
	Politechnika Bydgoska im. J. J. Śniadeckich Wydział Telekomunikacji, Informatyki i Elektrotechniki Zakład Systemów Teleinformatycznych		
Przedmiot	Zaawansowane Techniki Sztucznej Inteligencji		
Prowadzący	prof. dr hab. inż. prof. PBS Piotr Cofta		
Temat	<i>Project</i>		
Student	Cezary Tytko		
Ocena		Data oddania spr.	

W tym etapie projektu wdrażam i dopracowuje algorytm przedstawiony w poprzednim etapie.

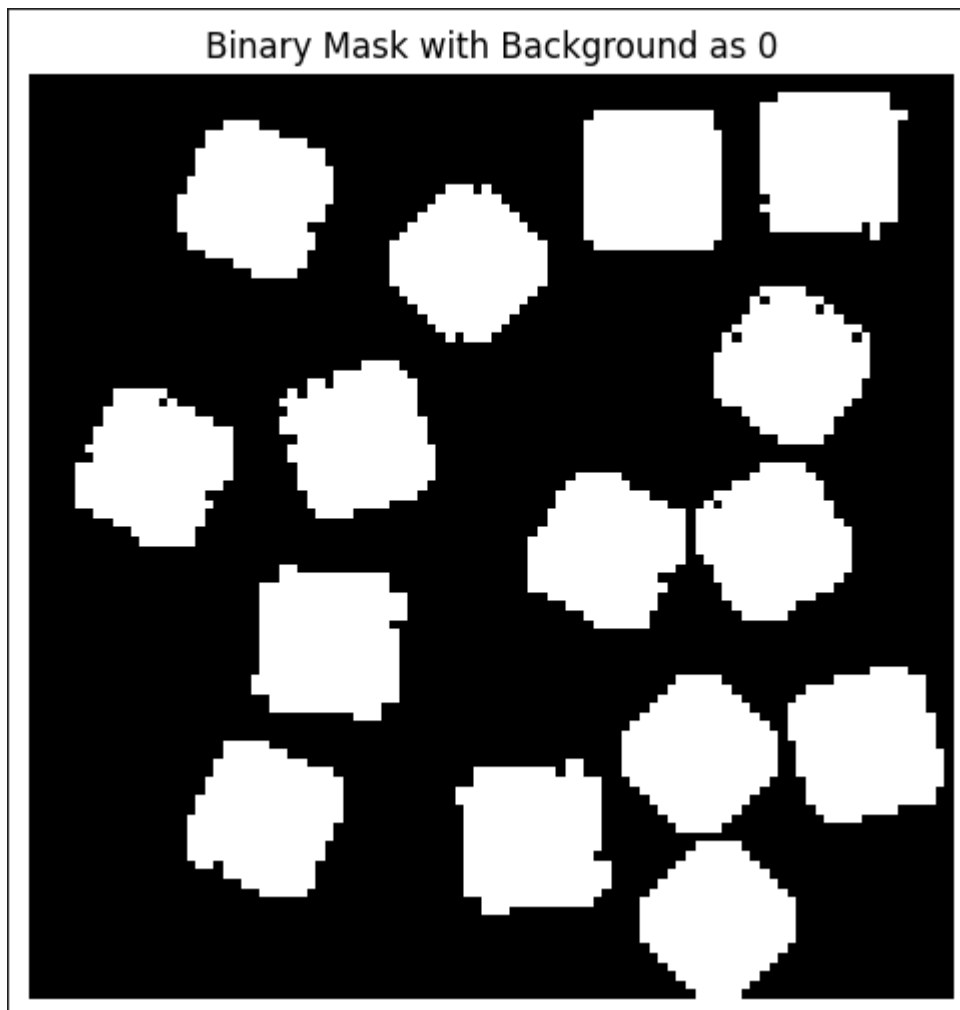
Zacząłem od przygotowania narzędzi do segmentacji obrazu i wydzielenia z niego wszystkich kostek jako osobne obrazy. Opracowałem w tym celu metodę:

```

1. def extract_dice_images_from_array_background_base(gray_image, showMask=True):
2.     if gray_image.dtype != np.uint8:
3.         gray_image = (gray_image * 255).astype(np.uint8) if gray_image.max() <= 1 else
gray_image.astype(np.uint8)
4.
5.     hist = cv2.calcHist([gray_image], [0], None, [256], [0, 256])
6.
7.     background_intensity = np.argmax(hist)
8.
9.     threshold_value = background_intensity
10.    binary_mask = np.where(gray_image != threshold_value, 1, 0).astype(np.uint8)
11.
12.    if showMask:
13.        plt.figure(figsize=(6, 6))
14.        plt.imshow(binary_mask, cmap='gray')
15.        plt.title("Binary Mask with Background as 0")
16.        plt.axis('off')
17.        plt.show()
18.
19.    contours, _ = cv2.findContours(binary_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
20.
21.    dice_images = []
22.    for contour in contours:
23.        x, y, w, h = cv2.boundingRect(contour)
24.        dice_image = gray_image[y:y+h, x:x+w]
25.        dice_image = resize_to_28x28(dice_image, background_intensity)
26.        dice_images.append(dice_image)
27.
28.    return dice_images
29.
30.
31.
32.
33.
34.
35.
36.
37.

