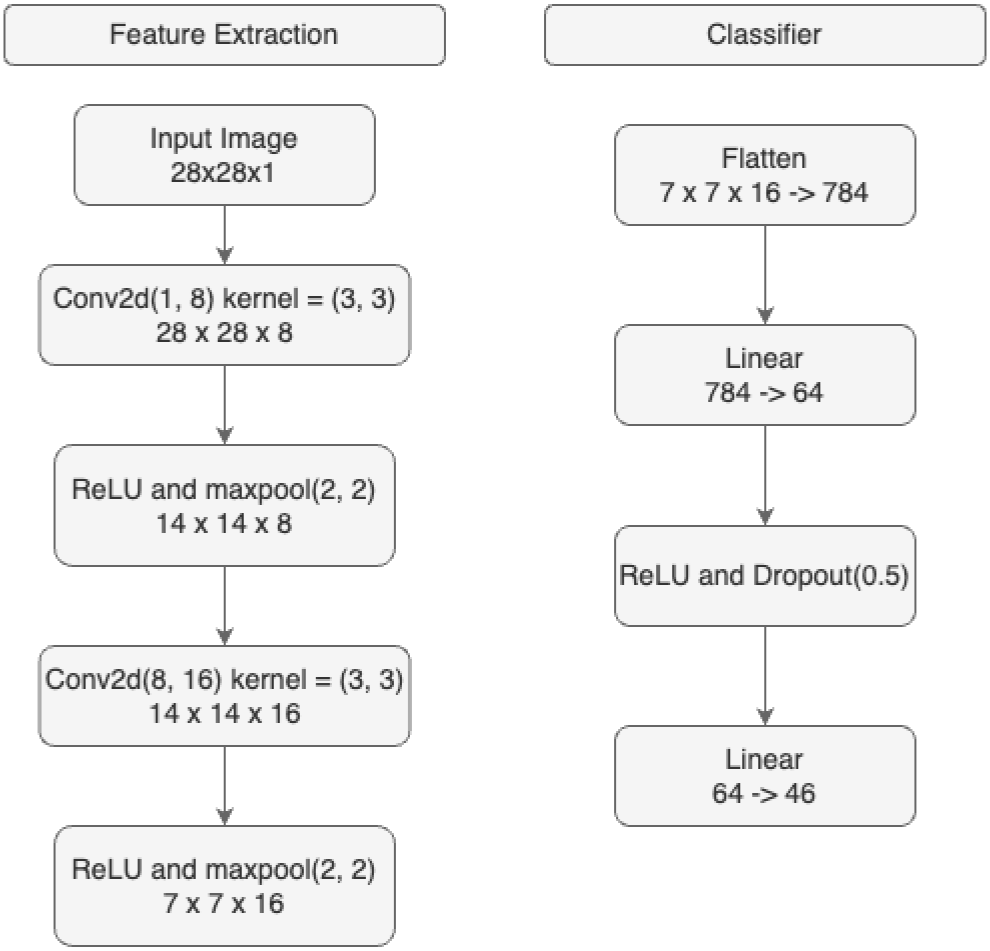


```
# perspective warp
delta = 5
pts1 = np.float32([[0,0],[img_size,0],[0,img_size],[img_size,img_size]])
pts2 = np.float32([
    [random.randint(0, delta),                  random.randint(0, delta)],
    [img_size - random.randint(0, delta),       random.randint(0, delta)],
    [random.randint(0, delta),                  img_size - random.randint(0, delta)],
    [img_size - random.randint(0, delta),       img_size - random.randint(0, delta)]
])
M_persp = cv2.getPerspectiveTransform(pts1, pts2)
img = cv2.warpPerspective(img, M_persp, (img_size, img_size), borderValue=0)

img = cv2.normalize(img, None, 0, 255, cv2.NORM_MINMAX)
train_images.append(img.astype(np.uint8))
train_labels.append(idx)
```

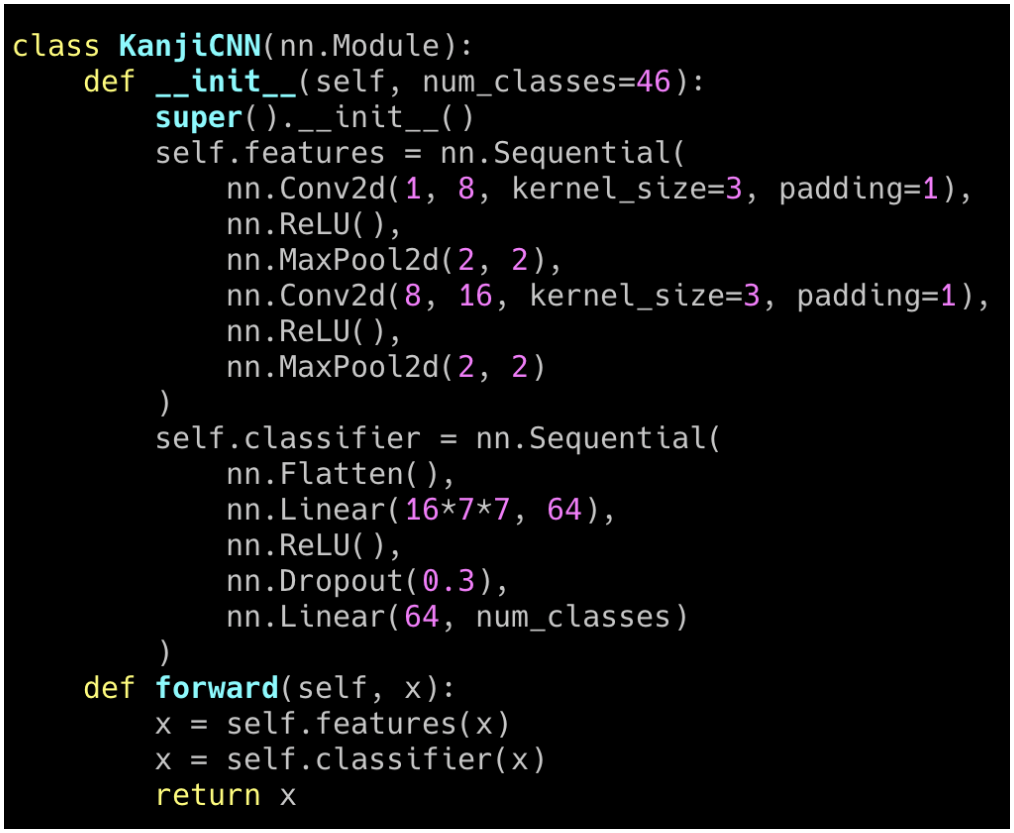
The same is done for the testing dataset. Once all the images are generated, a shuffle is applied to both datasets so that the model cannot just memorize the characters in order. The dataset is then saved to a .npz file to then load it at a later time.

