HW3: CIFAR10: Convolutional Neural Networks

In this homework assignment, we'll be focusing on all things CNN!

In [1]:
```
!python3 -VV
```

```
Python 3.9.13 | packaged by conda-forge | (main, May 27 2022, 17:00:33)
[Clang 13.0.1 ]
```

If you are running the notebook on Colab, you need to mount your drive. You can then add the directory where you python code is to the system path. The change in the system path is valid only for this session. If you are running the notebook in your local machine, the boolean variable `isColab` is going to be `False`, so anything inside the if statement will be ignored.

Package Imports

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import tensorflow as tf
import tensorflow_datasets as tfds
```

Code Imports

In [3]:
```
%load_ext autoreload
%autoreload 2
import   assignment, conv_model, layers_keras, layers_manual
%aimport assignment, conv_model, layers_keras, layers_manual
```

Data Pathing

Data Preprocessing: CIFAR10

This code block will help you get familiar with the shape and type of the data returned by `get_data()`. `get_data` returns X0 (training images), X1 (training labels), Y0 (testing images), Y1 (testing labels), and some additional info about the dataset.

In [4]:
```
data = assignment.get_data()
X0, Y0, X1, Y1, D0, D1, D_info = data
```

```
Metal device set to: Apple M1

2022-10-26 23:24:34.111729: I tensorflow/core/common_runtime/pluggable_de
vice/pluggable_device_factory.cc:305] Could not identify NUMA node of pla
tform GPU ID 0, defaulting to 0. Your kernel may not have been built with
NUMA support.
2022-10-26 23:24:34.111816: I tensorflow/core/common_runtime/pluggable_de
vice/pluggable_device_factory.cc:271] Created TensorFlow device (/job:loc
alhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical Plugga
bleDevice (device: 0, name: METAL, pci bus id: <undefined>)
2022-10-26 23:24:34.152515: W tensorflow/core/platform/profile_utils/cpu_
utils.cc:128] Failed to get CPU frequency: 0 Hz
```

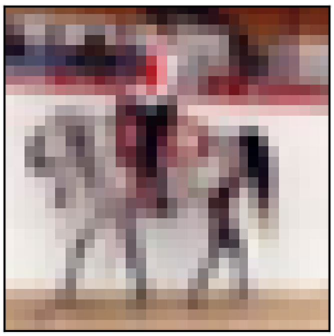
In [5]:
```
D_info
```

Out[5]:
```
tfds.core.DatasetInfo(
    name='cifar10',
    full_name='cifar10/3.0.2',
    description="""
    The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 clas
ses, with 6000 images per class. There are 50000 training images and 1000
0 test images.
    """,
    homepage='https://www.cs.toronto.edu/~kriz/cifar.html',
    data_path='/Users/henrydonahue/tensorflow_datasets/cifar10/3.0.2',
    download_size=162.17 MiB,
    dataset_size=132.40 MiB,
    features=FeaturesDict({
        'id': Text(shape=(), dtype=tf.string),
        'image': Image(shape=(32, 32, 3), dtype=tf.uint8),
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=10),
    }),
    supervised_keys=('image', 'label'),
    disable_shuffling=False,
    splits={
        'test': <SplitInfo num_examples=10000, num_shards=1>,
        'train': <SplitInfo num_examples=50000, num_shards=1>,
    },
    citation="""@TECHREPORT{Krizhevsky09learningmultiple,
        author = {Alex Krizhevsky},
        title = {Learning multiple layers of features from tiny images},
        institution = {},
        year = {2009}
    }""",
)
```

In [6]:
```
D_info.features['label']._int2str
```

Out[6]:
```
['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']
```

In [7]:
```
tfds.show_examples(D0, D_info);
```


horse (7)


ship (8)

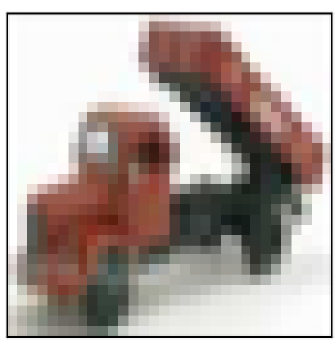

deer (4)


deer (4)


frog (6)


dog (5)


bird (2)


truck (9)


frog (6)

Augmenting train data

Once you've completed **[TODO 1]** in `get_default_CNN_model()` , you can run the cell below to visualize what your data augmentation pipeline is doing to the images. This will also hopefully help you determine which augmentations may help your model generalize and which will increase performance!

- First row shows original images but scaled
- Second row shows images after they have been preprocessed
- Third row shows your augmented images.

NOTE: You do not need to finish TODO 2 before running this cell

In [8]:
```
import conv_model