```

Algorytm najpierw określa kolor tła, który stanowi największą część obrazu, na tej podstawie tworzona jest maska binarna, która wyciąga zarysy kostek z obrazu:



Na podstawie jest maski metoda z biblioteki openCv `cv2.findContours` pozwala wydzielić kontury, na podstawie których za pomocą metody `cv2.boundingRect` tworzymy prostokąty wydzielające kostki. Otrzymane w ten sposób obrazy kostek przeważnie były rozmiaru 17 na 17, dlatego powstała metoda zwiększająca rozmiar do 28 na 28 (ze względu na model z poprzedniego projektu), zwiększenia rozmiaru polega na umieszczeniu kostki w centrum i wypełnieniu brakujących pikseli na około kolorem tła. Algorytm sprawdziłem przez zliczenia ile kostek zostało wykrytych dla każdego z obrazów, dla wszystkich uzyskano 15 kostek.

Pierwsza wersja algorytmu klasyfikującego zakłada tylko wykorzystanie modelu sieci neuronowej wytrenowanego na poprzednim projekcie:

```
1. res = []
2. model = torch.load('dice_classifier.pth')
3. for idx, (img, label) in enumerate(zip(dice_x, dice_y)):
4.     dice_image_list = get_dices(img)
5.     probabilities = classify_images(model, dice_image_list)
6.     probabilities_dict = create_probability_dict(probabilities, dice_image_list)
```

```

7.     y, _ = sum_top_classes(probabilities_dict)
8.     res.append(abs(label - y))
9.
10.  wartosci, liczba_wystapien = np.unique(res, return_counts=True)
11.  slownik_wystapien = dict(zip(wartosci, liczba_wystapien))
12.  posortowany_slownik = dict(sorted(slownik_wystapien.items(), key=lambda x: x[1],
reverse=True))
13.  print(posortowany_slownik)
14.  print(posortowany_slownik[0] / dice5_count * 100)
15.

```

System klasyfikuje wszystkie kostki (bez podziału na właściwe i nie właściwe), a następnie wybiera 5 kostek co do których model był najbardziej pewien, model zawsze zwraca klasę wskazującą liczbę oczek, ale naiwnie założyłem, że dla niewłaściwych kostek model mniej pewnie (z mniejszym prawdopodobieństwem) będzie zwracał predykcję. W ostatnim kroku sumuję liczbę oczek (klas) z 5 wybranych kostek. Warto zauważyć, że ten algorytm nie wymaga trenowania modelu sieci neuronowej dlatego nie dzielimy danych na zbiory trening/walidacja, tylko całość posłuży nam do sprawdzenia algorytmu. Algorytm uzyskał 24 % celność, co jest wynikiem lepszym niż 10% dla modelu stałego, warto też zwrócić uwagę na to że błąd liczbowy jaki popełnia algorytm (jakbyśmy spojrzeli na problem regresyjnie, a nie klasyfikacyjnie) jest niewielki jeżeli porównamy odległość klasy przewidzianej od właściwej, jeżeli założymy tolerancję ± 1 uzyskamy 44% poprawności, dla ± 2 będzie to 62%, a dla ± 3 aż 79%.