2.3 Training the Convolutional Neural Network (CNN)

Training the CNN involves using the Pytorch library, which has built-in functions to aid with the design of a CNN. CNN takes in a 28x28x1 image and then applies some convolutional and maxpool layers to get to a size of 7x7x16. After this is done, we then run the classifier that flattens the data from a 3d vector to a 1d vector of size 784. This is then passed to a fully connected dense layer which goes to 64 nodes. The training then applies the ReLU propagation function and a dropout rate of 50% to keep the model from overfitting. Then a reduction of the number of classes for nodes that can then be accessed to get the prediction of the models. The general design on the CNN is in the image below:

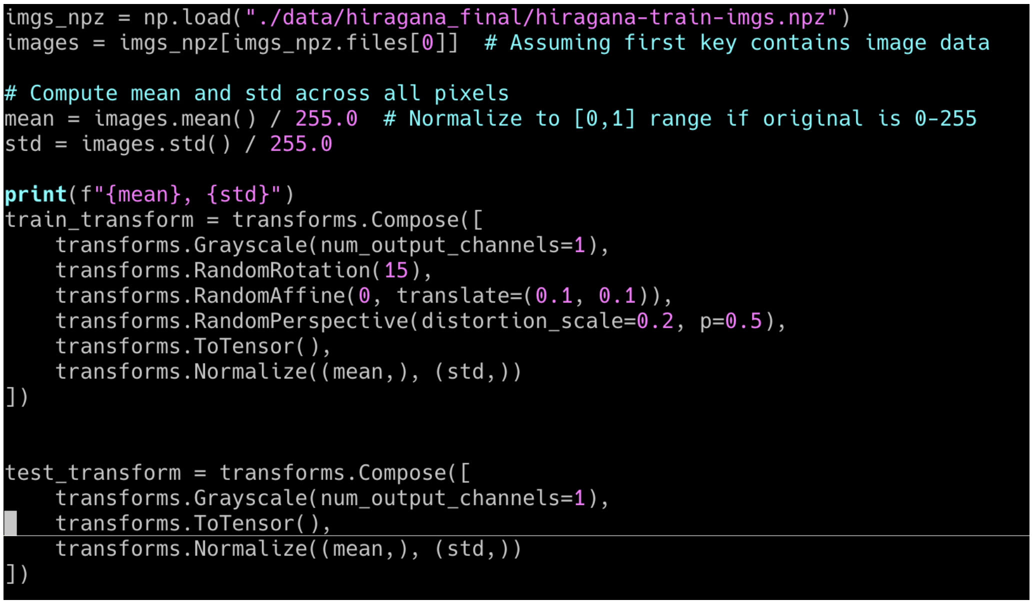


The code:



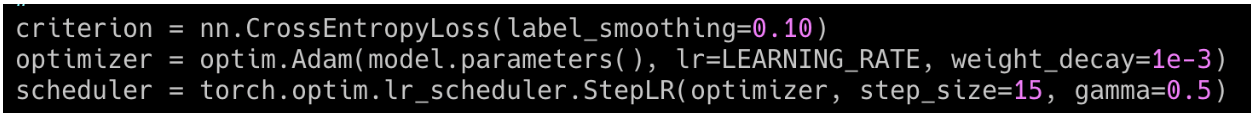
```
class KanjiCNN(nn.Module):
    def __init__(self, num_classes=46):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(8, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(16*7*7, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, num_classes)
        )
    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

The dataset has some transformations applied to the datasets to ensure more variety. That is done with these two lists:



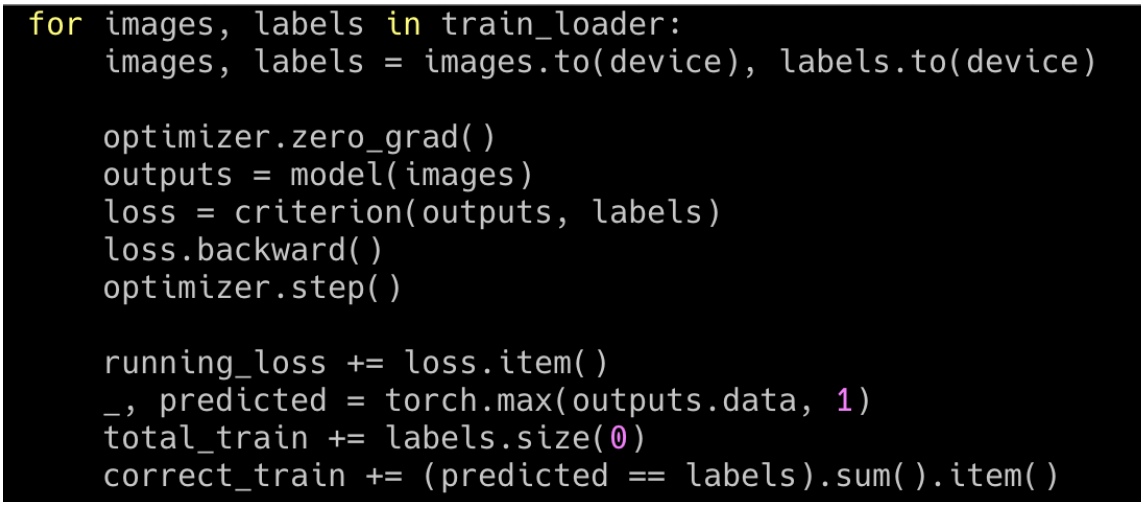
```
imgs_npz = np.load("./data/hiragana_final/hiragana-train-imgs.npz")
images = imgs_npz[imgs_npz.files[0]]  # Assuming first key contains image data