## You can use any list of 10 indices
sample_image_indices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sample_images = tf.cast(tf.gather(X0, sample_image_indices), tf.float32)
sample_labels = tf.gather(Y0, sample_image_indices)

fig, ax = plt.subplots(3, 10)
fig.set_size_inches(24, 8)

args = conv_model.get_default_CNN_model()

preprocessed_images = args.model.input_prep_fn(sample_images)
augmented_images = args.model.augment_fn(preprocessed_images)

for i in range(10):
    ax[0][i].imshow(sample_images[i]/255., cmap = "Greys")
    ax[1][i].imshow(preprocessed_images[i], cmap = "Greys")
    ax[2][i].imshow(augmented_images[i], cmap = "Greys")
```

Train A Basic Keras Model

As part of step 2 from the handout, we just want you to construct a simple keras model to run prediction on your dataset!

Implement **[TODO 2]** in `get_default_CNN_model` to return a CNN model that can train above an accuracy of 55% (note that the requirement for 1470 is 62% and for 2470, is 65% though). Feel free to play around with the number of layers, hyperparameters for layers, epochs, batch size, and anything else you can think of.

Requirements:

- Model must contain Conv2D, BatchNormalization, and Dropout layers.
- These must be imported from the argument namespaces (already done by default).
- Task 1 will automatically use `tf.keras.layers` implementations.

In [22]:
```
import assignment

## You can test with more epochs later
cnn_model = assignment.run_task(data, 1, epochs=10, batch_size=100)
```

```
Starting Model Training
Epoch 1/10
250/250 [==============================] - 17s 66ms/step - loss: 1.7903 -
categorical_accuracy: 0.3468 - val_loss: 1.9747 - val_categorical_accurac
y: 0.3500
Epoch 2/10
250/250 [==============================] - 16s 65ms/step - loss: 1.4505 -
categorical_accuracy: 0.4735 - val_loss: 1.3848 - val_categorical_accurac
y: 0.5065
Epoch 3/10
250/250 [==============================] - 18s 74ms/step - loss: 1.3107 -
categorical_accuracy: 0.5276 - val_loss: 1.3348 - val_categorical_accurac
y: 0.5151
Epoch 4/10
250/250 [==============================] - 18s 72ms/step - loss: 1.2218 -
categorical_accuracy: 0.5638 - val_loss: 1.3671 - val_categorical_accurac
y: 0.5137
Epoch 5/10
250/250 [==============================] - 19s 75ms/step - loss: 1.1447 -
categorical_accuracy: 0.5931 - val_loss: 1.2582 - val_categorical_accurac
y: 0.5444
Epoch 6/10
250/250 [==============================] - 19s 76ms/step - loss: 1.0908 -
categorical_accuracy: 0.6160 - val_loss: 1.2392 - val_categorical_accurac
y: 0.5629
Epoch 7/10
250/250 [==============================] - 20s 79ms/step - loss: 1.0322 -
categorical_accuracy: 0.6373 - val_loss: 1.2098 - val_categorical_accurac
y: 0.5749
Epoch 8/10
250/250 [==============================] - 24s 97ms/step - loss: 0.9982 -
categorical_accuracy: 0.6502 - val_loss: 1.1386 - val_categorical_accurac
y: 0.6066
Epoch 9/10
250/250 [==============================] - 24s 96ms/step - loss: 0.9561 -
categorical_accuracy: 0.6651 - val_loss: 1.1211 - val_categorical_accurac
y: 0.6157
Epoch 10/10
250/250 [==============================] - 25s 100ms/step - loss: 0.9286
- categorical_accuracy: 0.6720 - val_loss: 1.1776 - val_categorical_accur
acy: 0.6024
```

In [10]:
```
import assignment

