Challenge: AlexNet model from scratch and pretrained in Keras/TensorFlow and PyTorch.

Carolina Bernal Rodríguez *(Keras/TF)*. Colaborator: **Zoé Ariel García Martínez** *(PyTorch)*.

Due date: Fri, March 7th 2025.

The challenge of this exercise consisted of training the AlexNet model from scratch in both TensorFlow and PyTorch using CIFAR10 data, and also comparing the performance between a model trained from scratch and a pre-trained model of AlexNet.

1 Motivation

AlexNet is a convolutional neural network (CNN) architecture, designed by ***Alex Krizhevsky*** in collaboration with *Ilya Sutskever* and *Geoffrey Hinton*. The model consists of 8 layers: 5 convolutional layers followed by 3 fully connected linear layers. To produce the 1000-label classification needed for ImageNet, the final layer used a 1000-node softmax, creating a probability distribution over the 1000 classes.

The original paper's primary result was that the depth of the model was essential for its high performance, which was computationally expensive, but made feasible due to the utilization of graphics processing units (GPUs) during training. The training time of the model was reduced further by swapping the standard sigmoid or tanh activation functions of the time for *Rectified Linear Unit (ReLU)* activation functions.

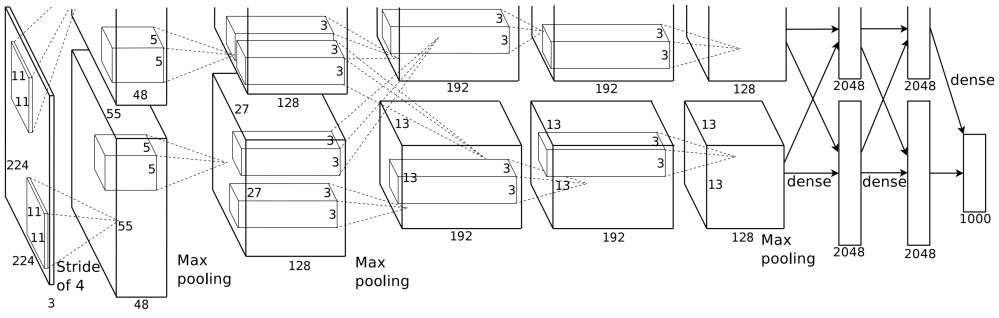


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

2 TensorFlow/Keras AlexNet model.

Prerequisites for running:
This code was run on a MacBook Pro(M1 Pro) with 10 cores, 10 threads, 2 efficiency cores and 8 performance cores, 16 GB RAM. A virtual environment was created `tensorflow_env` to be used for both TF and PyTorch. On the repository I shared the *requirements.txt* file for the versions of the packages installed. The models were run using GPU. Models were trained using CIFAR-10.

2.1 TF/Keras: AlexNet model from scratch.

To successfully run the model from scratch we need to code the model architecture as a Keras Sequential model. We also need to download the *CIFAR-10* data available in the `tensorflow.keras.datasets` using `(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()` , Also CIFAR-10 data consists on 10 classes defined by:

```
class_names = ['airplane', 'automobile', 'bird', 'cat',
'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```

We also need to create our ***testing, training a validation sets,*** on the code you'll see that we define some useful functions to achieve that like `resize_image(image, image_height=224, image_width=224)` that reshape the image since AlexNet model needs 224x224 images while CIFAR-10 consists on 32x32 images. We also defined `preprocess_image(image, label)` to normalize the image to [0,1] and also to apply one hot encoding on the labels. We define first the training and testing starting with using `tf.data.Dataset.from_tensor_slices` method:

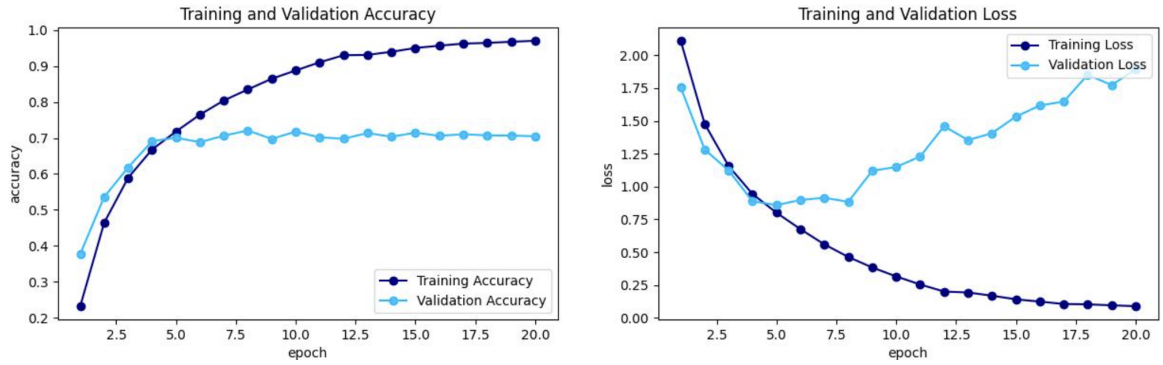
```
train_ds = tf.data.Dataset.from_tensor_slices((train_images,train_labels))
test_ds  = tf.data.Dataset.from_tensor_slices((test_images,test_labels))
```

This method takes the train, test, and validation dataset partitions and returns a corresponding TensorFlow Dataset representation, we will also split 10% to create the validation set.Here is the example of how we defined the training data set, testing and validation were defined following that structure as well:

