Documentatie proiect - Kaggle

Ciuperceanu Vlad-Mihai
Grupa 251

1 Introducere

Scopul competitiei de Kaggle a fost de a clasifica imagini in 3 categorii diferite. Aceasta a fost o problema de invatare supravegheata, intrucat am avut la dispozitie etichetele pentru datele de antrenament. Am incercat diverse abordari asupra acestei probleme prin 2 modele diferite de Machine Learning si multiple variatiuni asupra acestora, fie prin hiperparametri, fie prin schimbari in folosirea modelelor.

Cele 2 modele distincte de ML incercate au fost *SVM*, respectiv *Retele Neuronale Convolutionale*. Ambele abordari sunt cunoscute pentru capacitatea de clasificare, cea ulterioara oferind, in special, rezultate foarte bune pentru clasificarea imaginilor si extragerea caracteristicilor importante din acestea.

Pana sa intram in detalii despre folosirea modelelor, vom prezenta cateva aspecte ale manipularii datelor, pe care le vom folosi in codul ambelor metode, pentru a asigura o acurate mai buna.

2 Prelucrarea datelor

2.1 Organizarea in Dataset si DataLoader

Pentru organizarea datelor, ne vom folosi atat de bibliotecile *pytorch* si *pandas*, cat si de o clasa pentru definirea unui Dataset personalizat, similara cu cea din laboratorul 6:

```
class CustomImageDataset(Dataset):
    def __init__(self, csv_file, img_dir, transform=None):
        self.data_frame = pd.read_csv(csv_file)
        self.img_dir = img_dir
        self.transform = transform

    def __len__(self):
        return len(self.data_frame)

    def __getitem__(self, idx):
        img_name = os.path.join(self.img_dir, self.data_frame.iloc[idx, 0] + '.png')
        image = Image.open(img_name).convert('RGB')
        label = int(self.data_frame.iloc[idx, 1]) if 'label' in
```

```
                    self.data_frame.columns else -1

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

        return image, label
```

Trebuie mentionat faptul ca, din imagini, ne intereseaza valorile pixelilor pe canalele de culori, acestea reprezentand feature-urile noastre si datele cu care vom lucra.

2.2 Normalizarea datelor

Pentru a obtine o acuratete mai buna, am normalizat datele. In acest caz, in care datele noastre sunt niste imagini color, am considerat ca vom obtine un rezultat mai bun daca normalizam fiecare canal din *RGB*, decat sa normalizam la nivel de imagine.

Astfel, am iterat o data prin setul de date pentru a putea precalcula media si deviatia standard. Ulterior, am retinut valorile obtinute pentru a putea sa nu mai rulam de fiecare data bucata de cod in care obtinem aceste valori, intrucat chiar si aceasta este relativ costisitoare ca timp, avand in vedere datele. Aceste operatii au fost facute astfel:

```
transform = transforms.ToTensor()
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
train_dataset = CustomImageDataset(csv_file='realistic-image-classification/train.csv
                img_dir='realistic-image-classification/train', transform=transform)
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True, pin_memory=True

def calculate_mean_and_std(loader):
    mean = torch.zeros(3)
    std = torch.zeros(3)
    total_images_count = 0
    for images, _ in loader:
        images = images.to(device, non_blocking=True)
        batch_samples = images.size(0)
        images = images.view(batch_samples, images.size(1), -1)
        mean += images.mean(2).sum(0)
        std += images.std(2).sum(0)
        total_images_count += batch_samples
    mean /= total_images_count
    std /= total_images_count
    return mean, std

mean, std = calculate_mean_and_std(train_loader)
mean_list = mean.tolist()
std_list = std.tolist()
print(f'Mean: {mean_list}, Std: {std_list}')
```

Obtinem:

```
Mean:[0.4985780715942383, 0.4727059006690979, 0.42571836709976196],
Std: [0.2130184918642044, 0.2089148461818695, 0.2122994065284729]
```