## You can test with more epochs later
cnn_model = assignment.run_task(data, 2, epochs=10, batch_size=100)
```

```
Starting Model Training
Epoch 1/10
250/250 [==============================] - 14s 54ms/step - loss: 1.8251 -
categorical_accuracy: 0.3322 - val_loss: 2.0004 - val_categorical_accurac
y: 0.4045
Epoch 2/10
250/250 [==============================] - 14s 56ms/step - loss: 1.4455 -
categorical_accuracy: 0.4794 - val_loss: 1.4360 - val_categorical_accurac
y: 0.4819
Epoch 3/10
250/250 [==============================] - 16s 64ms/step - loss: 1.3056 -
categorical_accuracy: 0.5295 - val_loss: 1.5798 - val_categorical_accurac
y: 0.4814
Epoch 4/10
250/250 [==============================] - 14s 56ms/step - loss: 1.2001 -
categorical_accuracy: 0.5754 - val_loss: 1.2868 - val_categorical_accurac
y: 0.5458
Epoch 5/10
250/250 [==============================] - 14s 54ms/step - loss: 1.1306 -
categorical_accuracy: 0.6013 - val_loss: 1.1683 - val_categorical_accurac
y: 0.5870
Epoch 6/10
250/250 [==============================] - 13s 54ms/step - loss: 1.0681 -
categorical_accuracy: 0.6226 - val_loss: 1.1520 - val_categorical_accurac
y: 0.5915
Epoch 7/10
250/250 [==============================] - 14s 55ms/step - loss: 1.0132 -
categorical_accuracy: 0.6421 - val_loss: 1.1487 - val_categorical_accurac
y: 0.5939
Epoch 8/10
250/250 [==============================] - 14s 54ms/step - loss: 0.9723 -
categorical_accuracy: 0.6573 - val_loss: 1.0835 - val_categorical_accurac
y: 0.6199
Epoch 9/10
250/250 [==============================] - 14s 55ms/step - loss: 0.9361 -
categorical_accuracy: 0.6694 - val_loss: 1.1320 - val_categorical_accurac
y: 0.6039
Epoch 10/10
250/250 [==============================] - 14s 54ms/step - loss: 0.9019 -
categorical_accuracy: 0.6826 - val_loss: 1.0932 - val_categorical_accurac
y: 0.6245
```

In [11]:
```
cnn_model.summary()
```

```
Model: "custom_sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_10 (Conv2D)          (None, 16, 16, 32)        896       
                                                                 
 max_pooling2d_6 (MaxPooling  (None, 8, 8, 32)         0         
 2D)                                                             
                                                                 
 batch_normalization_4 (Batc  (None, 8, 8, 32)         2         
 hNormalization)                                                 
                                                                 
 conv2d_11 (Conv2D)          (None, 4, 4, 64)          18496     
                                                                 
 conv2d_12 (Conv2D)          (None, 2, 2, 64)          36928     
                                                                 
 max_pooling2d_7 (MaxPooling  (None, 1, 1, 64)         0         
 2D)                                                             
                                                                 
 conv2d_13 (Conv2D)          (None, 1, 1, 64)          36928     
                                                                 
 conv2d_14 (Conv2D)          (None, 1, 1, 64)          36928     
                                                                 
 max_pooling2d_8 (MaxPooling  (None, 1, 1, 64)         0         
 2D)                                                             
                                                                 
 flatten_2 (Flatten)         (None, 64)                0         
                                                                 
 dense_4 (Dense)             (None, 128)               8320      
                                                                 
 dropout_2 (Dropout)         (None, 128)               0         
                                                                 
 batch_normalization_5 (Batc  (None, 128)              2         
 hNormalization)                                                 
                                                                 
 dense_5 (Dense)             (None, 10)                1290      
                                                                 
=================================================================
Total params: 139,790
Trainable params: 139,790
Non-trainable params: 0
_________________________________________________________________
```

Make Your Own Layers

For steps 3, 4, and 5 from the handout, you'll need to implement the layers from scratch inside of `layers_keras.py`. Feel free to refer to the official documentation for how these methods are supposed to function. More details are included in the layer block comments, and the init methods are already provided.

Requirements:

- Implement Conv2D, BatchNormalization, and Dropout in `layers_keras.py`

- Cannot use existing layers as sub-components.
- Cannot use `tf.nn.batch_normalization` or `tf.nn.dropout`.
- CAN use `tf.nn.convolution` ...
- Should utilize all non-commented-out arguments.

2D Convolution

Use the below code block to confirm that your custom implementation of Conv2D runs without erroring. This does not guarantee that your forward pass calculations are correct. It serves only as a preliminary check.

In [12]:
```
import layers_keras