```
## Training set
train_ds = (
    train_ds
    .map(preprocess_image, num_parallel_calls=tf.data.AUTOTUNE)
    .shuffle(buffer_size=test_ds_size) # suffle the data to prevent bias
    .batch(batch_size,drop_remainder=True) # dismiss last batch if small
    .prefetch(tf.data.AUTOTUNE) # TF adjust automatically pre-loaded data
    ## Prefetch is used to optimize data access by loading data into memory
    ## before it is actually needed.
)
```

Once we have configured all that we can use the same structure with the Augmentation data and also use it to train again the pre-trained model. The code for the AlexNet architecture can be found in the `build_AlexNet` function.
The model was compiled and trained with the `train_model` function, for data Augmentation run it for 15 epochs while AlexNet from scratch with no data augmentation was run with 20 epochs.
One mistake I made while running was when setting the model architecture was instead to add a Dense layer with 1000 units , with that the model had `Total params: 202,839,952 (773.77 MB)` and took over 4hr to run 20 epochs, since took 8 min on average per epochs!! On that model I didn't include Dropout resulting in overfitting as well, the results were model evaluate `test_ds - accuracy: 0.9738 - loss: 0.0753` *[0.06679382920265198, 0.9774082899093628]* . The image below shows the graphs with the 1000 layer add and no Dropout layers resulting in model overfitting.

```
## Add an extra dense layer of 1000 units
model.add(keras.layers.Dense(units=1000))  # Dense , units= 1000
# output layer is softmax
model.add(keras.layers.Dense(units=class_count,activation='softmax'))

## Correct architecture
model.add(keras.layers.Dense(units=class_count))  # Dense, class_count =10
# output layer is softmax
model.add(keras.layers.Activation('softmax'))
```



(a) Output graphs for the AlexNet "big model" since there is no Dropout layer , we saw validation loss increasing as a result of overfitting.

Layer (type)	Output Shape	Param #
conv2d_25 (Conv2D)	(None, 56, 56, 96)	34,944
max_pooling2d_15 (MaxPooling2D)	(None, 27, 27, 96)	0
conv2d_26 (Conv2D)	(None, 27, 27, 256)	614,656
max_pooling2d_16 (MaxPooling2D)	(None, 13, 13, 256)	0
conv2d_27 (Conv2D)	(None, 13, 13, 384)	885,120
conv2d_28 (Conv2D)	(None, 13, 13, 384)	1,327,488
conv2d_29 (Conv2D)	(None, 13, 13, 256)	884,992
max_pooling2d_17 (MaxPooling2D)	(None, 6, 6, 256)	0
flatten_5 (Flatten)	(None, 9216)	0
dense_20 (Dense)	(None, 4096)	37,752,832
dropout (Dropout)	(None, 4096)	0
dense_21 (Dense)	(None, 4096)	16,781,312
dropout_1 (Dropout)	(None, 4096)	0
dense_22 (Dense)	(None, 10)	40,970
activation (Activation)	(None, 10)	0

Total params: 58,322,314 (222.48 MB)

Trainable params: 58,322,314 (222.48 MB)

Non-trainable params: 0 (0.00 B)

(b) AlexNet correct model architecture in Keras

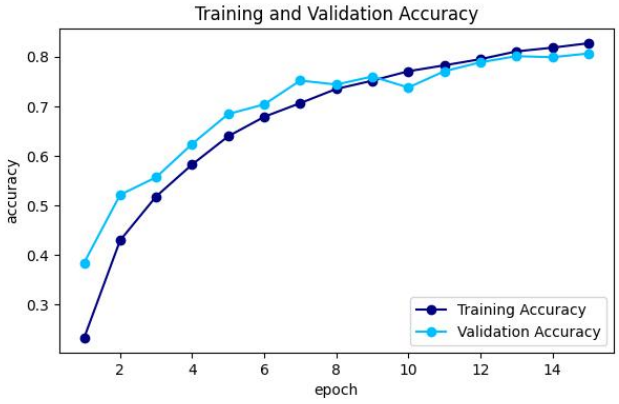
```python
def train_model(epochs=20):