Apoi, putem aplica transformari asupra datelor, cele care ne intereseaza fiind sa le transformam in tensori si sa aplicam normalizarea. In final, incarcam datele in Dataset, iar, mai apoi, in DataLoader:

```
# transformations for the images
transform = transforms.Compose([
    #transforms.Resize((80, 80)), # all images are already 80x80
    transforms.ToTensor(), # transform the image to a tensor
    transforms.Normalize(mean=[0.4985780715942383, 0.4727059006690979,
    0.42571836709976196], std=[0.2130184918642044, 0.2089148461818695,
    0.2122994065284729])
    #transforms.Normalize(mean=mean_list, std=std_list)
])

# dataset initialization
train_dataset = CustomImageDataset(csv_file='realistic-image-classification/train.csv
                                   transform=transform)
val_dataset = CustomImageDataset(csv_file='realistic-image-classification/validation.
                                   transform=transform)
test_dataset = CustomImageDataset(csv_file='realistic-image-classification/test.csv',
                                   transform=transform)

# dataloader initialization
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, pin_memory=True
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False, pin_memory=True)
```

Am impartit datele in batch-uri de 32 dupa capacitatea locala de calcul. De asemenea, am setat atributul *shuffle=True* la datele de antrenare, pentru ca modelul sa nu le intalneasca mereu in aceeasi ordine cand se antreneaza, astfel obtinand o invatare mai buna a trasaturilor in cazul retelelor neuronale.

2.3 Salvarea predictiilor

Pentru salvarea predictiilor, presupunand ca le-am obtinut intr-o lista *predictions* si ca vrem sa salvam in fisierul *predictions.csv*, in care am copiat deja continutul din fisierul dat *sample_ submission.csv*, atunci putem folosi biblioteca *pandas* pentru a scrie label-urile prezise:

```
test_df = pd.read_csv('realistic-image-classification/test.csv')
test_df['label'] = predictions
test_df.to_csv('predictions.csv', index=False)
```

3 SVM

3.1 Privire de ansamblu

Primul model de Machine Learning incercat a fost *SVM - Support Vector Machine*. Acesta gaseste hiperplanul de margine maxima care separa cel mai bine datele din 2 clase diferite. Hiperparametrii sai sunt dati de $C = 1$, care determina cat de mare sau mica va fi acea margine pana la clase si $gamma = 0.1$, coeficient folosit la kernel. De asemenea, se poate seta tipul kernelului, in cazul nostru avand *kernel='rbf'*, intrucat acesta trateaza cel mai bine neliniaritatea datelor, asa cum intalnim in imagini.

Un model *SVM* poate clasifica date in 2 categorii, iar, cum noi avem 3 tipuri de label, vom avea nevoie de mai multi astfel de clasificatori.

In particular, am folosit varianta de *One-vs-One*, in care vom avea cate un *SVM* pentru fiecare pereche de clase, reiesind 3 astfel de modele cu care vom lucra. Fiecare dintre acestea va invata sa separe cate 2 clase diferite, urmand sa luam o decizie de predictie pe datele de test pe baza tuturor celor 3 modele.

3.2 Implementare

Instantierea modelelor arata astfel:

```
svm_1_2 = SVC(kernel='rbf', C=1, gamma=0.1)
svm_2_3 = SVC(kernel='rbf', C=1, gamma=0.1)
svm_3_1 = SVC(kernel='rbf', C=1, gamma=0.1)
```

Pentru acest tip de model, vom mai face o prelucrare a datelor inainte de antrenare, intrucat ne trebuie separararea lor in perechi:

```
train_images_1 = []; train_images_2 = []; train_images_3 = []
train_labels_1 = []; train_labels_2 = []; train_labels_3 = []

for i in range(len(train_dataset)):
    image, label = train_dataset[i]
    if label == 0:
        train_images_1.append(image)
        train_labels_1.append(label)
    elif label == 1:
        train_images_2.append(image)
        train_labels_2.append(label)
    elif label == 2:
        train_images_3.append(image)
        train_labels_3.append(label)

train_images_1_2 = train_images_1 + train_images_2
train_labels_1_2 = train_labels_1 + train_labels_2
train_images_2_3 = train_images_2 + train_images_3
train_labels_2_3 = train_labels_2 + train_labels_3
```

```
train_images_3_1 = train_images_3 + train_images_1
train_labels_3_1 = train_labels_3 + train_labels_1

train_images_1_2 = torch.stack(train_images_1_2)
train_images_2_3 = torch.stack(train_images_2_3)
train_images_3_1 = torch.stack(train_images_3_1)
train_labels_1_2 = torch.tensor(train_labels_1_2)
train_labels_2_3 = torch.tensor(train_labels_2_3)
train_labels_3_1 = torch.tensor(train_labels_3_1)

train_images_1_2 = train_images_1_2.view(train_images_1_2.size(0), -1)
train_images_2_3 = train_images_2_3.view(train_images_2_3.size(0), -1)
train_images_3_1 = train_images_3_1.view(train_images_3_1.size(0), -1)
```

