Nume: Sociu Daniel
Grupa: 242

Documentatie proiect kaggle

În acest proiect am avut de făcut un model de Machine Learning care să clasifice nişte imagini astfel încât acurateţea lui să fie cât mai mare. Deci am folosit în toate cazurile un model de clasificare şi nu de regresie. Datele de antrenare şi validare sunt de pe kaggle, de la competiţia: Monochrome Dreams Classification, unde s-au trimis si rezultatele testării modelului.

Observăm ca datele sunt de tip monochrome si 32 de pixeli, adica au formatul 32x32x1, fiind diferit de cele colore care ar fi avut formatul 32x32x3. Am folosit librăriile tenserflow – keras, şi sklearn – SVM, KNN, NB şi LDA (Linear Decrease Analysis).

Pentru modelele din sklearn am transformat imaginile intr-un vector de lungime 32*32*1 = 1024 deoarece nu putem antrena majoritatea modelelor pe matrici, şi am scalat datele cu MinMaxScaler() pentru că a dat cele mai bune rezultate comparativ cu StandardScaler şi Robust, care este echivalent cu a împărţi valorile pixelilor la valoarea 255.

Obtinerea datelor:

```
X_train_vanilla = get_images("data/train/", X_data_train)
X_validation_vanilla = get_images("data/validation/", X_data_validation)
X_test_vanilla = get_images("data/test/", X_data_test)

X_train = X_train_vanilla.flatten().reshape(X_train_vanilla.shape[0], np.prod(X_train_vanilla.shape[1:]))
y_train = X_data_train[1]
X_valid = X_validation_vanilla.flatten().reshape(X_validation_vanilla.shape[0],
np.prod(X_validation_vanilla.shape[1:]))
y_valid = X_data_validation[1]
X_test = X_test_vanilla.flatten().reshape(X_validation_vanilla.shape[0],
np.prod(X_validation_vanilla.shape[1:]))
```

Şi prelucrarea:

```
preprocessor_KNN = MinMaxScaler()
X_train = preprocessor_KNN.fit_transform(X_train)
X_valid = preprocessor_KNN.transform(X_valid)
X_test = preprocessor_KNN.transform(X_test)
```

KNN – k-nearest neighbors

Ca primă încercare am folosit KNN, mai exact KneighborsClassifiers din sklearn.neighbors, unde am folosit un for pentru a verifica care este cel mai bun număr de vicini pentru acest model.

```
knn_scores =[]
 for i in range(5,15):
    KNN_model = KNeighborsClassifier(i)
    KNN_model.fit(X_train_KNN, y_train)
    score = KNN_model.score(X_valid_KNN, y_valid)
    print (i, score)
    knn_scores.append({"neighbors": i, 'score': score})
```

Am obţinut ca output următoarele valori:

- 5: 0.4772
- 6: 0.4824
- 7: 0.483
- 8: 0.4834
- 9: 0.4916
- 10: 0.4906
- 11: 0.4904
- 12: 0.4908
- 13: 0.485
- 14: 0.4844

Deci observăm ca 9 vecini este optim in cazul datelor noastre, aşadar am folosit următorul model pentru a antrena datele:

```
KNN_model = KNeighborsClassifier(
    n_neighbors=9,
    algorithm='auto',
    n_jobs=-1,
    p=1
)
```

Am încercat să folosesc şi alti algoritmi al KNeighborsClassifier, şi anume kd_tree, ball_tree şi brute dar au avut valori mai slabe decât cel default (auto). Am comparat şi valorile rezultate când am folosit l1 si l2, dar l1 a avut rezultate mai bune decat l2. (p=1 respectiv p=2).

