Multi-class classification with Artificial Neural Networks

Mirko di Gangi, Chiara Sangalli

January 5, 2024

1 Introduction

The aim of our project is to present an analysis of image classification using various neural network architectures, starting from a fundamental model and progressively incorporating advanced features such as dropout layers and convolutional neural networks (CNNs) with data augmentation. When it comes to image classification, neural networks represent a very powerful tool due to their ability of automatically learn hierarchical features from raw pixel data, allowing them to discern complex patterns and representations within images.

All models presented in this study were constructed using the *Sequential* model architecture provided by *TensorFlow*. The *Sequential* model allows for a sequential stacking of layers without the need for complex graph definitions and this facilitates the integration of various neural network components.

2 The dataset Animal-10

2.1 General overview

The dataset we used in this project is *Animal-10*, available online on the website *Kaggle*. This dataset is contains 26179 images, each belonging to different a animal class; there are 10 classes in total, *dog, spider, chicken, horse, butterfly, cow, squirrel, sheep, cat* and *elephant*.

Figure 4 in Appendix B represents a sample of five images from our dataset. The most frequent class is dog, with a total of 4863 images, while the elephant class comes last, with 1446 images. The distribution of the images across the different classes can be found in figure 6 in appendix B.

2.2 Data processing

To prepare our data for the image classification task we manipulated the original dataset: firstly we performed label encoding, to change the classes' labels into integers; this operation is due to the fact that *TensorFlow* requires a numerical representations of categorical variables. Then using different *TensorFlow* functions we were able to read and decode the JPEG image files specified in the dataset; to ensure uniformity in the dataset the images were resized to the minimum width and height present in it and the pixel values compressed to the range [0.1].

After splitting the dataset in training and test sets, we converted them into *Numpy* arrays and subsequently in *TensorFlow* datasets; these are shuffled for variability and batched for efficient training. Now our processed images are ready to be utilized for training neural networks; five of the these images are represented in figure 5 in appendix B.

3 ANN with and without dropout layer

3.1 The first model: a neural network with dense layers

The first architecture we implemented is a fully connected artificial neural network with 5 dense layers; it's configured to take as input images of size (32, 32, 3), where 3 is the number of color channels, and processes them through fully connected layers with decreasing units, concluding with an output layer of 10 units representing the classification scores for different classes.

The activation function implemented in all the layers is the *ReLU* function, a linear function that outputs the input for positive values and 0 if the value is smaller than 0:

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

After defining the model, we compiled it: we chose the *SparseCategoricalCrossentropy* as loss function, the *Adam* algorithm with learning rate equal to 0.0003 as optimizer and *Accuracy* as metric.

SparseCategoricalCrossentropy is employed in multi-class classification problems where each instance belongs to exactly one class and labels are provided as integers, just like our case; it measures the cross-entropy loss between the true labels and the predicted probabilities. We added the argument *from_logits=True* because the *Softmax* function still needs to be applied to transform the output on the model into probabilities.

Adam is the algorithm that we used to update the weights and the bias that connect the different layers of our ANN; it's an adaptive algorithm whose parameter, the learning rate, controls the size of the update in each iteration.

Finally, *Accuracy*, the proportion of correctly classified images, is the metric used to evaluate the performance of the model. All these elements are commonly used in multi-class classification problems.

Eventually, we fit the model with 100 epochs using the images in our train set and we evaluate it on the test set. The growth of the accuracy and the decreasing of the loss are represented in figure 1. As we can see this model presents some problematic aspects: the accuracy of the training phase is 70%, but when it comes to the test set the accuracy drops around 40%; moreover this value is reached around the 25-th epoch and then remains stable, meaning that the our model from such point is not able to improve the predictions. This poor result is confirmed by the convex shape of the validation loss function. Its increasing phase suggests that the model's performance on the validation set is degrading and the cause of that can be found in overfitting: the model is becoming too complex and is fitting the training data too closely, causing a decline in its ability to generalize.

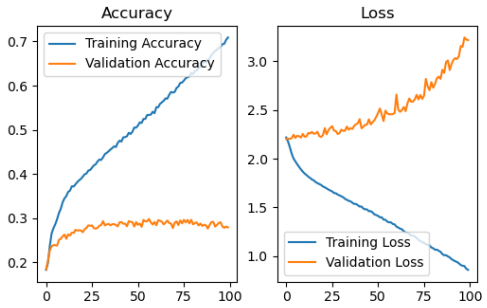

Figure 1: Accuracy and Loss for the train and test set obtained fitting an ANN with dense layers.