  ##train AlexNet model

  model = build_AlexNet(class_count=10)
  # display model info
  model.build((None, 224, 224, 3))
  model.summary()

  init_lr = 0.01
  decay_steps = 10000
  decay_rate = 0.96
  lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(initial_learning_rate=init_lr, decay_steps=decay_steps
                                                               , decay_rate=decay_rate)

  optimizer = SGD(learning_rate=lr_schedule, momentum=0.9)
  model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
  history = model.fit(
      x=train_ds,
      validation_data=valid_ds,
      epochs=epochs,
  )
  return history, model

# Train model
history, model = train_model(epochs=15)
```

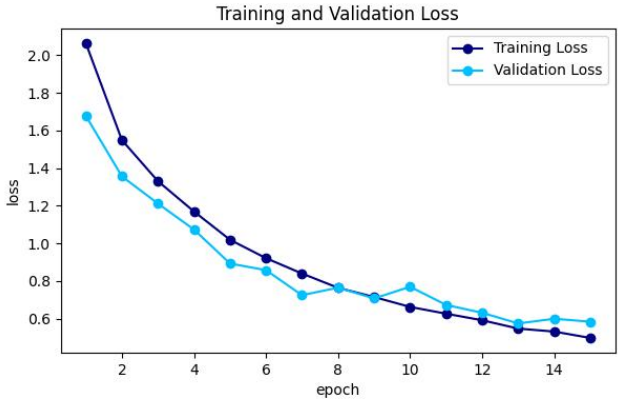
Python

(c) Function for training using SGD as optimizer

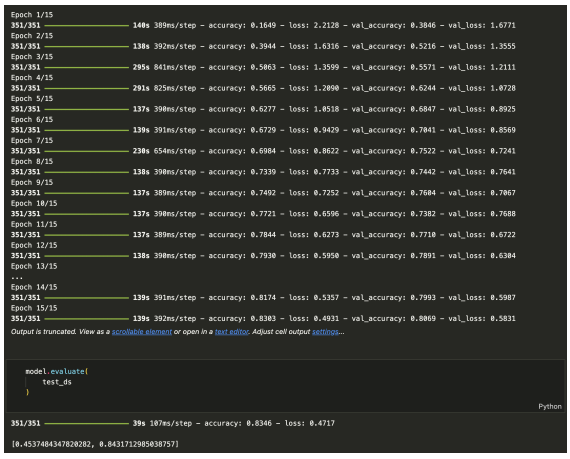
2.2 TF/Keras: AlexNet model from scratch using Augmentation data.

We will use the same code from above and for `data_augmentation` we will add a function that help us to include random flips and crops to the image, this function will be used on the training dataset to substitute `preprocess_image`, that gave us an accuracy above 80%, both adding the Dropout layer and Data Augmentation helped with the performance. For data augmentation decided to use 15 epochs and a 128 Batch.

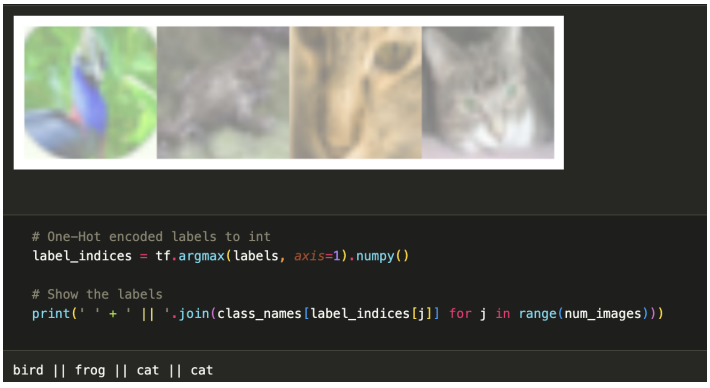




(a) Plots. Training AlexNet Augmentation 15 epoch 128 Batch.



(b) build AlexNet function model performance



(c) Output images and predictions from the model in Keras.

Using data augmentation helped us for the model to prevent overfitting, therefore, both the accuracy and the loss performed better as well. Here are some examples of image outputs and performance.

2.3 TF/Keras: AlexNet model from scratch using PyTorch pretrained model.

On this step I tried to used pretrained weights of PyTorch available model to transfer those weights to Keras, to achieve that each layer in the PyTorch model should match the corresponding layer of the Keras model where for example `"features.0.weight": "conv2d_tf_1"` and `"classifier.6.weight": "dense_tf_3"` you also need for the Dense layers to Transpose and Conv need `.transpose(2, 3, 1, 0)`. Spoiler the performance was not so great we trained the model using fine-tunning by frozen the last layer and train and then de frozen all the layers and train again.