# Compute mean and std across all pixels
mean = images.mean() / 255.0  # Normalize to [0,1] range if original is 0-255
std = images.std() / 255.0

print(f"{mean}, {std}")
train_transform = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.RandomRotation(15),
    transforms.RandomAffine(0, translate=(0.1, 0.1)),
    transforms.RandomPerspective(distortion_scale=0.2, p=0.5),
    transforms.ToTensor(),
    transforms.Normalize((mean,), (std,))
])


test_transform = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
    transforms.Normalize((mean,), (std,))
])
```

The loss and optimizers are set up with these lines in the code. This ensures that we have proper label smoothing and prevents any overfitting from happening to the model.



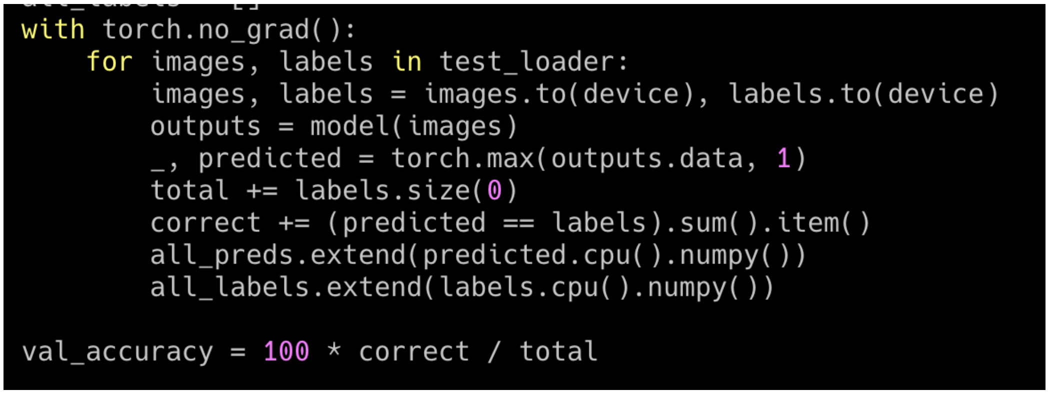
```
criterion = nn.CrossEntropyLoss(label_smoothing=0.10)
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE, weight_decay=1e-3)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=15, gamma=0.5)
```

The model is trained in only 15 epochs. This is because the dataset itself is pretty robust with 6000 images being generated per character. Training is handled in the following loop. This loop ensures that the proper steps are being applied to train the model correctly.



```
for images, labels in train_loader:
    images, labels = images.to(device), labels.to(device)

    optimizer.zero_grad()
    outputs = model(images)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    running_loss += loss.item()
    _, predicted = torch.max(outputs.data, 1)
    total_train += labels.size(0)
    correct_train += (predicted == labels).sum().item()
```

Validation accuracy is tested here. Mind that the test and training datasets have different font styles.

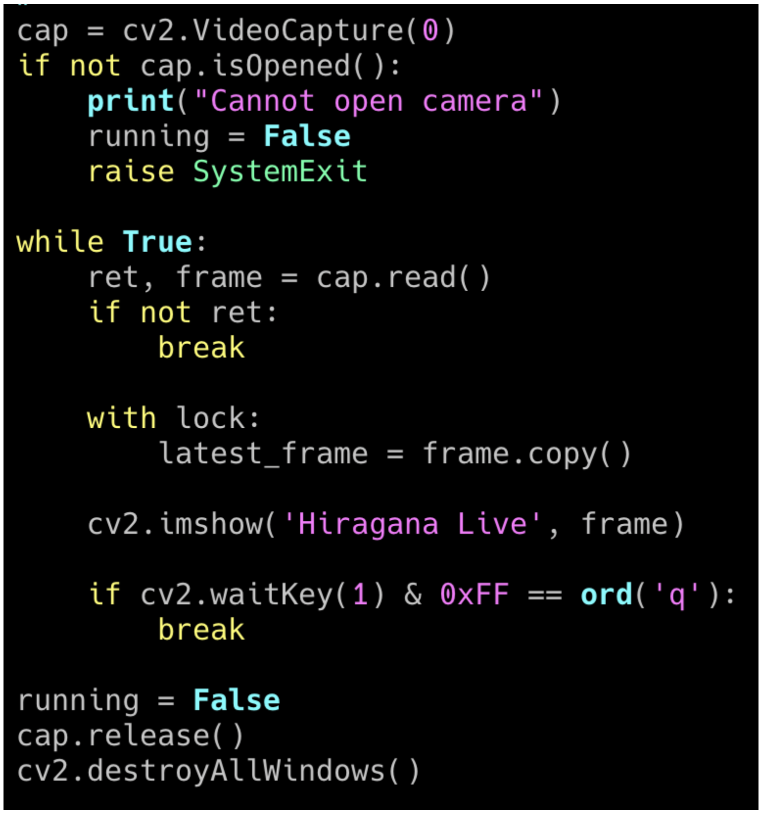