3.2 The second model: adding dropout layers

To deal with these problems we added a dropout layer in between every dense layer of the model previously fitted. These new layers randomly "drop out" a 20% of the neurons during training and this should help mitigate the risk of overfitting and improve the model's generalization performance when it comes to new data. The neurons that are deactivated (dropped out) are randomly selected at every iteration and this introduces variability inside the model.

The model was compiled and fitted with the exact same parameters as the previous one, and the results can be seen in figure 2. We can see that this trick improved the shape of the validation loss function, but did not improve the low validation accuracy obtained by the model that remains a little above 40% .

4 Data augmentation and CNN

4.1 Data augmentation process

Data augmentation is a technique used to artificially increase the size of a training dataset by applying various transformations to the existing data, that lead to an improvement in model's

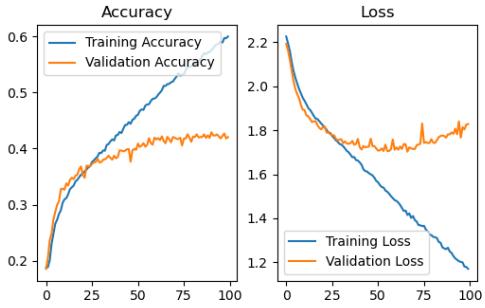

Figure 2: Accuracy and Loss for the train and test set obtained fitting an ANN with dropout layers

generalization. Each observation in the dataset is replicated many times and each replication is distorted in a natural way such that it is not recognisable by an human being; typical distortions involve vertical or horizontal shifts, zoom, horizontal flips or small rotations.

We defined a data transformer to perform transformations on the images of *Animal-10*; these transformations include random rotations (*rotation_range*), horizontal and vertical shifts (*width_shift_range, height_shift_range*), random zooming (*zoom_range*) and others. We then used this transformer to create augmented training set that is ready to become the input of the CNN. A sample of these augmented datasets is represented in figure 7 in appendix B.

4.2 Implementing a CNN

The CNN algorithm is based on the concept of convolution, which is a mathematical operation defined by an integral of two functions after one of them has been reversed and shifted.

$$(f * g)t = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

We built a model that comprises an input layer expecting images of shape (32, 32, 3), followed by a convolutional layer with 64 filters and *padding* = *same* to maintain the spatial dimensions of the output feature maps. The third layer is a batch normalization layer that helps stabilize and speed up the training process, the forth one is a *ReLU* activation, the fifth is a max pooling layer that reduces the spatial dimensions of the representation, helping the model focus on the most important features.

This pattern is repeated a second time and in this case padding is added: now our model flattens our output and feed it to the final fully connected layers, one with 64 neurons and the final one with 10 and activation function *Sigmoid*.

After compiling and fitting the model with the same parameter used before, we can see the results in figure 3: the accuracy of the test set has reached 53% and both the train and validation losses decreases. We must note however that the training accuracy is now lower than the one obtained in the previous two models.

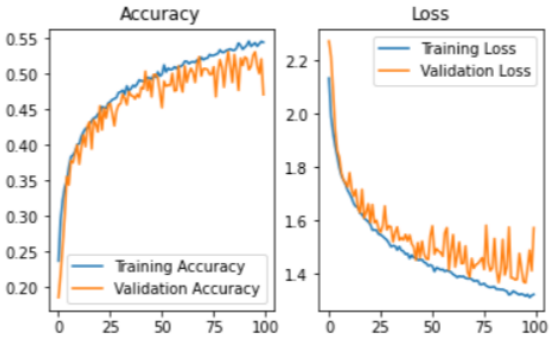

Figure 3: Accuracy and Loss for the train and test set obtained fitting a CNN.

5 Conclusion

In this report we mainly focused on improving the validation accuracy of our model rather than the training one: the reason is that training accuracy measures how well the model fits the training data during the training process. However, this metric may not accurately reflect the model's ability to generalize to new data, as it might simply memorize specific details of the training set without acquiring the ability to handle broader or slightly different data. The ability to generalize is captured by the validation accuracy, the metrics we tried to improve.

The CNN model was the one that retrieved the highest validation accuracy and that minimized in the best way the validation lost function: in both these senses it outperformed the first two models we fitted. However it was not able to reach a significantly high level of accuracy, and this suggests that there might be modifications that could improve the model's performance.