```
Transformation of features.0.weight to conv2d_tf_1 (Conv2D)
✔ Weights successfully assigned to conv2d_tf_1 from features.0.weight
✔ Weights successfully assigned to conv2d_tf_1 from features.0.bias
Transformation of features.3.weight to conv2d_tf_2 (Conv2D)
✔ Weights successfully assigned to conv2d_tf_2 from features.3.weight
✔ Weights successfully assigned to conv2d_tf_2 from features.3.bias
Transformation of features.6.weight to conv2d_tf_3 (Conv2D)
✔ Weights successfully assigned to conv2d_tf_3 from features.6.weight
✔ Weights successfully assigned to conv2d_tf_3 from features.6.bias
Transformation of features.8.weight to conv2d_tf_4 (Conv2D)
✔ Weights successfully assigned to conv2d_tf_4 from features.8.weight
✔ Weights successfully assigned to conv2d_tf_4 from features.8.bias
Transformation of features.10.weight to conv2d_tf_5 (Conv2D)
✔ Weights successfully assigned to conv2d_tf_5 from features.10.weight
✔ Weights successfully assigned to conv2d_tf_5 from features.10.bias
Transformation of classifier.1.weight to dense_tf_1 (Dense)
✔ Weights successfully assigned to dense_tf_1 from classifier.1.weight
Transformation of classifier.1.bias to dense_tf_1 (Dense)
✔ Weights successfully assigned to dense_tf_1 from classifier.1.bias
Transformation of classifier.4.weight to dense_tf_2 (Dense)
✔ Weights successfully assigned to dense_tf_2 from classifier.4.weight
Transformation of classifier.4.bias to dense_tf_2 (Dense)
✔ Weights successfully assigned to dense_tf_2 from classifier.4.bias
Transformation of classifier.6.weight to dense_tf_3 (Dense)
✔ Weights successfully assigned to dense_tf_3 from classifier.6.weight
Transformation of classifier.6.bias to dense_tf_3 (Dense)
✔ Weights successfully assigned to dense_tf_3 from classifier.6.bias
```