```
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

val_accuracy = 100 * correct / total
```

The model is then saved in a Pytorch style file to then be loaded for any fine-tuning, or any actual real-world use we would like to test the model in. The next step will be to use the model in a setting for testing how confident it is at identifying handwritten characters.

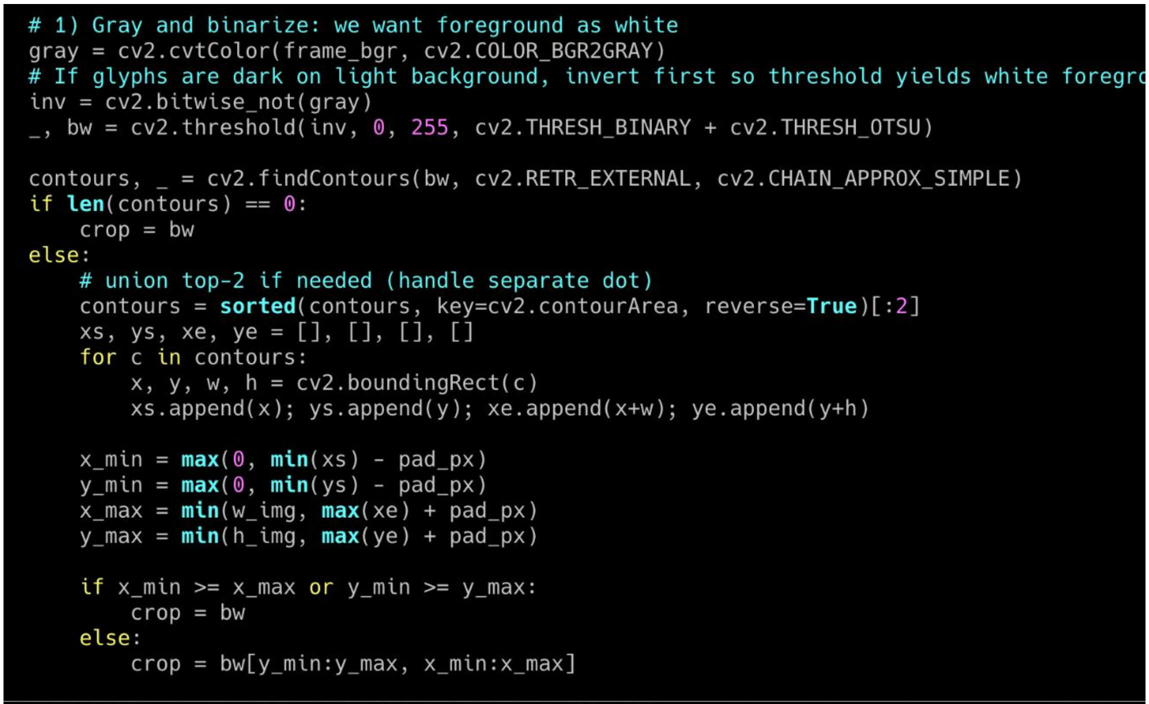
2.4 Image Retrieval, Preprocessing, and Using the Model

Video capture and image preprocessing are handled by the OpenCV library. This is a tool that can capture and alter an image in many ways. To capture a camera video feed, we need to pass the camera index to OpenCV. Then we run an infinite loop capturing frames and pass them over to another thread running preprocessing and getting the prediction from the model.

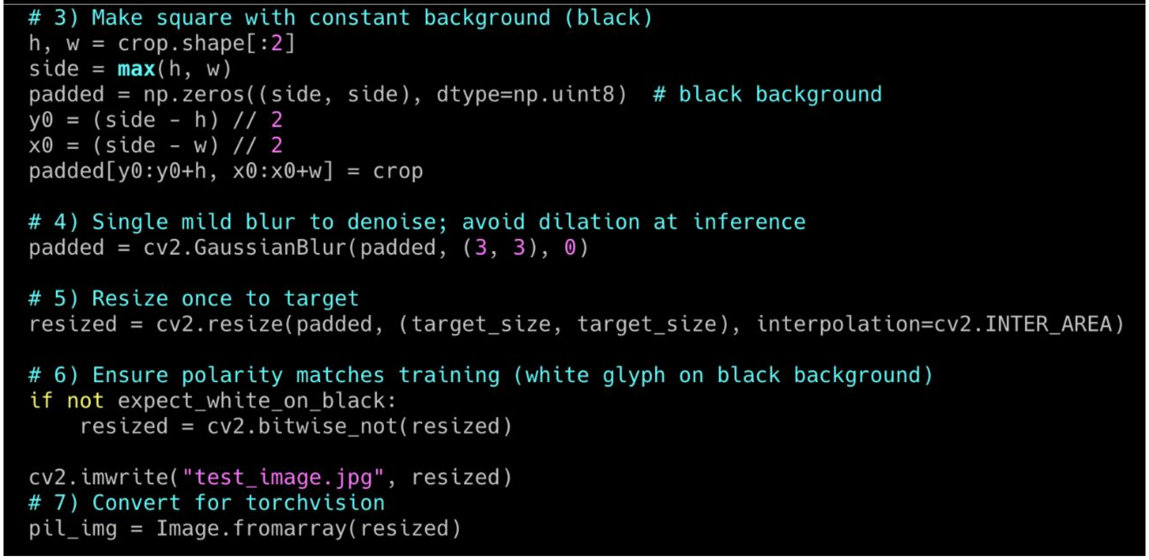


```
cap = cv2.VideoCapture(0)
if not cap.isOpened():
    print("Cannot open camera")
    running = False
    raise SystemExit

while True:
    ret, frame = cap.read()
    if not ret:
        break

    with lock:
        latest_frame = frame.copy()

    cv2.imshow('Hiragana Live', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

running = False
cap.release()
cv2.destroyAllWindows()
```

The preprocessing pipeline is in this order: grayscale, invert pixels, threshold the bits, find biggest contours, crop to a square around those contours, apply a little blur, resize to 28x28.

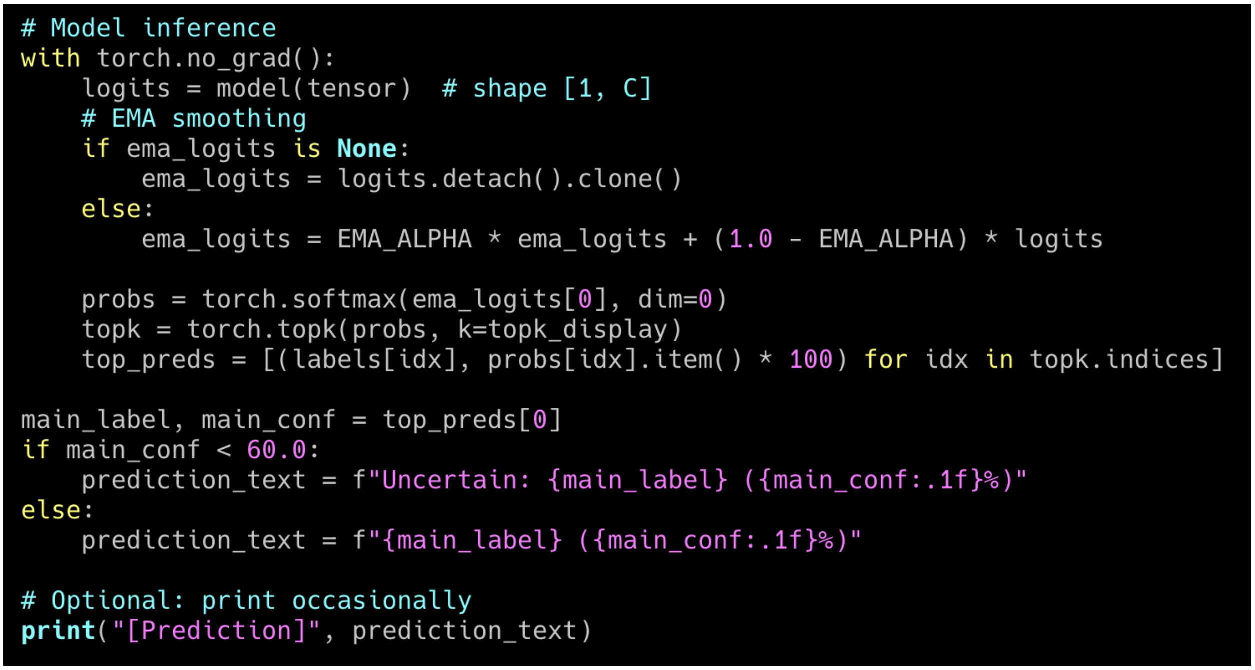