A Appendix A: code

A.1 The dataset

In this appendix we will provide the code that we used in our project; we start with importing all the libraries and our dataset *Animal-10*; note that to replicate this code the **path** must be changed accordingly.

```
# Libraries
import pathlib
import os
import random
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
import matplotlib.pyplot as plt
from PIL import Image
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from tensorflow.keras import Sequential, layers, Input, datasets,
    optimizers
from PIL import Image
import matplotlib.cm as cm
from matplotlib.image import imread
import matplotlib.image as mpimg
from numpy import expand_dims
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator

# Dataset
path = "C:/Users/Lucia/Desktop/statistical␣learning/final/raw-img"

translate = {
    "cane": "dog",
    "cavallo": "horse",
    "elefante": "elephant",
    "farfalla": "butterfly",
    "gallina": "chicken",
    "gatto": "cat",
    "mucca": "cow",
    "pecora": "sheep",
    "scoiattolo": "squirrel",
    "ragno": "spider"
}

data = {"imgpath": [], "labels": []}

for img_class in os.listdir(path):
     folderpath = os.path.join(path, img_class)
     filelist = os.listdir(folderpath)

     english_label = translate.get(img_class, img_class)
     data['labels'] += [english_label] * len(filelist)
     data['imgpath'] += [os.path.join(folderpath, file) for file in
         filelist]

print("Number␣of␣images␣contaned␣in␣the␣dataset:\n", df.shape[0])
print("_____________________________________")
print("Number␣of␣null␣values:\n␣", df.isnull().sum())
print("_____________________________________")
print("Number␣of␣unique␣values:\n", df.nunique())
print("_____________________________________")
counts = df['labels'].value_counts()
print("Number␣of␣images␣for␣every␣category:\n␣", counts)
```

The following code shows the image processing steps that we applied to our original dataset:

```
# Encode string labels to integers using LabelEncoder
label_encoder = LabelEncoder()
data["encoded_labels"] = label_encoder.fit_transform(data["labels"])

# Load images using TensorFlow
image_data = [tf.image.decode_jpeg(tf.io.read_file(img_path), channels=3)
    for img_path in data['imgpath']]
image_data = [tf.image.resize(img, (32, 32)) for img in image_data]
image_data = [tf.cast(img, tf.float32) / 255.0 for img in image_data]

# Ensure all images have the same dimensions
min_width = min(img.shape[0] for img in image_data)
min_height = min(img.shape[1] for img in image_data)
image_data = [tf.image.resize_with_crop_or_pad(img, min_width, min_height)
    for img in image_data]

# Convert labels to NumPy array
label_data = np.array(data["encoded_labels"])

# Split the dataset into train and test set and transforming them into
    numpy array
X_train, X_test, y_train, y_test = train_test_split(image_data, label_data,
     test_size=0.2, random_state=42)
tr_x = np.array(X_train)
tr_y = np.array(y_train)
te_x = np.array(X_test)
te_y = np.array(y_test)

# Define a function that normalizes the input tensor values to the range
    [0, 1] by dividing and converts the label tensor to the integer data
    type
def preprocess(x, y):
  x = tf.cast(x, tf.float32)/255.
  y = tf.cast(y, tf.int32)
  return x, y

# Create TensorFlow datasets
BATCH_SIZE = 128
tr_ds = tf.data.Dataset.from_tensor_slices((tr_x, tr_y))
te_ds = tf.data.Dataset.from_tensor_slices((te_x, te_y))
tr_ds = tr_ds.map(preprocess).shuffle(len(tr_ds)).batch(batch_size =
    BATCH_SIZE)
te_ds = te_ds.map(preprocess).batch(batch_size = BATCH_SIZE)
```

A.2 Artificial Neural Network

We now define, compile and fit the model with dense layers, called **dense**:

```
# Define useful parameters
EPOCH = 100
LEARNING_RATE = 3e-4

# Model
model_dense = Sequential([
  layers.InputLayer(input_shape=(32,32,3)),
  layers.Flatten(),
  layers.Dense(units=512, activation='relu'),
  layers.Dense(units=256, activation='relu'),
  layers.Dense(units=128, activation='relu'),
  layers.Dense(units=32, activation='relu'),
  layers.Dense(10)
])

# Compile
```

```
model_dense.compile(
  loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
  optimizer = optimizers.Adam(learning_rate=LEARNING_RATE),
  metrics = ['accuracy']
)

# Fit
history_dense = model_dense.fit(tr_ds, validation_data=te_ds, epochs=EPOCH,
    verbose=2)
```