random_input = tf.random.uniform((1, 4, 4, 3), 0, 10, dtype=tf.float32)

seed = 8675309
tf.random.set_seed(seed)
conv_layer = layers_keras.Conv2D(1, 2, strides=2)
print("Output:", conv_layer(random_input, training=True))

tf.random.set_seed(seed)
conv_layer = tf.keras.layers.Conv2D(1, 2, strides=2)
print('Expected:', conv_layer(random_input, training=True))
```

```
Output: tf.Tensor(
[[[[ 3.449257 ]
   [ 2.7008412]]

  [[-1.7479116]
   [-1.3944378]]]], shape=(1, 2, 2, 1), dtype=float32)
Expected: tf.Tensor(
[[[[ 3.449257 ]
   [ 2.7008412]]

  [[-1.7479116]
   [-1.3944378]]]], shape=(1, 2, 2, 1), dtype=float32)
```

Batch Normalization

Use the below code block to confirm that your custom implementation of Batch Normalization runs without erroring. This does not guarantee that your forward pass calculations are correct. It serves only as a preliminary check.

In [13]:
```
import layers_keras

random_input = tf.random.uniform((3,3), 0, 10, dtype=tf.float32)
print("Input:", random_input)

batch_norm = layers_keras.BatchNormalization()
print("Output:", batch_norm(random_input, training=True))

batch_norm = tf.keras.layers.BatchNormalization()
print('Expected:', batch_norm(random_input, training=True))
```

```
Input: tf.Tensor(
[[4.9200354  4.1594877  0.15996099]
 [0.17104864 3.8494146  4.5375834 ]
 [3.0904114  8.937246   6.2552547 ]], shape=(3, 3), dtype=float32)
Output: tf.Tensor(
[[ 1.1211213  -0.6394283  -1.3602846 ]
 [-1.3068337  -0.77256405  0.34549025]
 [ 0.18571234  1.4119927   1.0147943 ]], shape=(3, 3), dtype=float32)
Expected: tf.Tensor(
[[ 1.121121   -0.63942826 -1.3602844 ]
 [-1.3068335  -0.77256405  0.34549022]
 [ 0.18571234  1.4119925   1.0147942 ]], shape=(3, 3), dtype=float32)
```

Dropout

Use the below code block to confirm that your custom implementation of Dropout runs without erroring. This does not guarantee that your forward pass or input gradients calculations are correct. It serves only as a preliminary check.

In [14]:
```
import layers_keras

random_input = tf.ones((2, 11))
print("Input:\n", random_input)

seed = 8675309
for mode_str, mode in zip(['Training', 'Testing'], [True, False]):
    print()
    for layer_str, layer in zip(['Output','Expected'], [layers_keras.Dropou
        tf.random.set_seed(seed)
        dropout_fn = layer(rate=0.2)
        print(f'{layer_str} {mode_str}:')
        print(dropout_fn(random_input, training=mode))

# Expected: Around rate% of the entries should be zeros in training mode.
#   Should also be normalized such that, on average, magnitude perserved.
```

```
Input:
 tf.Tensor(
[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]], shape=(2, 11), dtype=float32)

Output Training:
tf.Tensor(
[[1.25 1.25 1.25 1.25 1.25 1.25 0.   1.25 1.25 1.25 0.  ]
 [1.25 1.25 1.25 1.25 1.25 0.   1.25 0.   0.   1.25 1.25]], shape=(2, 1
1), dtype=float32)
Expected Training:
tf.Tensor(
[[1.25 1.25 1.25 1.25 1.25 1.25 0.   1.25 1.25 1.25 0.  ]
 [1.25 1.25 1.25 1.25 1.25 0.   1.25 0.   0.   1.25 1.25]], shape=(2, 1
1), dtype=float32)