Score: 0.503

După am incercat să obţin cea mai bună valoare posibilă pe modelul de tip Naive Bayes. Am folosit aceeasi idee ca la laborator în care am convertit imaginea cu digitize in “buckets”. Dupa am verificat care e cea mai buna valoare pentru numarul de buckets:

```
for num_bins in range(2,50,2):
    bins = np.linspace(start=0, stop = 255, num=num_bins)
    train = values_to_bins(X_train, bins)
    valid = values_to_bins(X_valid, bins)
    naive_bayes_model.fit(train, y_train)
    score = naive_bayes_model.score(valid, y_valid)
    print ('bins: ' + str(num_bins) + ':',  score)
    bins_scores.append({"bins": num_bins, "score": score})
```

Output-ul fiind:

- 0.3904
- 3: 0.3582
- 5: 0.3856
- 7: 0.3894
- 9: 0.391
- 10: 0.392
- 11: 0.3916
- 12 0.392 -- best with MultinomialNB -> 12 bins
- 13: 0.3906
- 15: 0.3894
- 17: 0.391
- 19: 0.3908
- 21: 0.3912
- 23: 0.3904
- 25: 0.3906

NB – Naive Bayes

Am încercat mai multe tipuri de naive bayes models, şi anume MultinomialNB, BernoulliNB şi GaussianNB. Am obținut diverse valori dar Gaussian a fost cea mai buna:

1. Multinomial: 0.392
   naive_bayes_model = MultinomialNB(alpha=0.2)

2. Bernoulli: 0.3214
   bernoulli_naive_bayes = BernoulliNB(binarize=0.2)

3. Gaussian: 0.3996
   gaussian_naive_bayes = GaussianNB()

Deci GaussianNB a obtinut 0.3996, scorul maxim comparativ cu restul NB, dar era mult prea mic pentru a trimite o submisie cu acest model.

LDA – Linear Discrimiant Analysis

După aceasta am trecut la modelul LinearDiscriminantAnalysis din sklearn.discriminant_analysis, care a obținut cele mai bune valori de până acum. LDA fără parametrii dădea o valoare de 0.588 ca test acuratețe.

Am încercat să schimb parametrii și am obținut următoarelel valori:

1. solver-ul lsqrt si shrinkage-ul auto: 0.596

```
linear_discriminant_model = LinearDiscriminantAnalysis(
    solver='lsqr',
    shrinkage='auto'
)
```

2. solver-ul eigen si shrinkage-ul covariance.ShrunkCovariance(): 0.604

```
linear_discriminant_model = LinearDiscriminantAnalysis(
    solver='eigen',
    covariance_estimator=covariance.ShrunkCovariance()
)
```

3. solver-ul lsqr si shrinkage-ul covariance.ShrunkCovariance(): 0.605

```
linear_discriminant_model = LinearDiscriminantAnalysis(
    solver='lsqr',
    covariance_estimator=covariance.ShrunkCovariance()
)
```

Am obținut 0.623 pe kaggle cu această submisie.

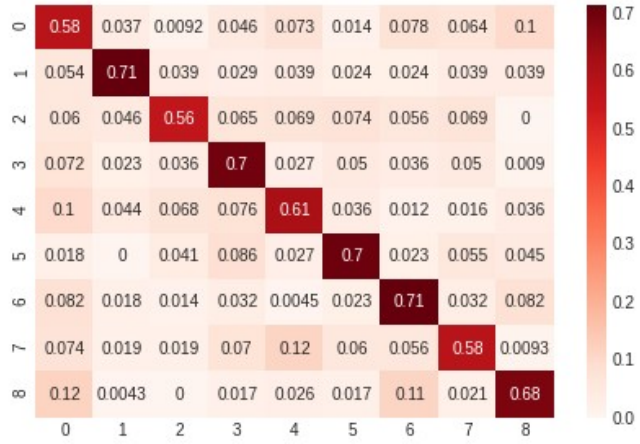
SVM – Support Vector Machine

Am testat toți kernelii posibili, linear, poly, rbf, sigmoid, precomputed și am ajuns la concluzia ca rbf este cel mai bun, după care am încercat să folosesc diferiți parametrii pentru model (pe acest model am facut testele pe 10.000 date pentru a rula intr-un timp relativ scurt și am folosit funcția train_test_split, și după ce am ales cel mai bun model, am folosit toate datele de train pentru a-l antrena).