```
351/351 ━━━━━━━━━━━━━━━━━━━━ 95s 269ms/step - accuracy: 0.3048 - loss: 530.4138 - val_accuracy: 0.4912 - val_loss: 517.4256
Epoch 3/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 95s 269ms/step - accuracy: 0.3148 - loss: 2100.3696 - val_accuracy: 0.5459 - val_loss: 645.3294
Epoch 4/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 96s 271ms/step - accuracy: 0.3569 - loss: 2212.8298 - val_accuracy: 0.5577 - val_loss: 764.5931
Epoch 5/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 96s 268ms/step - accuracy: 0.3809 - loss: 2184.4048 - val_accuracy: 0.5693 - val_loss: 768.8715
Epoch 6/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 96s 269ms/step - accuracy: 0.4061 - loss: 2087.3032 - val_accuracy: 0.5931 - val_loss: 698.9427
Epoch 7/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 96s 271ms/step - accuracy: 0.4389 - loss: 1819.7129 - val_accuracy: 0.6024 - val_loss: 684.1219
Epoch 8/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 95s 268ms/step - accuracy: 0.4485 - loss: 1714.0049 - val_accuracy: 0.6200 - val_loss: 650.3654
Epoch 9/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 97s 272ms/step - accuracy: 0.4559 - loss: 1638.5991 - val_accuracy: 0.6334 - val_loss: 593.3567
Epoch 10/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 97s 271ms/step - accuracy: 0.4759 - loss: 1466.4913 - val_accuracy: 0.6546 - val_loss: 540.1196
Epoch 11/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 97s 270ms/step - accuracy: 0.4914 - loss: 1400.7213 - val_accuracy: 0.6440 - val_loss: 565.9611
Epoch 12/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 96s 270ms/step - accuracy: 0.5053 - loss: 1308.7759 - val_accuracy: 0.6683 - val_loss: 525.5766
Epoch 13/20
...
Epoch 19/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 96s 268ms/step - accuracy: 0.5700 - loss: 923.0313 - val_accuracy: 0.7181 - val_loss: 366.7787
Epoch 20/20
351/351 ━━━━━━━━━━━━━━━━━━━━ 533s 2s/step - accuracy: 0.5754 - loss: 901.6725 - val_accuracy: 0.7113 - val_loss: 389.3192
```

3 PyTorch AlexNet model

3.1 PyTorch AlexNet model from scratch.

We used the documentation `https://github.com/pytorch/vision/blob/main/torchvision/models/alexnet.py` to replicate the model architecture. We used `criterion = nn.CrossEntropyLoss()`, and a batch size of 64. We'll use 20 epochs to train our model, we also use the same optimizer we used for Keras:

`optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)` Using `DataLoader` the training and testing sets were created as it wraps an iterable around the Dataset to enable easy access to the samples. AlexNet expects input images to be tensors of (3XHXW) dimensions where 3 represents the 3 color channels RGB , H is height and W is Width of the images (224x224), so we must resize our images as well.
The performance of the model was the following Accuracy of the network on the 10000 test images: 74.00 % . With :
Accuracy of plane : 83.93 % ; Accuracy of car : 80.00 % ; Accuracy of bird : 68.35 % ; Accuracy of cat : 54.79 % ; Accuracy of deer : 78.18 % ; Accuracy of dog : 64.41 % ; Accuracy of frog : 82.14 % ; Accuracy of horse : 68.75 % ; Accuracy of ship : 81.03 % ; Accuracy of truck : 78.21 % ;
Also PyTorch model architecture available is an improved version from the original with faster and improved performance

3.2 PyTorch AlexNet model pretrained.

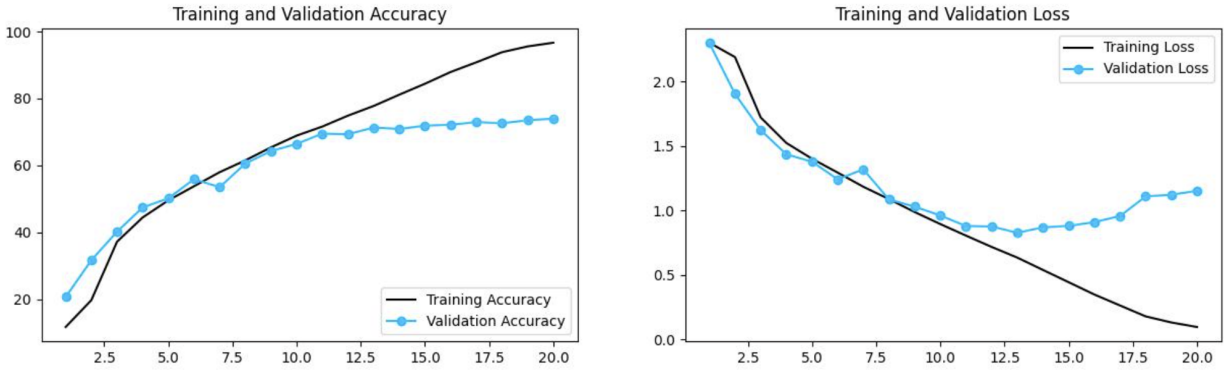
In this case Pytorch does have a pretrained model available so the configuration is easy since we also have the code done to train from scratch so the step by step is straightforward. As expected for this model we get a better accuracy than the model from scratch with **Accuracy of the network on the 10000 test images: 91.77 %**. This make since since the model has already info from the previous training. That's also the reason