Output Testing:
tf.Tensor(
[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]], shape=(2, 11), dtype=float32)
Expected Testing:
tf.Tensor(
[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]], shape=(2, 11), dtype=float32)
```

Training your model

Now, let's see if your model works with the new components in place?

In [15]:
```
import assignment

# assignment.run_task(data, 2, 1, epochs=2)   ## Just manual conv
# assignment.run_task(data, 2, 2, epochs=2)   ## Just manual bnorm
# assignment.run_task(data, 2, 3, epochs=2)   ## Just manual dropout
assignment.run_task(data, 2, epochs=2)        ## Test all 3!
```

```
Starting Model Training
Epoch 1/2
100/100 [==============================] - 8s 75ms/step - loss: 1.9125 -
categorical_accuracy: 0.2966 - val_loss: 2.2419 - val_categorical_accurac
y: 0.2597
Epoch 2/2
100/100 [==============================] - 8s 77ms/step - loss: 1.5696 -
categorical_accuracy: 0.4269 - val_loss: 2.1106 - val_categorical_accurac
y: 0.3155
```

Out[15]:
```
<conv_model.CustomSequential at 0x17d59d550>
```

Manual Convolution!

Now, go ahead and implement convolution manually! This should be done inside of the `layers_manual.py` file. It's very non-trivial to perform convolution differentiably without using `tf.nn.convolution`, so the manual convolution should only run during inference time. Below is a quick test to see if your convolution is consistent with the Keras layered version:

In [16]:
```
import layers_manual

random_input = tf.random.uniform((2, 4, 4, 3), 0, 10, dtype=tf.float32)

seed = 8675309
tf.random.set_seed(seed)
conv_layer = layers_manual.Conv2D(1, (2, 2), strides=2, padding='valid')
print("Output:", conv_layer(random_input, training=False))

tf.random.set_seed(seed)
conv_layer = tf.keras.layers.Conv2D(1, (2, 2), strides=2, padding='valid')
print('Expected:', conv_layer(random_input, training=False))
```

```
Output: tf.Tensor(
[[[[ 0.02713728]
   [ 6.7044606 ]]

  [[ 2.2370446 ]
   [-0.6432083 ]]]


 [[[-1.5071795 ]
   [ 3.6331491 ]]

  [[ 0.9108801 ]
   [ 0.46051323]]]], shape=(2, 2, 2, 1), dtype=float32)
Expected: tf.Tensor(
[[[[ 0.02713721]
   [ 6.7044606 ]]

  [[ 2.237044  ]
   [-0.6432083 ]]]


 [[[-1.5071793 ]
   [ 3.6331491 ]]

  [[ 0.9108796 ]
   [ 0.46051353]]]], shape=(2, 2, 2, 1), dtype=float32)
```

Inside the loop, this will happen at the end of every epoch because a validation set is being evaluated alongside your training set. The following will test it out for you! Don't worry if your categorical accuracy looks low here. As long as everything works without erroring, feel free to move on and test the whole model together.

In [17]:
```
import assignment

assignment.run_task(data, 3, epochs=5)
```

```
Starting Model Training
Epoch 1/5
1/1 [==============================] - 0s 325ms/step - loss: 2.6259 - cat
egorical_accuracy: 0.1120 - val_loss: 2.2993 - val_categorical_accuracy:
0.1000
Epoch 2/5
1/1 [==============================] - 0s 112ms/step - loss: 2.4053 - cat
egorical_accuracy: 0.1200 - val_loss: 2.2992 - val_categorical_accuracy:
0.1200
Epoch 3/5
1/1 [==============================] - 0s 97ms/step - loss: 2.3048 - cate
gorical_accuracy: 0.1920 - val_loss: 2.2985 - val_categorical_accuracy:
0.1200
Epoch 4/5
1/1 [==============================] - 0s 95ms/step - loss: 2.2261 - cate
gorical_accuracy: 0.2000 - val_loss: 2.2978 - val_categorical_accuracy:
0.1120
Epoch 5/5
1/1 [==============================] - 0s 94ms/step - loss: 2.1640 - cate
gorical_accuracy: 0.2560 - val_loss: 2.2972 - val_categorical_accuracy:
0.1080
```

Out[17]:
```
<conv_model.CustomSequential at 0x295ebb2b0>
```

Wrapping Up

Make sure your model runs and trains up to standards! When you find a model configuration that you like, feel free to update your `get_default_CNN_model` function so that the autograder can use it with your arguments. If your model takes too long to train (> 10 mins), the autograder may time out, so take consideration of that.

In [18]:
```
## Run at least once
from types import SimpleNamespace
from conv_model import CustomSequential
```

For convenience, you can copy your code here for quick testing!

Make sure to put it back into your `conv_model.py` file for the autograder!

In [19]:
```
def get_default_CNN_model(
    conv_ns=tf.keras.layers,
    norm_ns=tf.keras.layers,
    drop_ns=tf.keras.layers,
    man_conv_ns=tf.keras.layers,
):
    return None
```

In [20]:
```
conv_ns = tf.keras.layers
norm_ns = tf.keras.layers
drop_ns = tf.keras.layers
man_conv_ns = tf.keras.layers