Apoi, putem sa incepem antrenarea modelelor prin intermediul metodei *fit(data, labels)*:

```
svm_1_2.fit(train_images_1_2, train_labels_1_2)
svm_2_3.fit(train_images_2_3, train_labels_2_3)
svm_3_1.fit(train_images_3_1, train_labels_3_1)
```

In acest moment, toate cele 3 modele sunt antrenate si sunt pregatite pentru a obtine predictii pe datele de validare, respectiv de antrenare.

Procedura va fi similara atat pentru datele de validare, cat si pentru cele de testare, diferenta fiind ca obtinem o acuratete de pe urma datelor de validare, ceea ce va fi un indicator bun pentru hiperparametri si pentru acuratetea pe datele de test.

Dupa ce am preluat datele de validare, vom face predictii pe acestea din fiecare model folosind metoda *predict(data)*, adica pentru fiecare pereche de clase:

```
val_images = []
val_labels = []
for i in range(len(val_dataset)):
    image, label = val_dataset[i]
    val_images.append(image)
    val_labels.append(label)

val_images = torch.stack(val_images)
val_images = val_images.view(val_images.size(0), -1).numpy()
val_labels = np.array(val_labels)

val_pred_1_2 = svm_1_2.predict(val_images)
val_pred_2_3 = svm_2_3.predict(val_images)
val_pred_3_1 = svm_3_1.predict(val_images)
```

Ne intereseaza, pentru fiecare imagine, sa aflam categoria din care provine. Dreptele de separare formeaza, in plan / hiperplan, un "triunghi", astfel ca, ne dorim sa aflam situarea fiecarei date fata de aceste drepte. De fapt, ne intereseaza, dintre cele 3 label-uri prezise, care apare cel mai des. Acela ne va da predictia finala pentru fiecare imagine:

```
val_pred = np.vstack((val_pred_1_2, val_pred_2_3, val_pred_3_1)).T
val_pred = np.apply_along_axis(lambda x: np.bincount(x).argmax(), axis=1,
                    arr=val_pred)
```

Evident, ne intereseaza sa obtinem o acuratete pe aceste date de validare, pentru a ne da seama de cum se comporta si ce performanta au modelele noastre:

```
val_labels = np.array(val_labels)
val_acc = np.mean(val_pred == val_labels)
print(f'Validation accuracy: {val_acc}')
```

Procedand similar pentru datele de test, vom genera predictiile:

```
test_images = []
for i in range(len(test_dataset)):
    image, _ = test_dataset[i]
    test_images.append(image)

test_images = torch.stack(test_images)
test_images = test_images.view(test_images.size(0), -1).numpy()

test_pred_1_2 = svm_1_2.predict(test_images)
test_pred_2_3 = svm_2_3.predict(test_images)
test_pred_3_1 = svm_3_1.predict(test_images)

test_pred = np.vstack((test_pred_1_2, test_pred_2_3, test_pred_3_1)).T
test_pred = np.apply_along_axis(lambda x: np.bincount(x).argmax(), axis=1, arr=test_p
```

Acuratetea pe aceasta varianta este de 0.34, care, dupa cum vom vedea ulterior, poate fi usor surclasata de modelul *CNN*, motiv pentru care nu am insistat prea mult pe aceasta implementare.

3.3 Hiperparametri, matricea de confuzie

Matricea de confuzie obtinuta pentru acest model este:

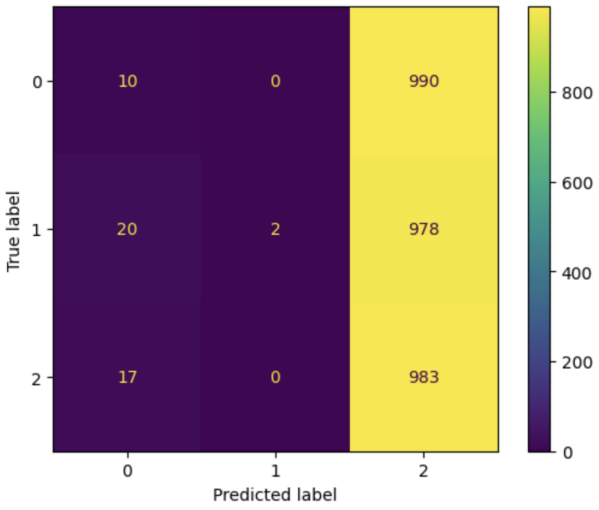

Figura 1: Matricea de confuzie

Din matricea de confuzie ne dam seama ca modelul nu a putut invata caracteristicile imaginilor si nu ajunge sa gaseasca un label potrivit pentru imaginea curenta. Este posibil sa apara situatia in care sa nu gaseasca un label predominant, de unde pare sa provina faptul ca selecteaza label-ul 2 adesea.

Asa cum mentionasem anterior, hiperparametrii in aceasta metoda sunt *C*, *gamma* si *kernel*.

Am stabilit ca, pentru acest tip de problema, este clar ca un kernel *rbf* va performa mult mai bine, detectand neliniaritatile. Raman ceilalti doi hiperparametri care pot fi modificati.

Atunci cand am incercat sa modificam *C*, obtinem aproximativ aceeasi acuratete:

Valoare C	**Acuratete**
0.5	0.33
1.0	0.34
3.0	0.33

Tabela 1: Acuratetea în functie de valorile lui C

Pentru valorile lui *gamma*, am obtinut:

Valoare gamma	**Acuratete**
0.01	0.32
0.1	0.34
0.5	0.33

Tabela 2: Acuratetea în functie de valorile lui gamma

Astfel, valorile alese mai sus sunt potrivite, incercand sa lasam o margine adecvata fata de clase. Din nou, exista modificari pe care le putem face ca modelul *SVM* sa obtina o performanta mai buna, dar, in final, acesta nu poate obtine niste performante foarte mari pe clasificarea imaginilor, astfel ca ne-am concentrat pe dezvoltarea unui model de retele neuronale.

4 CNN

4.1 Privire de ansamblu

Modelul cu care am obtinut cel mai bun rezultat este cel al *retelelor neuronale convolutionale*, care este cunoscut pentru performanta in problemele de clasificare a imaginilor.

Acesta este un tip special de model ce are la baza retelele neuronale, dar in care apar, ca element nou, straturile convolutionale. Acestea reprezinta niste *filtre convolutionale*, folosite pentru a extrage caracteristicile(feature-urile) cele mai importante pentru clasificare. De asemenea, vom folosi, in final, si straturi simple, *fully-connected*, la fel ca la

retelele neuronale simple. Acestea au rolul de a folosi caracteristicile extrase de straturile convolutionale pentru a face clasificarea.

Pe langa acestea, vom folosi *Max Pooling* pentru a reduce complexitatea si pentru a preveni *overfitting-ul*, detectand, in esenta, caracteristicile cele mai importante dupa care ar trebui sa clasificam, reducand zgomotul adiacent. De asemenea, vom normaliza batch-urile dupa aplicarea filtrelor si vom folosi *dropout* pentru straturile simple, conectate complet, ca o forma de regularizare. Functia de activare va fi *ReLU*, intrucat aceasta da cele mai bune rezultate in practica.

In cadrul tuturor incercarilor, am pastrat $kernel\_size = 3$, $stride = 1$, $padding = 1$ pentru straturile convolutionale (varianta cea mai populara), intrucat marirea $kernel\_size$ ducea la o acuratete mai slaba. Totodata, pentru *pooling* am folosit $kernel\_size = 2$, $stride = 2$, $padding = 0$ (din nou, cea mai populara optiune), intrucat aceasta a dat cele mai bune rezultate.