And we repeat the process for the model **dropout**:

```
# Model
model_dropout = Sequential([
  layers.InputLayer(input_shape=(32,32,3)),
  layers.Flatten(),
  layers.Dense(units=512, activation='relu'),
  layers.Dropout(0.2),
  layers.Dense(units=256, activation='relu'),
  layers.Dropout(0.2),
  layers.Dense(units=128, activation='relu'),
  layers.Dropout(0.2),
  layers.Dense(units=64, activation='relu'),
  layers.Dropout(0.2),
  layers.Dense(units=34, activation='relu'),
  layers.Dropout(0.2),
  layers.Dense(10)
])

# Compile
model_dropout.compile(
  loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
  optimizer = optimizers.Adam(learning_rate=LEARNING_RATE),
  metrics = ['accuracy']
)

# Fit
history_dropout = model_dropout.fit(tr_ds, validation_data=te_ds, epochs=
    EPOCH, verbose=2)
```

For both models we define store the values of training and validation accuracy and loss:

```
# Dense model
acc_dense = history_dense.history['accuracy']
val_acc_dense = history_dense.history['val_accuracy']
loss_dense = history_dense.history['loss']
val_loss_dense = history_dense.history['val_loss']

# Dropout model
acc_dropout = history_dropout.history['accuracy']
val_acc_dropout = history_dropout.history['val_accuracy']
loss_dropout = history_dropout.history['loss']
val_loss_dropout = history_dropout.history['val_loss']
```

A.3 Convolutional Neural Network

First we perform data augmentation:

```
# Creating an ImageDataGenerator with data augmentation settings
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.3,
    height_shift_range=0.4,
    shear_range=0.1,
    zoom_range=0.4,
    horizontal_flip=True,
    fill_mode='nearest'
```

```
)

# Fit the data generator on the training data
datagen.fit(X_train)

# Creating the augmented training and test sets:
tr_aug = datagen.flow(tr_x,tr_y,batch_size=128)
te_aug = datagen.flow(te_x,te_y,batch_size=128)
```

We can now define and implement the CNN model, called **conv_model**:

```
# Model
  conv_model = Sequential([
  layers.InputLayer(input_shape=(32,32,3)),
  layers.Conv2D(64, kernel_size=3, padding='same'),
  layers.BatchNormalization(),
  layers.ReLU(),
  layers.MaxPooling2D(),
  layers.Conv2D(128, kernel_size=3, padding='valid'),
  layers.BatchNormalization(),
  layers.ReLU(),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(units=64, activation='relu'),
  layers.Dense(10, activation="sigmoid")
])

# Compile
conv_model.compile(
  loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
  optimizer = optimizers.Adam(learning_rate=0.0001),
  metrics = ['accuracy']
)

# Fit
conv_fit=conv_model.fit(tr_aug, validation_data=te_aug, epochs=100,
    batch_size=128)
```

We then store the resulting values of accuracy and loss:

```
acc_conv =conv_fit.history['accuracy']
val_acc_conv = conv_fit.history['val_accuracy']
loss_conv = conv_fit.history['loss']
val_loss_conv = conv_fit.history['val_loss']
```

B Appendix B: figures

Figure 4 and 5 represents five images from the dataset *Animals-10*: figure 5 showscases the original version of the images, while figure 6 shows the same five images after data manipulation. In the second pictures we can clearly notice the pixel in which we decomposed our images, as well as the fact that now every picture has the same dimension.

Figure 4: Original images.

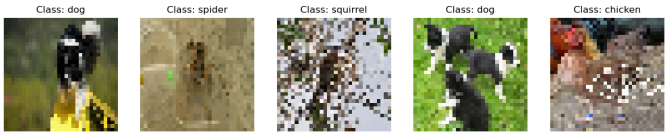

Figure 5: Processed images

Figure 6 shows the distribution of the images in *Animals-10* between the different classes.