Mai jos avem modelele de SVM:

1. SVM, kernel rbf, decision_function_shape ovo: 0.648

```
SVC_model = SVC(
    C=0.8,
    kernel='rbf',
    cache_size=4000,
    tol=1e-5,
    decision_function_shape="ovo",
    random_state=0
)
```

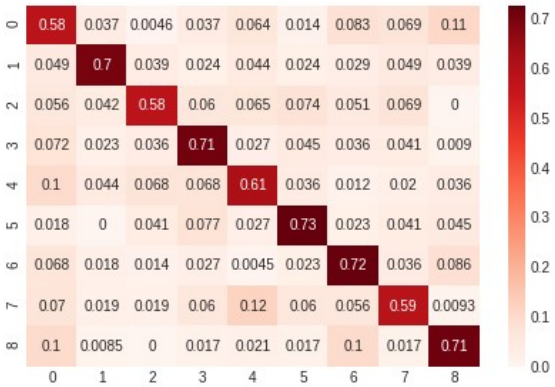


2. SVM, kernel rbf, decision_function_shape ovr: 0.65

```
SVC_model = SVC(
    C=0.8,
    kernel='rbf',
    cache_size=4000,
    tol=1e-5,
    decision_function_shape="ovr",
    class_weight='balanced',
    random_state=0
)
```

3. SVM, kernel rbf, decision_function_shape ovr: 0.6605, aici am schimbat C înapoi la 1 şi am obținut: 0.6605

```
SVC_model = SVC(
    C=1.0,
    kernel='rbf',
    cache_size=4000,
    tol=1e-5,
    decision_function_shape="ovr",
    class_weight='balanced',
    random_state=0
)
```



CNN – Convolutional Neural Network

Am folosit keras.Sequential ca model de neural network şi diferite layere si modele bazate pe arhitecturi de CNN precum LeNet-5 si VGG16. În primul rând am încercat un CNN cu valori de bază care avea o structură asemanătoare cu LeNet-5. De asemenea am adaugat early stopping şi am testat diferiți optimizers dar am ajuns la concluzia ca adam este obține cele mai bune valori şi este recomandat să-l folosim în curs. Aici datele de intrare sunt de tipul 32x32x1.

Pentru cazurile in care nu am folosit image augmentation, am compilat modelul cu următorii parametri:

```
early_stopping = callbacks.EarlyStopping(
    min_delta=0.01,
    patience=10,
    restore_best_weights=True
)
adam = keras.optimizers.Adam(
    learning_rate=0.0005,
    name="Adam"
)
neural_model.compile(
    optimizer = adam,
    loss = "categorical_crossentropy",
    metrics=['accuracy']
```

```
)
result = neural_model.fit(
    X_train_CNN,
    y_train_CNN,
    batch_size = 128,
    validation_data=(X_valid_CNN, y_valid_CNN),
    epochs = 100,
    #shuffle=True,
    #verbose=False,
    use_multiprocessing=True,
    callbacks=[early_stopping]
)
```

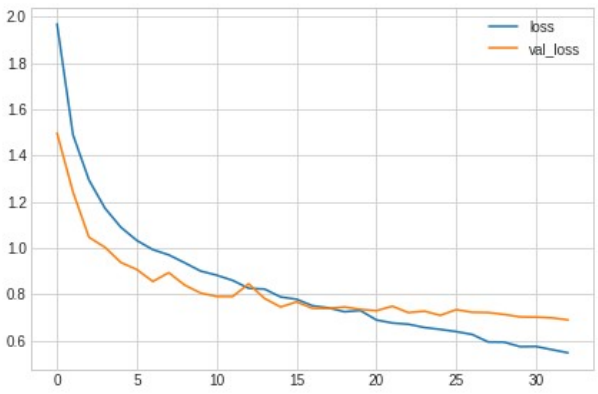
Am folosit 100 epochs deoarece am early stopping care are o toleranță de 10 epoci care nu îmbunătățesc soluția cu cel putin 0.01. Având o clasificare loss-ul este categorical crossentropy, iar un metricul accuracy pentru a obține acuratețea modelului pe fiecare epocă. Am rulat tot timpul cu verbose=True pentru a vedea cum se modifică acuratețea între epoci, și cum e la final. Pentru adam am schimbat doar learning rate-ul, depinzând de cum creștea accuracy-ul între epoci, dar în principal a stat 0.0005.