4.2 Implementare

Modelul nostru arata astfel, extinzand clasa Module:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)
        self.conv5 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.conv6 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        self.fc1 = nn.Linear(512 * 10 * 10, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 3)

        self.dropout = nn.Dropout(0.5)

        self.batch_norm1 = nn.BatchNorm2d(64)
        self.batch_norm2 = nn.BatchNorm2d(128)
        self.batch_norm3 = nn.BatchNorm2d(256)
        self.batch_norm4 = nn.BatchNorm2d(512)
        self.batch_norm5 = nn.BatchNorm2d(512)
        self.batch_norm6 = nn.BatchNorm2d(512)

        self.batch_norm_fc1 = nn.BatchNorm1d(512)
        self.batch_norm_fc2 = nn.BatchNorm1d(256)

    def forward(self, x): # the computation of the forward function
        x = F.relu(self.batch_norm1(self.conv1(x)))
```

```
        x = self.pool(F.relu(self.batch_norm2(self.conv2(x))))
        x = F.relu(self.batch_norm3(self.conv3(x)))
        x = self.pool(F.relu(self.batch_norm4(self.conv4(x))))
        x = F.relu(self.batch_norm5(self.conv5(x)))
        x = self.pool(F.relu(self.batch_norm6(self.conv6(x))))

        x = x.view(-1, 512 * 10 * 10)
        x = self.dropout(F.relu(self.batch_norm_fc1(self.fc1(x))))
        x = self.dropout(F.relu(self.batch_norm_fc2(self.fc2(x))))
        x = self.fc3(x)
        return x


model = CNN()
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
```

Modelul are 6 straturi convolutionale, care cresc in dimensiune, 3 straturi liniare, simple si incorporeaza normalizarea batch-urilor si dropout ca regularizare. Ordinea operatiilor cu straturile a fost aleasa tot prin incercari, aplicand astfel functia de activare si facand pooling odata la 2 straturi convolutionale. Dropout-ul se face asupra straturilor liniare, la final.

Functia de *loss* aleasa va fi:

```
criterion = nn.CrossEntropyLoss()
```

Optimizer-ul ales este *Adam*, cu un *learning_ rate* de 0.005 (am observat ca este nevoie de un learning_rate mai mare la inceput pentru a ajunge la un punct de minim mai bun).

```
optimizer = optim.Adam(model.parameters(), lr=0.005)
```

Vom ajusta *learning_ rate* in mod dinamic, pentru a ajuta modelul sa conveaarga spre minim:

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max', factor=0.1,
                                patience=3)
```

In cazul in care dupa 3 epoci nu se imbunatateste metrica, atunci vom inmulti rata de invatare cu factorul respectiv. De asemenea, este setat modul *'max'*. Initial am incercat cu modul *'min'*, prin care urmarea minimizarea loss-ului pe epoca, insa, am observat o crestere in acuratete atunci cand l-am modificat sa urmareasca maximizarea acuratetii pe datele de validare. Desi poate parea ca face ca modelul sa se axeze prea mult pe datele de validare, in aceasta situatie particulara, aceasta decizie a dus la cresterea acuratetii cu cel putin 0.02.

Vom lasa modelul sa se antreneze timp de 60 de epoci. Initial, foloseam *early stopping*, dar am observate ca este mai bine sa il lasam sa se antreneze pe parcursul tuturor epocilor. La final, vom reactualiza modelul cu cea mai buna varianta a sa obtinuta pe parcurs (in orice caz, datorita numarului de epoci, modelul va ajunge sa conveaarga cu acuratetea

spre o valoare, astfel ca acest lucru doar ne ajuta sa nu ramana ultimul model calculat, care ar putea sa devieze putin de la cel mai bun rezultat). In cadrul fiecarei epoci, vom antrena modelul, apoi vom vedea acuratetea pe datele de validare, pentru a ne putea da seama de cea mai buna varianta a modelului si de modificarile pe care le putem face la hiperparametri.

```
num_epochs = 60

best_val_loss = np.inf
best_epoch = 0
best_model_state = None

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device, non_blocking=True), labels.to(device, non_

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * images.size(0)

    epoch_loss = running_loss / len(train_loader.dataset)
    print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {epoch_loss:.4f}')

    model.eval()
    correct = 0
    total = 0
    val_loss = 0.0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device, non_blocking=True), labels.to(device, 
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    val_accuracy = 100 * correct / total
    val_loss = val_loss / len(val_loader.dataset)
    print(f'Validation Accuracy: {val_accuracy:.2f}%, Validation Loss: {val_loss:.4f}

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_epoch = epoch
```

```
            best_model_state = model.state_dict()
        # else:
        #     if epoch - best_epoch > 7:
        #         print('Early stopping')
        #         break
        # scheduler.step(epoch_loss)
        scheduler.step(val_accuracy)

model.load_state_dict(best_model_state)
```