args = get_default_CNN_model(
    conv_ns=conv_ns,
    norm_ns=norm_ns,
    drop_ns=drop_ns,
    man_conv_ns=man_conv_ns
)

history = args.model.fit(
    X0,
    Y0,
    epochs=args.epochs,
    batch_size=args.batch_size,
    validation_data=(X1, Y1),
)
```

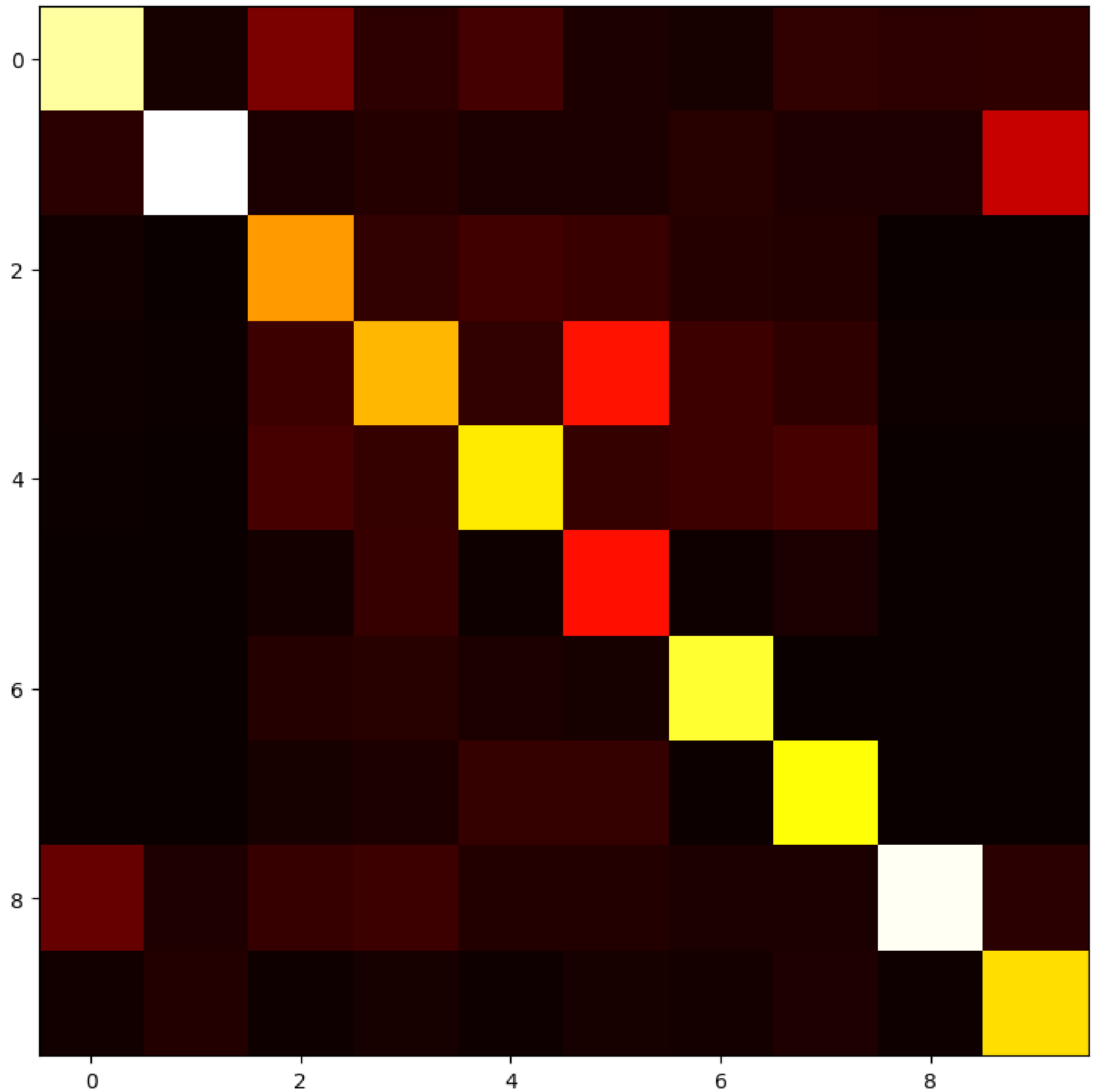
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call las
t)
Cell In [20], line 13
      4 man_conv_ns = tf.keras.layers
      6 args = get_default_CNN_model(
      7     conv_ns=conv_ns,
      8     norm_ns=norm_ns,
      9     drop_ns=drop_ns,
     10     man_conv_ns=man_conv_ns
     11 )
---> 13 history = args.model.fit(
     14     X0,
     15     Y0,
     16     epochs=args.epochs,
     17     batch_size=args.batch_size,
     18     validation_data=(X1, Y1),
     19 )

AttributeError: 'NoneType' object has no attribute 'model'
```