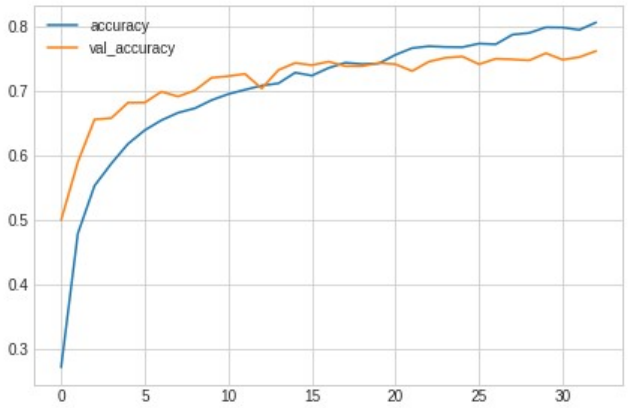
Pentru primul model am am folosit un 3 layere de convoluție și am obținut următoarele rezultate pe 10.000 de date (pentru testing, le-am antrenat pe 10.000 sau pe 5.000, depinzând de numărul parametrilor):

Accuracy: 0.8062499761581421
Val accuracy:0.7620000243186951

```
neural_model = keras.Sequential ([
    layers.Input(shape=(32,32,1)),
    layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dropout(0.5),
    layers.Dense(9, activation="softmax")
])
```

Am folosit un layer de convoluție pentru a crea mai multe filtre, și am folosit maxpooling pentru a injumătăți numărul pixelor al imaginii. După fac flatten pentru a-l reduce la o formă vectorială si folosesc dropout ca formă de normalizare. După un layer de softmax care obține indicele, adică clasa din care face parte. Sunt 9 perceptroni pentru că avem 9 clase in datele noastre de antrenare. Soluția a obtinut un punctaj de 0.79 de kaggle.





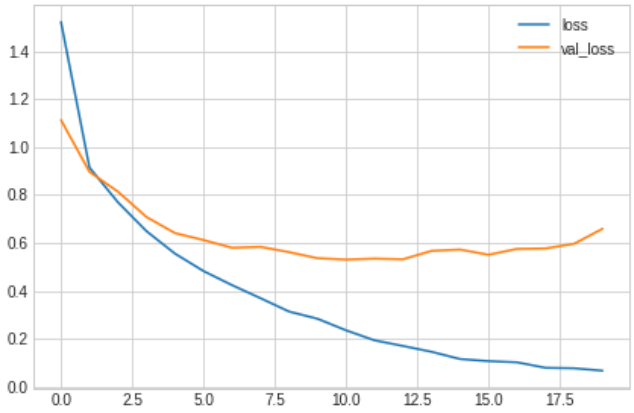
Avem grafice care ne arată cum a evoluat acurateţea pe datele de antrenare şi pe datele de validare incepând de la epoca 0, şi se poate observa ca a rulat pentru ~32 de epoci.

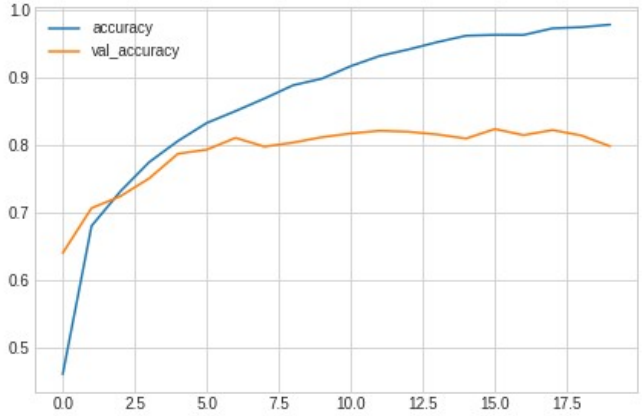
Pentru al 2-lea model am mai adaugat un layer de tipul dense care functionează doar pe date de tip vectorial, deci le-am adăugat după layer-ul de flatten. Am mai adăugat nişte droupout-uri şi am încercat diferite valori pentru numărul de filtre şi de perceptroni. În principal am scăzut 10 şi am verificat dacă soluţia este mai buna sau mai proasta, depinzând de caz, scădeam din nou sau creşteam 10 la valoarea iniţială. Am facut acest proces la câte un singur layer, adică optimizam un singur layer la fiecare rulare. Am încercat si alţi optimizers(RMSprop, adagrad) dar adam a dat cele mai bune rezultate.