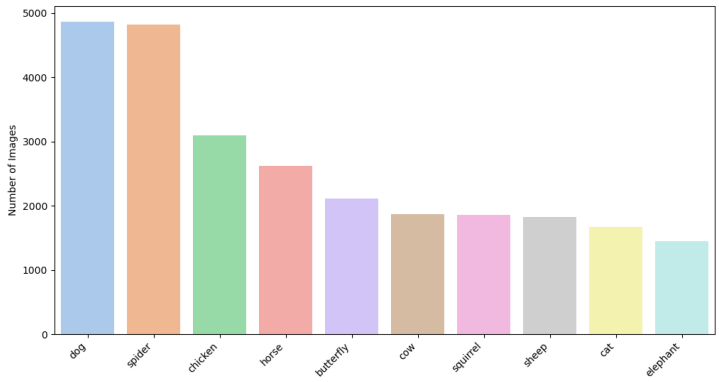

Figure 6: Image distribution

Figure 7 shows the images after the data augmentation manipulation

Figure 7: Post-augmentation images.

In the following, we provide the code used to create all the images included in the report and in this appendix:

```
plt.figure(figsize=(6, 3.5))

plt.subplot(1, 2, 1)
plt.plot(range(EPOCH), acc_dense, label='Training Accuracy')
plt.plot(range(EPOCH), val_acc_dense, label='Validation Accuracy')
plt.legend(loc='upper left')
plt.title('Accuracy')

plt.subplot(1, 2, 2)
plt.plot(range(EPOCH), loss_dense, label='Training Loss')
plt.plot(range(EPOCH), val_loss_dense, label='Validation Loss')
plt.legend(loc='lower left')
plt.title('Loss')

plt.savefig('f1.png')
```

Listing 1: figure 1 code.

```
plt.figure(figsize=(6, 3.5))

plt.subplot(1, 2, 1)
plt.plot(range(EPOCH), acc_dropout, label='Training Accuracy')
plt.plot(range(EPOCH), val_acc_dropout, label='Validation Accuracy')
plt.legend(loc='upper left')
plt.title('Accuracy')

plt.subplot(1, 2, 2)
plt.plot(range(EPOCH), loss_dropout, label='Training Loss')
plt.plot(range(EPOCH), val_loss_dropout, label='Validation Loss')
plt.legend(loc='lower left')
plt.title('Loss')

plt.savefig('f2.png')
```

Listing 2: figure 2 code.

```
plt.figure(figsize=(7, 8))

plt.subplot(1, 2, 1)
plt.plot(range(EPOCH), acc_conv, label='Training Accuracy')
plt.plot(range(EPOCH), val_acc_conv, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Accuracy')

plt.subplot(1, 2, 2)
plt.plot(range(EPOCH), loss_conv, label='Training Loss')
plt.plot(range(EPOCH), val_loss_conv, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Loss')

plt.savefig('f3.png')
```

Listing 3: figure 3 code.

```
sample random.sample(range(len(data['imgpath'])),5)

fig, axes = plt.subplots(1, 5, figsize=(15, 4))

for i, idx in enumerate(sample):
    img_path = data['imgpath'][idx]
    img = Image.open(img_path)
    axes[i].imshow(img)
    axes[i].set_title(f'Class: {data["labels"][idx]}')
    axes[i].axis('off')

plt.savefig('fB1.png')
```

Listing 4: figure 4 code.

```
fig, axes = plt.subplots(1, 5, figsize=(15, 4))

for i, idx in enumerate(sample):
    img = image_data[idx].numpy()
    label = label_encoder.inverse_transform([label_data[idx]])[0]
    axes[i].imshow(img)
    axes[i].set_title(f'Class: {label}')
    axes[i].axis('off')

plt.savefig('fB2.png')
```

Listing 5: figure 5 code.

```
df = pd.DataFrame(data)

plt.figure(figsize=(12, 6))
sns.countplot(data=df, x='labels', order=df['labels'].value_counts().index,
    palette='pastel')
plt.xlabel('Class')
plt.ylabel('Number of Images')
plt.xticks(rotation=45, ha='right')

plt.savefig('fB3.png')
```

Listing 6: figure 6 code.

```
augmented_images = []
num_images_to_display = 5

for i in range(num_images_to_display):
    # Extraction of reconstructed images
    batch = next(tr_aug)
    augmented_image = batch[0][0]
    augmented_images.append(augmented_image)

# Showing transformed images
plt.figure(figsize=(10, 5))
for i in range(num_images_to_display):
    plt.subplot(1, num_images_to_display, i + 1)
    plt.imshow(augmented_images[i])
    plt.axis('off')

plt.show()
```

Listing 7: figure 7 code.