```
AlexNet(
  (cn1): Conv2d(3, 96, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
  (cn2): Conv2d(96, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (cn3): Conv2d(256, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (cn4): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (cn5): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=9216, out_features=4096, bias=True)
  (dropout1): Dropout(p=0.5, inplace=False)
  (fc2): Linear(in_features=4096, out_features=4096, bias=True)
  (dropout2): Dropout(p=0.5, inplace=False)
  (fc3): Linear(in_features=4096, out_features=10, bias=True)
)
```

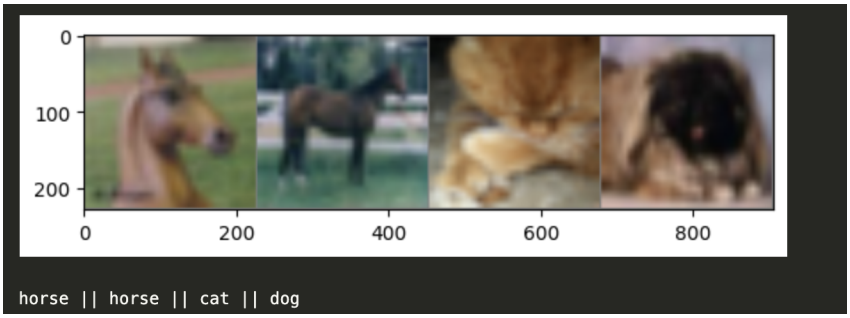
(a) AlexNet architecture in PyTorch.

```
Epoch [1/20], Loss: 2.2969, Accuracy: 11.74%
Epoch [2/20], Loss: 1.6175, Accuracy: 19.76%
Epoch [3/20], Loss: 1.3504, Accuracy: 37.21%
Epoch [4/20], Loss: 1.3190, Accuracy: 44.47%
Epoch [5/20], Loss: 1.1009, Accuracy: 49.58%
Epoch [6/20], Loss: 1.1522, Accuracy: 53.80%
Epoch [7/20], Loss: 1.2300, Accuracy: 58.00%
Epoch [8/20], Loss: 1.0196, Accuracy: 61.52%
Epoch [9/20], Loss: 0.8218, Accuracy: 65.42%
Epoch [10/20], Loss: 0.7776, Accuracy: 68.92%
Epoch [11/20], Loss: 1.0306, Accuracy: 71.63%
Epoch [12/20], Loss: 0.9408, Accuracy: 74.89%
Epoch [13/20], Loss: 1.0432, Accuracy: 77.79%
Epoch [14/20], Loss: 1.0249, Accuracy: 81.16%
Epoch [15/20], Loss: 0.8457, Accuracy: 84.43%
Epoch [16/20], Loss: 0.6578, Accuracy: 87.93%
Epoch [17/20], Loss: 0.7642, Accuracy: 90.82%
Epoch [18/20], Loss: 1.2038, Accuracy: 93.85%
Epoch [19/20], Loss: 1.3959, Accuracy: 95.58%
Epoch [20/20], Loss: 1.1862, Accuracy: 96.69%
Finished Training
```

(b) build AlexNet function model performance PyTorch



(c) Output images and predictions from the model in PyTorch from scratch.



(d) Images from PyTorch model.

why the model starts with a high accuracy but i reality as we move forward on the training the validation accuracy flatten around 91 %

4 Lessons Learned from AlexNet in Keras and PyTorch

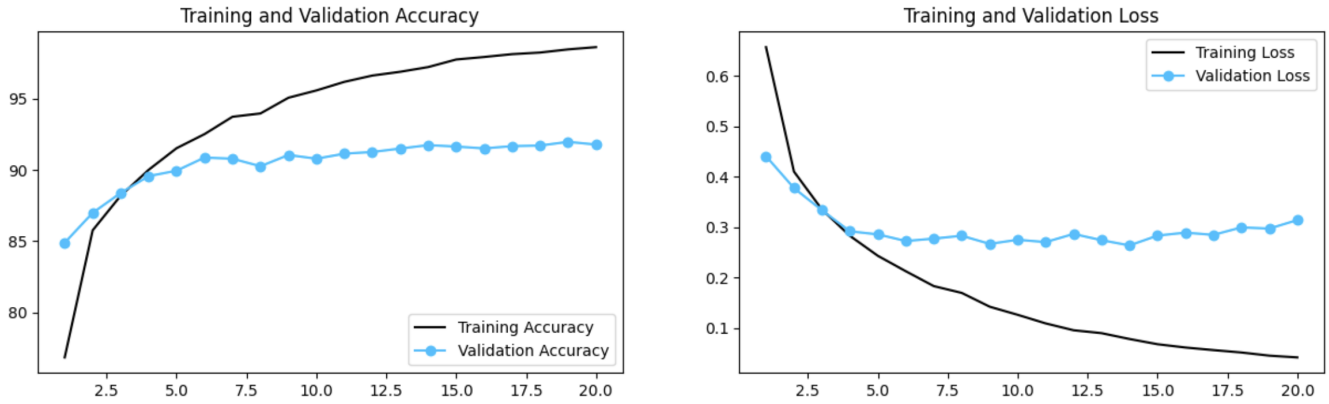
- Learnings from errors like using an extra layer of Dense 1000 the model performance in terms of accuracy is not so great since accuracy: 0.9745 loss: 0.0782 - val accuracy: 0.7045 - val loss: 1.8938 on the final epoch and since the model do not have a Dropout layer, then the model output is overfitting since the validation loss increases from around epoch 8 while the training loss decreases. This error was fixed by adding the dropout layer and also with data augmentation
- Learned a small hack in VSCode to download .ipynb from the interactive terminal since at the beginning I was only able to configure the environment on the terminal, but then I successfully include the venv on the .ipynb kernel, downloading my outputs from the terminal as a .ipynb helped me a lot to not loss those outputs (those

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

(a) Pretrained Pytorch AlexNet Architecture.