```
# 1) Gray and binarize: we want foreground as white
gray = cv2.cvtColor(frame_bgr, cv2.COLOR_BGR2GRAY)
# If glyphs are dark on light background, invert first so threshold yields white foregr
inv = cv2.bitwise_not(gray)
_, bw = cv2.threshold(inv, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

contours, _ = cv2.findContours(bw, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
if len(contours) == 0:
    crop = bw
else:
    # union top-2 if needed (handle separate dot)
    contours = sorted(contours, key=cv2.contourArea, reverse=True)[:2]
    xs, ys, xe, ye = [], [], [], []
    for c in contours:
        x, y, w, h = cv2.boundingRect(c)
        xs.append(x); ys.append(y); xe.append(x+w); ye.append(y+h)

    x_min = max(0, min(xs) - pad_px)
    y_min = max(0, min(ys) - pad_px)
    x_max = min(w_img, max(xe) + pad_px)
    y_max = min(h_img, max(ye) + pad_px)

    if x_min >= x_max or y_min >= y_max:
        crop = bw
    else:
        crop = bw[y_min:y_max, x_min:x_max]
```



```
# 3) Make square with constant background (black)
h, w = crop.shape[:2]
side = max(h, w)
padded = np.zeros((side, side), dtype=np.uint8)  # black background
y0 = (side - h) // 2
x0 = (side - w) // 2
padded[y0:y0+h, x0:x0+w] = crop

# 4) Single mild blur to denoise; avoid dilation at inference
padded = cv2.GaussianBlur(padded, (3, 3), 0)

# 5) Resize once to target
resized = cv2.resize(padded, (target_size, target_size), interpolation=cv2.INTER_AREA)

# 6) Ensure polarity matches training (white glyph on black background)
if not expect_white_on_black:
    resized = cv2.bitwise_not(resized)

cv2.imwrite("test_image.jpg", resized)
# 7) Convert for torchvision
pil_img = Image.fromarray(resized)
```

The image then gets passed to the model in the following code; we extract the prediction from the model here as well.



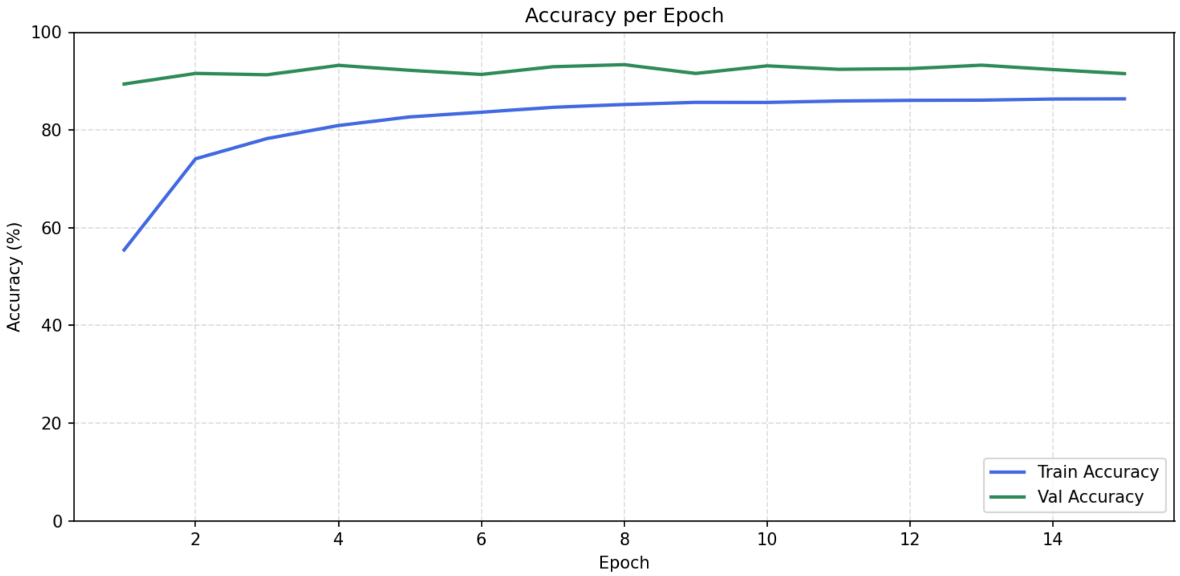
```
# Model inference
with torch.no_grad():
    logits = model(tensor)  # shape [1, C]
    # EMA smoothing
    if ema_logits is None:
        ema_logits = logits.detach().clone()
    else:
        ema_logits = EMA_ALPHA * ema_logits + (1.0 - EMA_ALPHA) * logits

    probs = torch.softmax(ema_logits[0], dim=0)
    topk = torch.topk(probs, k=topk_display)
    top_preds = [(labels[idx], probs[idx].item() * 100) for idx in topk.indices]

main_label, main_conf = top_preds[0]
if main_conf < 60.0:
    prediction_text = f"Uncertain: {main_label} ({main_conf:.1f}%)"
else:
    prediction_text = f"{main_label} ({main_conf:.1f}%)"