Accuracy: 0.9785000085830688
Val accuracy:0.8240000009536743

```
neural_model = keras.Sequential ([
    layers.Input(shape=(32,32,1)),
    layers.Conv2D(20, kernel_size=5, padding="same", activation="relu"),
    layers.MaxPooling2D(pool_size=2, strides=2),
    layers.Dropout(0.2),
    layers.Conv2D(60, kernel_size=5, padding="same", activation="relu"),
    layers.Dropout(0.3),
    layers.MaxPooling2D(pool_size=2, strides=2),
    layers.Flatten(),
    layers.Dense(400, activation='relu'),
    #layers.Dense(84, activation='relu'),
    layers.Dense(9, activation="softmax")
])
```

Această varianta a facut ceva mai mult overfit dar a obţinut un scor de 0.893 pe kaggle (evident antrenat pe toate datele de train).





Se poate vedea că a scăzut la final, dar nu mă afectează deoarece early stopping are parametrul restore_best_weights setat ca true, deci va restaora poziția optimă.

Am mai încercat diverse schimbări ai parametrilor pe tipul acesta de arhitectură (de tipul LeNet-5) dar niciuna nu a fost mai bună decât cea prezentată mai sus. Această încercare a dat exact același rezultat când l-am trimis pe kaggle (0.893):

```
neural_model = keras.Sequential ([
    layers.Input(shape=(32,32,1)),
    layers.Conv2D(25, kernel_size=5, padding="same", activation="relu"),
    layers.MaxPooling2D(pool_size=2, strides=2),
    layers.Dropout(0.2),
    layers.Conv2D(85, kernel_size=5, padding="same", activation="relu"),
    layers.Dropout(0.3),
    layers.MaxPooling2D(pool_size=2, strides=2),
    layers.Flatten(),
    layers.Dense(220, activation='relu'), # + 20 -- revert back, + is usually better
    layers.Dropout(0.3),
    layers.Dense(100, activation='relu'), # - 10 -- - is much better
    layers.Dense(9, activation="softmax")
])
```

Am mai încercat si modele care inițial se bazau pe arhitectura VGG16 dar având imagini de rezoluție mică am fost nevoit să scot layere, și deveniseră nişte modele care erau asemănătoare cu cele prezentate mai sus. De asemenea deşi e recomandat in curs să folosim BatchNormalisation nu am reuşit să îl fac să dea rezultate mai bune, ci dădea mult mai proaste, şi surprinzător, deşi ar trebui sa normalizeze datele, adică să aibe fluctuații mai mici, batch normalisation-ul creştea fluctuațiile foarte mult, iar micşorarea learning rate-ului a micşorat fluctuațiile, dar nu am reuşit să obțin valori mai bune cu acest layer.

Deci dropout si batch normalisation nu ajutau la faptul ca făceam overfit, aşadar am încercat o nouă metodă si anume image augmentation. Dar din păcate când rulam cu un model asemănător cu cele de mai sus avem un underfit extrem de mare, și din cauza aşteptării foarte mari pe valori mai mari/ mai multe layere, adică când măream numarul de parametri, am renunțat la idee. Deci in final am reuşit sa fac un model cu 0.893 acurateţe pe testul de pe kaggle.

Pentru image augmentation am folosit ImageDataGenerator din keras.preprocessing.image și a trebuit sa modific modul în care compilam CNN-ul.