```
Accuracy of plane : 96.43 %
Accuracy of car   : 100.00 %
Accuracy of bird  : 91.14 %
Accuracy of cat   : 76.71 %
Accuracy of deer  : 96.36 %
Accuracy of dog   : 88.14 %
Accuracy of frog  : 94.64 %
Accuracy of horse : 89.06 %
Accuracy of ship  : 100.00 %
Accuracy of truck : 93.59 %
```

(b) build AlexNet function model performance for pretraines model



(c) Output images and predictions from the model in Keras.

were the ones form the Keras model which took 8min per epoch to run.) Data Augmentation with Keras took around 2 min per epoch , AlexNet in PyTorch took 65 min in total and pretrained in PyTorch took 90 min in total.

- Keras/Tensorflow doesn't have a pretrained model for AlexNet like PyTorch does, so to achieve a pretrained model we need to transfer learning and it is crucial to successfully convert the weights for PyTorch model to Keras, this is a difficult task since the architecture of the model is different between them and also the weights need a correct conversion to achieve good results (names of the layers are different and if you don't fix a name of the Keras model layers, this may change on each run) also noticed that we need to change the image input normalization to also match the formats and transpose to match TF format since it's different from pyTorch and the transformation depends if the layer is either Dense (Fully connected) or convolutional.
- Since the weights from AlexNet pretrained model are based on ImageNet, when converting to Keras, it is crucial to also transform the input images using mean and std of RGB.
- Performance in Keras using data augmentation was better than without it , but the performance of pretrained AlexNet with Pyorch was the best from all with an acurracy of 91% . - As the references mention The AlexNet architecture dominated in 2012 by achieving a top-5 error rate of 15.3%, significantly lower than the runner-up's 26.2%. This large reduction in error rate excited the researchers with the potential of deep neural networks in handling large image datasets. Subsequently, various Deep Learning models were developed later.
- Special thanks to Zoé Ariel García Martínez who guided me with the pyTorch model, we worked together learning from the documentation and testing the code. We learned also on TF Keras since I tried using pyTorch model while he tested with Toronto weights.

5 References

https://github.com/camachojua/diplomado-ia
https://lamaquinaoraculo.com/deep-learning/alexnet/
https://d2l.ai/chapter_convolutional-modern/alexnet.html
https://www.pinecone.io/learn/series/image-search/imagenet/
https://developer.apple.com/metal/tensorflow-plugin/
https://github.com/bznick98/Learn-AlexNet-Keras/blob/master/AlexNet.ipynb
https://github.com/notem/keras-alexnet/blob/master/alexnet.py
https://medium.com/@siddheshb008/alexnet-architecture-explained-b6240c528bd5
https://tejasmohanayyar.medium.com/a-practical-experiment-for-comparing-lenet-alexnet-vgg-and-resnet-models-with-their-advantages-d932fb7c7d17
https://viso.ai/deep-learning/alexnet/
https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pd
https://www.cs.toronto.edu/ kriz/cifar.html