Antrenarea se face cu modelul in modul *train()*, iar evaluarea se face in modul *eval()*. Pentru antrenare se face si *forward-propagation* si *backpropagation*, insa, pentru evaluarea datelor, se dezactiveaza propagarea inapoi, pentru a creste viteza de calcul, prin precizarea:

```
with torch.no_grad():
```

Desigur, pe parcurs, se vor calcula loss-ul si acuratetea pe datele curente. La final, vom reactualiza modelul cu varianta sa care a dat cea mai buna acuratete pe datele de validare:

```
model.load_state_dict(best_model_state)
```

Pentru datele de test, vom proceda asemanator cu cele de validare, dupa ce :

```
model.eval()
predictions = []
with torch.no_grad():
    for images, _ in test_loader:
        images = images.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        predictions.extend(predicted.cpu().numpy())
```

Cu acest model am obtinut cea mai buna acuratete, de 0.757. Desigur, modificari asupra hiperparametrilor vor schimba acuratetea, dar, in general, se va obtine o acuratete de cel putin 0.7.

4.3 Hiperparametri, matricea de confuzie

Matricea de confuzie obtinuta pentru acest model este:

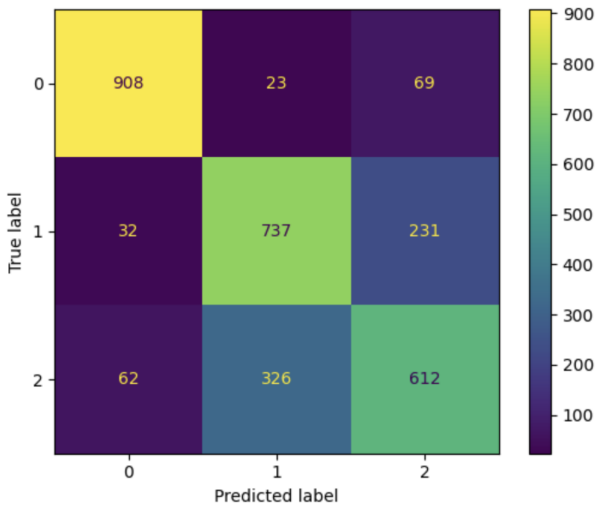

Figura 2: Matricea de confuzie

In acest caz, avem mai multi hiperparametri si, in general, mai multe alegeri de facut pentru modelul nostru. Acestea includ: numarul de straturi din fiecare tip ale modelului, dimensiunile pentru filtrele convolutionale si pentru pooling, rata de dropout, ordinea operatiilor cu straturile, numarul de epoci, rata de invatare, tipul de ajustare dinamica al acesteia, decizia de pastrare a modelului. Vom vedea, in cele ce urmeaza, cum influenteaza acestia predictiile.

Acestea sunt diverse incercari facute care au crescut acuratetea pe datele de validare si de test(desigur, acestea pot sa difere, intr-o masura relativ mica, in functie de initializarea cu valori a straturilor):

Model	**Caracteristici**	**Acuratete**
1	20 de epoci, 3 straturi convolutionale, 2 fully-connected, learning_rate de 0.001, fara scheduler, retine ultimul model	0.516
2	10 epoci, 3 straturi convolutionale, 2 fully-connected, learning_rate de 0.001, scheduler care minimizeaza loss-ul, patience de 3, early stopping	0.64
3	10 de epoci, 5 straturi convolutionale, 3 fully-connected	0.65
4	20 de epoci, 4 straturi convolutionale, 3 fully-connected	0.704
5	25 de epoci, patience de 4, salvam modelul cel mai bun	0.697
6	40 de epoci, 4 straturi convolutionale, 4 fully-connected, scheduler care maximizeaza acuratetea pe validare, patience de 3, early stopping	0.714
7	50 de epoci, learning_rate de 0.005 si patience de 2	0.734
8	50 de epoci, learning_rate de 0.048 si patience de 2	0.72
9	50 de epoci, learning_rate de 0.001, patience de 3, droupout de 0.375	0.723
10	60 de epoci, 6 straturi convolutionale, 3 fully-connected, dropout de 0.5	0.757

Tabela 3: Acuratetea modelelor în functie de modificarea hiperparametrilor

Astfel, din practica, am observat ca:

- este important sa avem un numar semnificativ de epoci, pentru a asigura convergenta modelului; este posibil sa obtinem rezultate relativ bune si cu un numar de epoci mai mic, dar nu ne asigura o acuratete suficient de buna si nici nu ne putem baza ca vom putea replica rezultatul. In cazul nostru, un numar de epoci de cel putin 60 este suficient.