# Optional: print occasionally
print("[Prediction]", prediction_text)
```

The above code keeps a feed of frames from the video to the model to be predicted, doing some preprocessing beforehand to match the training data. The output is a confidence percentage along with the hiragana label. We also have implemented a form of this script that identifies a set of hiragana to then pass to google translate to get the English translation of the written text.

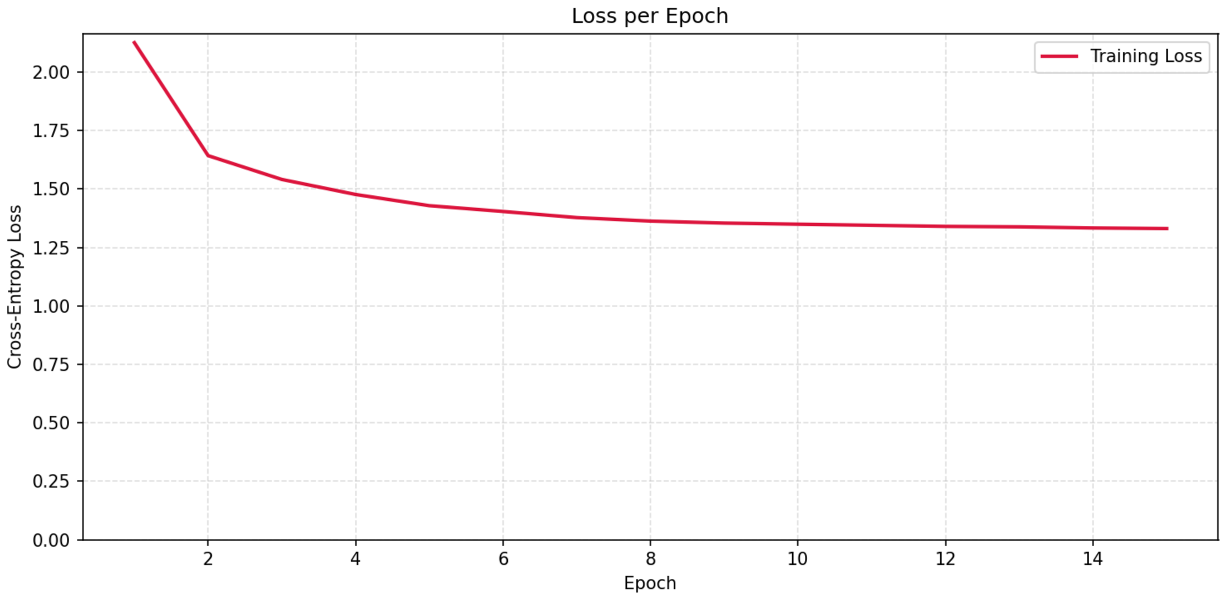
3 Key Findings

3.1 Analysis of Model Training Accuracy and Loss

As stated above in 2.2 Training the Convolutional Neural Network, the training dataset of 6000 images is fed into our CNN for training. We do this in 15 epochs, each time the model improves its neural network. The following graph models the agent's accuracy per epoch.



In this graph, the blue line represents our training accuracy, and the green line represents our validation accuracy. For training accuracy, our model begins at around 55% through the first epoch and slowly increases through each subsequent epoch until approaching 85%. For the validation accuracy, it starts at around 88% and hovers in that area for the remainder of the epochs.

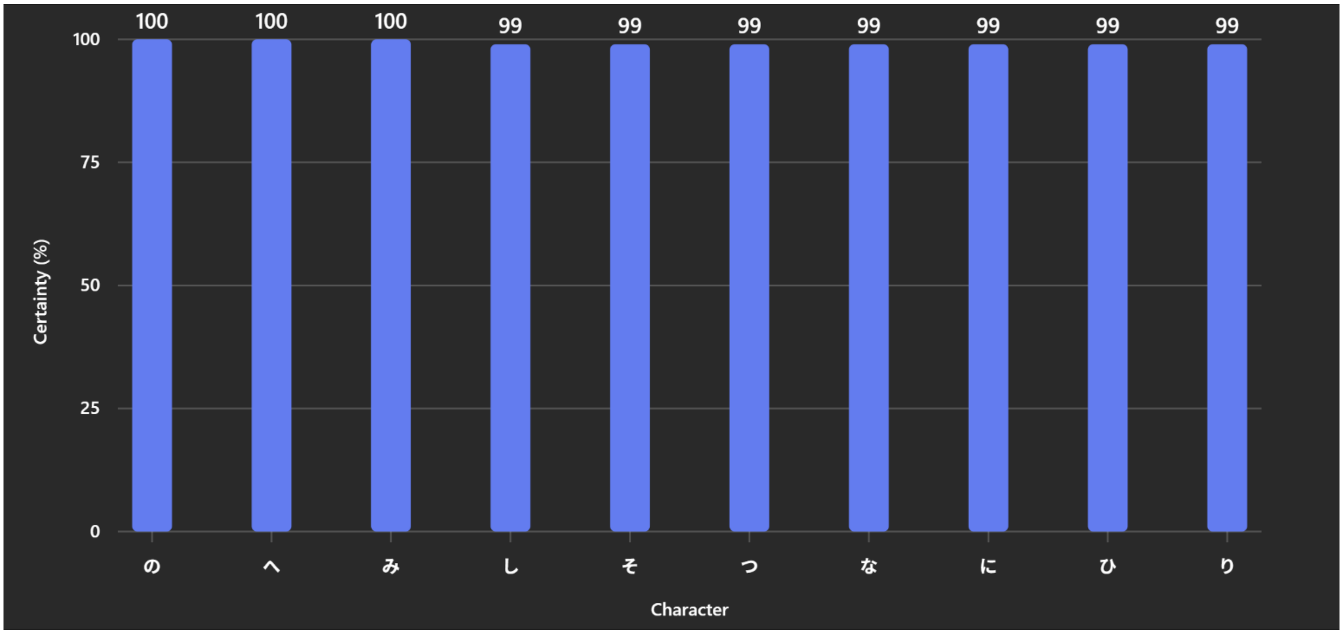


In this graph, our red line represents training loss. For the first epoch, our loss starts at around 2.25 and through each subsequent epoch reduces until approaching 1.30.

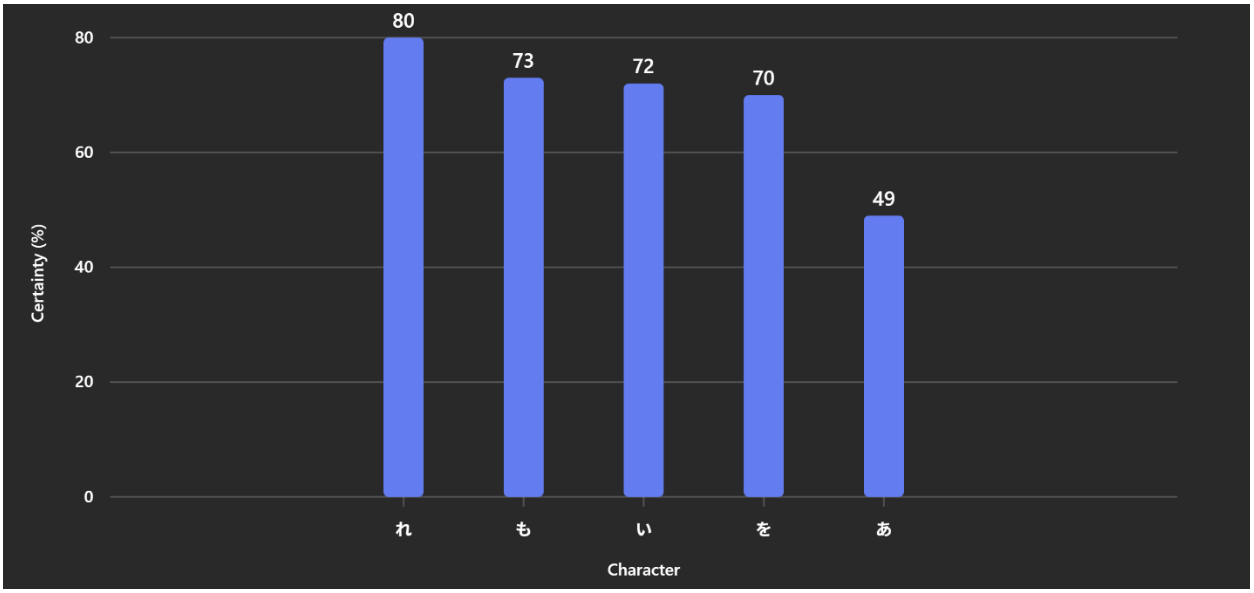
Total training time for our model would on average take around 2-3 minutes given standard CPU hardware on a Mac laptop. This time could be improved with the usage of a GPU.

3.2 **Analysis of Prediction Accuracy of Hiragana Characters**

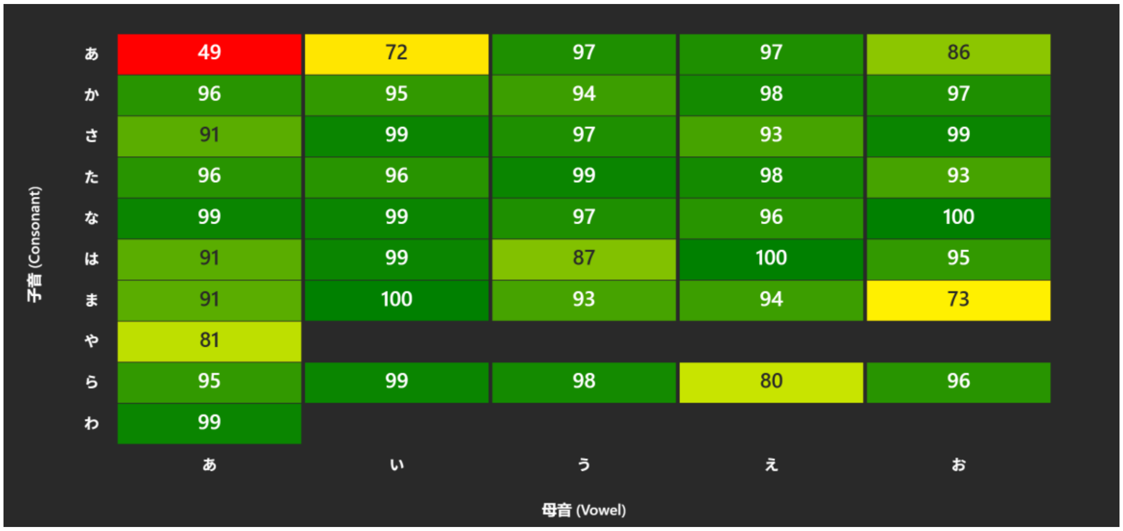
After the datasets were fully created and our model was properly trained, we proceeded to test our model's ability to accurately predict the Hiragana from the camera display. For this testing, we used a standard classroom document camera to capture our input drawing. We used standard paper with a sharpie marker to draw the characters one at a time for our model to interpret.