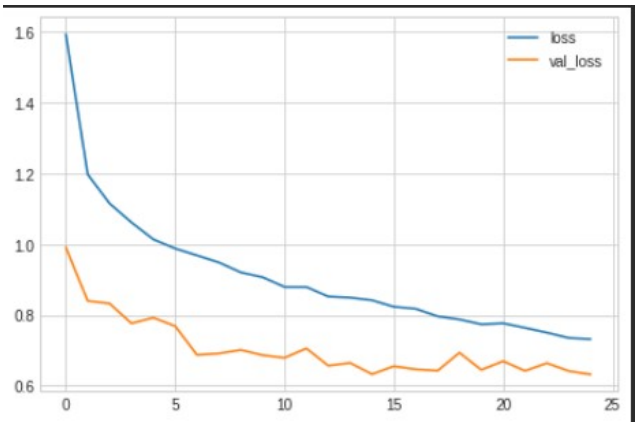
```
datagen = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    #shear_range=0.2,
    zoom_range=0.1,
    horizontal_flip=False,
    vertical_flip=False,
    fill_mode='nearest'
)
X_generator = X_train
X_generator = preprocessor_CNN.transform(X_generator)
X_generator = X_generator.reshape(-1, 32, 32, 1)
y_generator = y_train
y_generator = keras.utils.to_categorical(y_generator, 9)
datagen.fit(X_generator)
generator = datagen.flow(X_generator, y_generator, batch_size=128)
result = neural_model.fit(
    generator,
    steps_per_epoch = X_generator.shape[0] // 128,
    validation_data=(X_valid_CNN, y_valid_CNN),
    epochs = 100,
    #shuffle=True,
    #verbose=False,
    #use_multiprocessing=True,
    callbacks=[early_stopping]
)
```

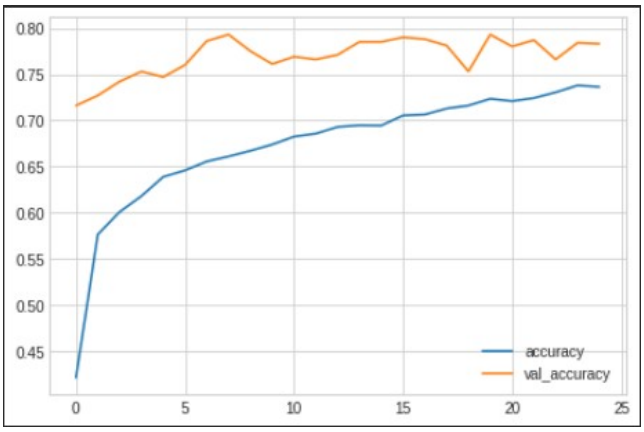
Dar cum am zis lua foarte mult timp la rulare:
Epoch 25/100
234/234 [==============================] - 139s 593ms/step - loss: 0.7303 - accuracy: 0.7329 - val_loss: 0.6319 - val_accuracy: 0.7830 –
2+ minute pe epocă, adică aproximativ o oră, si pe lânga asta era underfit extrem de mult.

Pe modelul următor:

```
neural_model = keras.Sequential ([
    layers.Input(shape=(32,32,1)),
    layers.Conv2D(32, kernel_size=3, activation="relu"),
    layers.Conv2D(32, kernel_size=3, activation="relu"),
    layers.Conv2D(32, kernel_size=5, padding="same", activation="relu"),
    layers.MaxPooling2D(pool_size=2, strides=2),
    layers.Dropout(0.2),
    layers.Conv2D(64, kernel_size=3, activation="relu"),
    layers.Conv2D(64, kernel_size=3, activation="relu"),
    layers.Conv2D(128, kernel_size=5, padding="same", activation="relu"),
    layers.Dropout(0.3),
    layers.MaxPooling2D(pool_size=2, strides=2),
    layers.Flatten(),
```

```
    layers.Dense(1000, activation='relu'), # + 20 -- revert back, + is usually better
    layers.Dropout(0.3),
    layers.Dense(500, activation='relu'), # - 10 -- - is much better
    layers.Dropout(0.1),
    layers.Dense(9, activation="softmax")
])
```





Am încercat să adaug si batch normalisation, deoarece batch normalisation poate să scadă uneori numărul de layere necesare, dar tot timpul a rămas underfitted. Am trimis această antrenare, și a obținut 0.804 pe kaggle, deci ar fi avut potențial sa ajungă la ~0.95.

Deci ca concluzie, cea mai bună antrenare în urma competiției este de 0.893, cu un overfit destul de mare, deci singura soluție era să normalizez cumva imaginile astfel încât să nu mai fie overfitted, dar cu image augmentation consuma prea mult timp.