- trebuie sa avem mai multe straturi convolutionale, pentru a avea cat mai multe filtre care detecteaza caracteristicile. Totusi, vom fi nevoiti sa facem si pooling dupa un numar de straturi convolutionale, care va reduce dimensiunea imaginilor. Este important sa nu reducem exagerat de mult dimensiunea, ajungand sa pierdem din informatie. In cazul nostru, solutia a fost data de 6 straturi convolutionale, 3 straturi fully-connected si pooling dupa fiecare 2 straturi de filtre. Valorile pentru dimensiunea kernel-ului, precum si stride si padding vor fi cele clasice, intrucat acestea dau cea mai buna performanta.

- regularizarea adusa de dropout este semnificativa, pentru a asigura ca nu ajungem sa facem overfitting. In cazul nostru, o rata de 0.5 a functionat cel mai bine - atat pentru o rata mai mica (de ex. 0.375), cat si pentru o rata mai mare (de ex. 0.8), obtinem o performanta mai slaba, intrucat fie lasam prea multi neuroni din straturile fully-connected, fie prea putini, astfel ca invatarea nu mai este eficienta.

- ordinea operatiilor cu straturile conteaza, intrucat, de exemplu, aplicarea normalizarii dupa ce am aplicat functia de activare pe datele trecute prin stratul convolutional duce la scaderea acuratetii cu pana la 0.03, din incercarile avute pe problema noastra. De aceea, am pastrat aceasta formula, intrucat a dat cele mai bune rezultate.

- rata de invatare este esentiala in minimizarea loss-ului si in cresterea preciziei de prezicere. O rata de invatare mai mica ne va conduce cu o siguranta mai mare spre un minim, dar va dura mai mult timp (epoci). De asemenea, este posibil sa ramana blocal intr-un minim local, de unde nu va putea scoate o acuratete suficient de buna. De aceea, am vazut ca este nevoie de un learning_rate putin mai mare decat de obicei, adica de 0.005. Totodata, este important ca acesta sa fie ajustat in timp, in functie de conditiile intalnite. Astfel, am introdus un scheduler, care modifica dinamic rata de invatare. In functie de performantele dorite pe metrica monitorizata si de un patience dat, acesta inmulteste rata de invatare cu un factor, pentru a reduce learning_rate-ul ca modelul sa fluctueze mai putin. Combinatiile cu care am obtinut cele mai bune rezultate au fost de *learning_rate=0.005, patience=2* (pentru un numar mai mic de straturi convolutionale, in care ne doream sa invete mai repede si cu o rata mai abrupta), respectiv *learning_rate=0.001, patience=3* (cu mai multe straturi convolutionale, in care il lasam mai multe epoci in care sa ajunga la performanta dorita, asa cum am facut in final). Monitorizarea metricii a fost facuta in 2 feluri diferite: minimizarea loss-ului pe datele de antrenare, respectiv maximizarea acuratetii pe datele de validare. Prima varianta este mai sigura, intrucat invata cat se poate de bine datele de antrenare, urmand sa testeze pe datele de validare. Totusi, in cazul nostru, am obtinut rezultate mai bune in momentul in care am schimbat monitorizarea pe maximizarea acuratetii pe datele

de validare, bazandu-ne pe faptul ca datele de validare sunt un indicator foarte bun si pentru datele de test (obtinand o crestere, la acel moment, de 0.02).

- am decis sa pastram cel mai bun model, intrucat ajungem la convergenta si ne dorim cel mai bun rezultat din acea zona in care a ajuns modelul. Initial, am folosit early stopping, fie dupa 3, 5 sau 7 epoci, insa aceasta metoda nu ne dadea o idee suficient de clara asupra capacitatii modelului, mai ales pe un numar care nu era suficient de mare de epoci. In final, rezultatul fara early stopping este cel putin la fel de bun ca acela in care aplicam aceasta metoda, intrucat ii dam timp modelului sa faca o clasificare mai buna. Desigur, ne bazam pe faptul ca datele de validare ilustreaza suficient de bine comportamentul datelor de test, ceea ce s-a intamplat in problema noastra.