This graph shows our model's predictions for the top 10 characters, and its associated percentage of certainty. Our model predicted three characters with 100% accuracy, those being the hiragana for "no", "he", and "mi". Our model also predicted a total of 24 characters with over 95% confidence level as well as 36 characters over 90% accuracy. That means for over half of hiragana characters, our model is at least 95% confident in its prediction. Our model is also over 90% confident with over 78% hiragana characters.



This graph shows our model's prediction for its worst five characters, and its associated percentage of certainty. The graph clearly shows that even in our worst-case hiragana, it is still averaging 68.8% confidence. This means that our model can give a certainty of at least 49% per character and excluding our singular outlier of "a", it gives a minimum certainty of 70%.

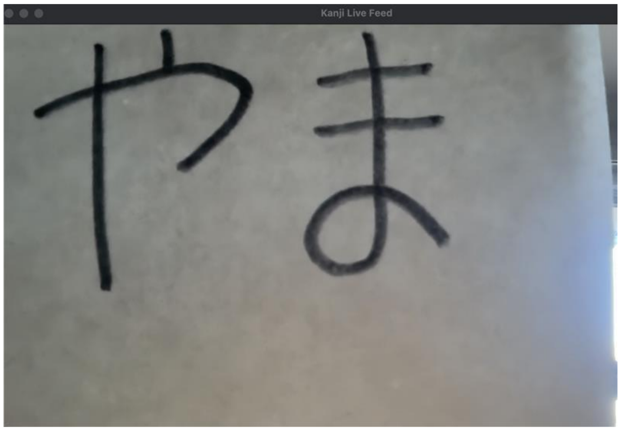


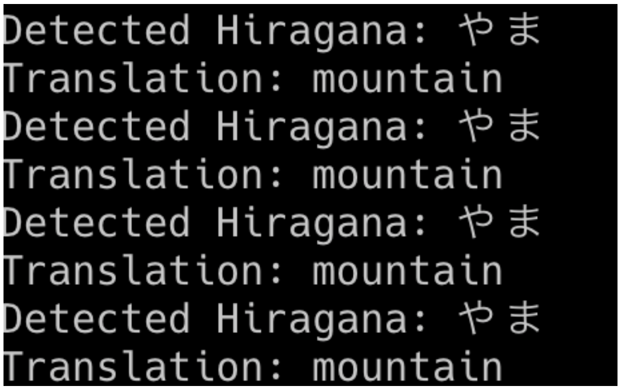
	あ	い	う	え	お
あ	49	72	97	97	86
か	96	95	94	98	97
さ	91	99	97	93	99
た	96	96	99	98	93
な	99	99	97	96	100
は	91	99	87	100	95
ま	91	100	93	94	73
や	81				
ら	95	99	98	80	96
わ	99				

The following chart is a heatmap representing all 46 hiragana our model processed. The average prediction confidence percentage for this dataset is 92.2%. Our model's best row in the hiragana chart was the "na, ni, nu, ne, no" row, averaging a percentage of 98.2% prediction confidence. Our model's best column in the hiragana chart was the "i" column, averaging a percentage of 94.9% prediction confidence.

3.3 **Analysis of Google Translated Hiragana**

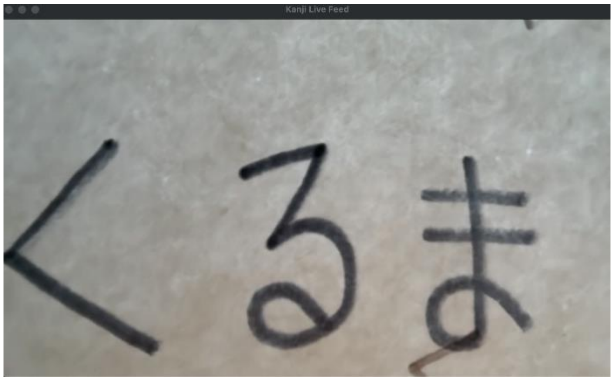
The final task we wanted our model to be capable of was the ability to translate a string of hiragana into an associated word. For it to do this, we would have to define the model where specific regions and boundaries between hiragana were. As stated in 1.4 Limitations, we were able to get the model to do this with limited success. Here are some of the results of our attempts.

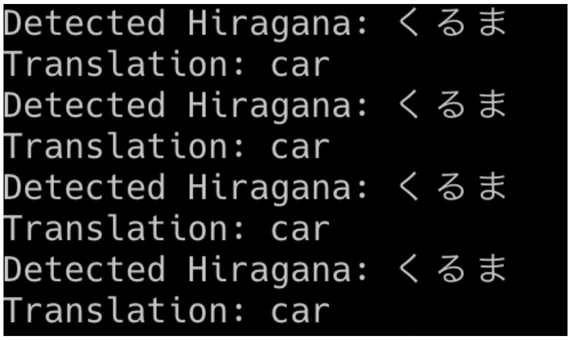




```
Detected Hiragana: やま
Translation: mountain
Detected Hiragana: やま
Translation: mountain
Detected Hiragana: やま
Translation: mountain
Detected Hiragana: やま
Translation: mountain
```

For the first word, "yama" which means mountain, we were able to get the model to see both the "ya" character as well as the "ma" character and output "yama" into our google translate feature. This returned mountain as we had hoped.





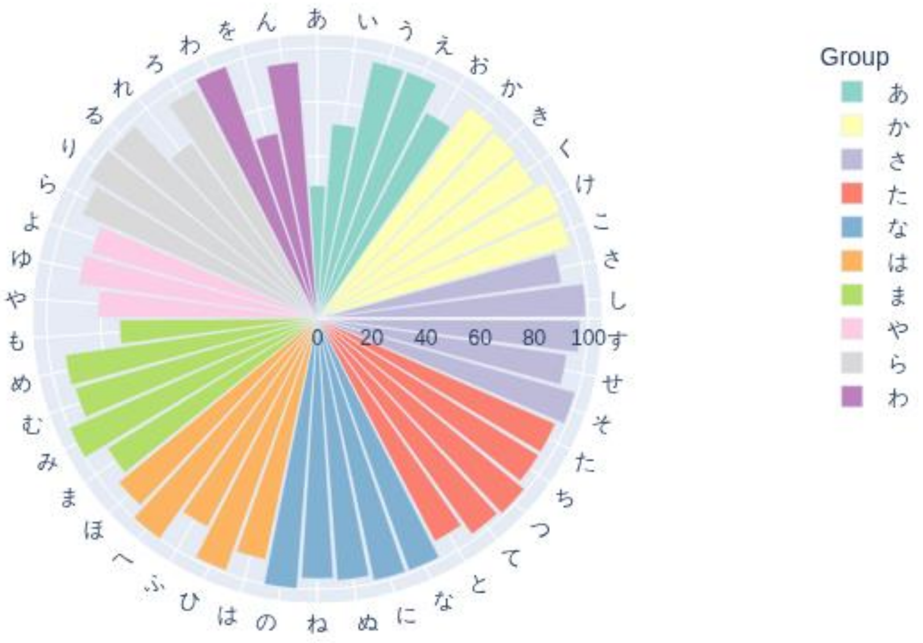
```
Detected Hiragana: くるま
Translation: car
Detected Hiragana: くるま
Translation: car
Detected Hiragana: くるま
Translation: car
Detected Hiragana: くるま
Translation: car
```

For our other successful word, we chose “kuruma” which means car. Again, our model was successful in recognizing three individual hiragana and passing that to our translator.

The reason we believe these two examples were the most successful was because each individual hiragana used in these examples is a singular structure and not divided into non-connected segments. Because the agent was not conflicted over separated parts being one combined hiragana or two separate hiragana, it was able to more accurately display the correct translation

3.4 Main Takeaways from the Results

Overall, we found that our model was quite successful in predicting handwritten hiragana characters. With the average confidence percentage being over 90%, it is safe to assume that it would be able to read most people’s attempt at a character.



The following chart is another representation of our model's prediction for each character. The more the bar is full, the closer to 100% the prediction confidence is. The circle is fairly full, meaning our model is very confident with most characters.

4 Conclusion

In conclusion, our AI model was generally successful in its ability to properly detect, predict, and translate hiragana. We believe that through further improvements in dataset variation and hiragana character boundary definition for our translator, it could become a quite reliable tool.

We found that there are many challenges when developing a project involving something abstract such as language and character identification. Specifically, we learned that finding a dataset that is all encompassing as well as robust and avoids overfitting is no straightforward process. It took many iterations of datasets for us to come to one that works well for our model.

We found this project to be a great way to explore more about AI image capturing and processing through tools like OpenCV and Pytorch. This allowed us to learn about transforming a video into a specific image frame and transforming it into a usable scaled version of itself. We also learned how to create datasets from different font packages and apply filters to them to vary our pool.

Overall, we found that this project was a good introduction and playground to the possibilities of AI specifically in language detection and interpretation.

5 References

5.1 **Fonts Used with Download Links**

ipagothic

https://japanesefonts.net/fonts/ipagothic-regular

KleeOne-Regular

https://github.com/fontworks-fonts/Klee

NotoSansJP-Regular
https://fonts.google.com/noto/specimen/Noto+Sans+JP

SlacksideOne-Regular
https://fonts.google.com/specimen/Slackside+One

TekitouPoem
https://github.com/Hagi42/TekitouPoem-Font

ZenKurenaido-Regular
https://fonts.google.com/specimen/Zen%20Kurenaido

Kaorigel-Z9YK

https://github.com/AkiSat310/fontfont4
Yomogi-Regular
https://github.com/satsuyako/YomogiFont

HinaMincho-Regular
https://github.com/satsuyako/Hina-Mincho

GL-CurulMinamoto
https://github.com/Gutenberg-Labo/GL-CurulMinamoto

AoyagiSosekiFont2
https://truefonts.net/fonts/aoyagisosekifont2/

KouzanMouhituFontOTF

https://truefonts.net/fonts/kouzanmouhitufontotf/