Sanity Checks

In case you need them!

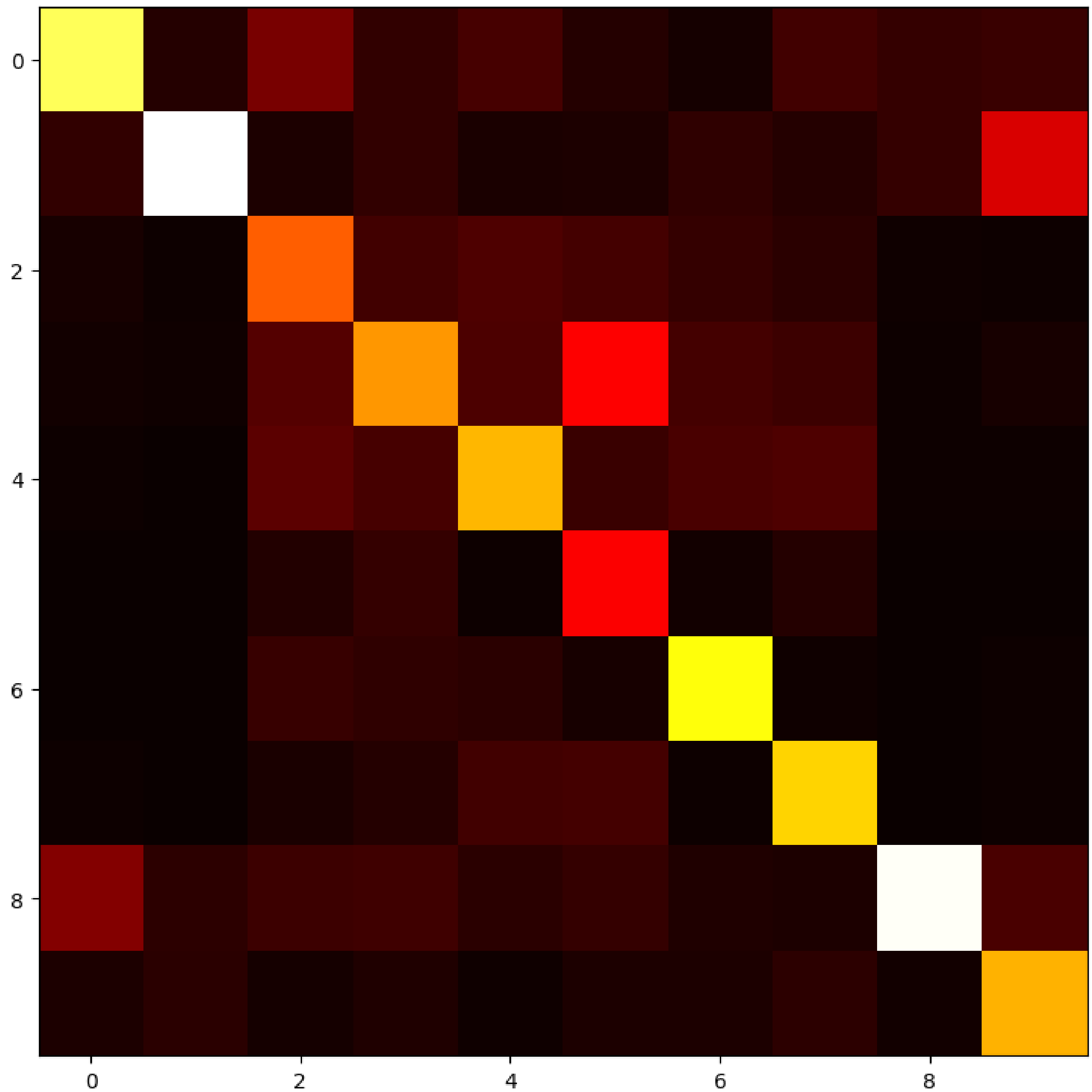
In [23]:
```
P1 = np.argmax(cnn_model.predict(X1), -1)
confusion_mtx = tf.math.confusion_matrix(P1, Y1)

P0 = np.argmax(cnn_model.predict(X0), -1)
confusion_mtx = tf.math.confusion_matrix(P0, Y0)
plt.figure(figsize=(12, 9))
plt.imshow(confusion_mtx, cmap='hot', interpolation='nearest')
```