Nie będą zadowolony z wyniku 24% postanowiłem rozbudować algorytm o model klasyfikujący kostki na właściwe i nie właściwe, ale nie miałem o etykietowanych danych. Wykorzystując algorytm segmentacji zapisałem obrazy kostek wydzielone z pierwszych 20, ręcznie przypisałem je do 2 folderów dobre i złe. Utworzyłem prostą sieć neuronową:

```

1.  class SimpleCNN(nn.Module):
2.      def __init__(self):
3.          super(SimpleCNN, self).__init__()
4.          self.conv_layers = nn.Sequential(
5.              nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1),
6.              nn.ReLU(),
7.              nn.MaxPool2d(2, 2),
8.
9.              nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
10.             nn.ReLU(),
11.             nn.MaxPool2d(2, 2),
12.         )
13.         self.fc_layers = nn.Sequential(
14.             nn.Flatten(),
15.             nn.Linear(7 * 7 * 32, 32),
16.             nn.ReLU(),
17.             nn.Dropout(0.3),
18.             nn.Linear(32, 2)
19.         )

```

```

20.
21.     def forward(self, x):
22.         x = self.conv_layers(x)
23.         x = self.fc_layers(x)
24.         return x
25.

```

Ze względu na małą liczbę danych (300) i niezbalansowanie klas, mamy dwa razy więcej kostek złych niż dobrych, augmentuje je przez nad próbkowanie obu klas, i zbalansowanie ich (tylko dla danych treningowych), zakres modyfikacji obrazów w trakcie augmentacji ulegał zmianie w celu poprawy wyników, ostatecznie wykorzystuję odbicia, obrót, oraz delikatna zmianę jasności i kontrastu co pomogło aby model się nie przeuczał w przypadku nad próbkowania aż do około 6000 (różnie w zależności od iteracji) obserwacji dla każdej klasy. W kolejnych iteracjach zmianie ulegał także model sieci neuronowej.

Finalnie stworzyłem taki skrypt:

```

1. res = []
2. device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
3. model_classifier = torch.load("dice_classifier.pth")
4. model_classifier.eval()
5. model_simple_classifier = load_model(SimpleCNN().to(device), "dice_simple_classifier.pth")
6. dices_len = dice5_count
7. enum = enumerate(zip(dice_x[:dices_len], dice_y[:dices_len]))
8. for idx, (img, label) in enum:
9.     dice_image_list = get_dices(img)
10.    img_tensor = torch.stack([torch.tensor(img).float() for img in
dice_image_list]).to(device)
11.    img_tensor = img_tensor.unsqueeze(1)
12.    probabilities = simple_classify_images(model_simple_classifier, img_tensor)
13.    probabilities_dict = create_simple_probability_dict(probabilities, dice_image_list)
14.    top5img = get_top_image_simple_classifier(probabilities_dict)
15.    classified_img = simple_classify_images(model_classifier, top5img)
16.    y = sum_dice_classes(classified_img)
17.    res.append(abs(label - y))
18.
19.
20. wartosci, liczba_wystapien = np.unique(res, return_counts=True)
21. slownik_wystapien = dict(zip(wartosci, liczba_wystapien))
22. posortowany_slownik = dict(sorted(slownik_wystapien.items(), key=lambda x: x[1],
reverse=True))
23. print(posortowany_slownik)
24. print(posortowany_slownik[0] / dices_len * 100)
25.

```

Jego idea polega na wykorzystaniu najpierw modelu klasyfikującego kostki na niepoprawne i popraw, tutaj przeniosłem rozwiązanie z pierwszej wersji, będąc pewnym że poprawnych jest zawsze 5 kostek wybieram 5 najbardziej pewnych klasyfikacji, reszta algorytmu jest jak poprzednio. Tym sposobem uzyskałem wynik 52%, z czego nie byłem zadowolony spodziewałem się że takie podejście da lepsze rezultaty, dlatego manipulowałem parametrami augmentacji i

siecią klasyfikującą na właściwe i nie właściwe, po kilku wersjach poprawiłem wynik do zadowalających mnie 74%, co jest solidnym wynikiem dla 26 klas.

Wnioski:

Podsumowując projekt uważam go za bardzo interesujący i wymagający etapowego podejścia do rozwiązania problemu, być może zastosowanie jednej dużej sieci neuronowej kompleksowo rozwiązało by cały problem w jednym kroku, tak samo interesujące byłoby podejście tylko z wykorzystaniem analizy obrazu bez sieci neuronowych i mogło by przynieść równie dobre wyniki. Najłabszym punktem mojego podejścia jest sieć klasyfikująca na właściwe i nie właściwe, ponieważ przy treningu zostało wykorzystane tylko 20 bazowych obrazów, co dało 300 kostek, dane zostały nadpróbkowane, jednak na zbiorze testowym model uzyskiwał w granicach 90-95%, co i tak jest mało miarodajne ze względu na niewielki rozmiar zbioru walidacyjnego.