Out[23]: <matplotlib.image.AxesImage at 0x3643f4310>

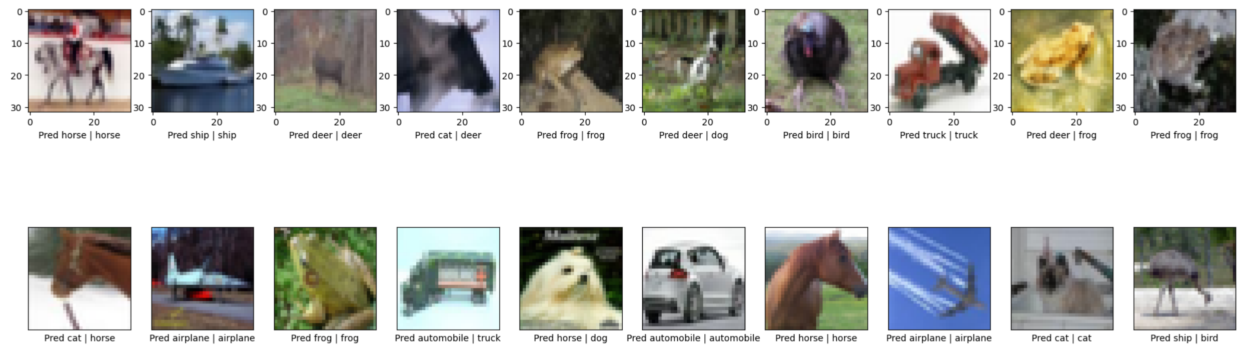


In [24]:
```
P1 = np.argmax(cnn_model.predict(X1), -1)
confusion_mtx = tf.math.confusion_matrix(P1, Y1)
plt.figure(figsize=(12, 9))
plt.imshow(confusion_mtx, cmap='hot', interpolation='nearest')
```

Out[24]:
```
<matplotlib.image.AxesImage at 0x33aaa2bb0>
```



In [25]:
```
fig, ax = plt.subplots(2, 10)
fig.set_size_inches(24, 8)

pred0 = cnn_model.predict(X0[:10])
pred1 = cnn_model.predict(X1[:10])

def p2l(pred):
    return D_info.features['label']._int2str[pred]

for i in range(10):
    ax[0][i].imshow(X0[i], cmap = "Greys")
    ax[1][i].imshow(X1[i], cmap = "Greys")
    ax[1][i].tick_params(left=False, bottom=False, labelleft=False, labelbo
    ax[0][i].set_xlabel(f"Pred {p2l(np.argmax(pred0[i], -1))} | {p2l(Y0[i])
    ax[1][i].set_xlabel(f"Pred {p2l(np.argmax(pred1[i], -1))} | {p2l(Y1[i])
```



In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: