COSC 2753 | Machine Learning

Week 10 Lab Exercises: **Neural Networks (MLP)**

Introduction

In this lab you will be:

1. Learning how to use Keras API (in tensorflow library).
2. Implement simple NN to classify images in the CIFAR-10 dataset.

The CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Here are the classes in the dataset, as well as 10 random images from each:

drawing

The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.

In this lab we will only use a subset of CIFAR10 data that has 10000 images.

Reading Data

The data is available on Canvas. You can upload it to the notebook work environment and unzip using the following code.

```
import zipfile
with zipfile.ZipFile('CIFAR10_Lab9.zip', 'r') as zip_ref:
    zip_ref.extractall('./')

```

The dataset consists of the images and a csv file. The labels and the image paths are in the CSV file.

Lets randomly split the data into train/val/test

```
import numpy as np
from sklearn.model_selection import train_test_split
import pandas as pd

data = pd.read_csv('./CIFAR_Data.csv')

train_data, test_data = train_test_split(data, test_size=0.2, random_state=42)
train_data, val_data = train_test_split(train_data, test_size=0.25, random_state=42)

print("Train data : {}, Val Data: {}, Test Data: {}".format(train_data.shape[0], val_data.shape[0], test_data.shape[0]))
```

```
Train data : 6000, Val Data: 2000, Test Data: 2000
```

EDA

> ☞ **Task: Do the EDA and identify the suitable performance measure.**

> ☞ **Task: Plot some images from the dataset and observe.**

```
print(train_data.columns)
```

```
Index(['ImgPath', 'Class'], dtype='object')
```

Exploratory Data Analysis (EDA) is a crucial step in understanding your CIFAR-10 dataset. For your image dataset, EDA can involve:

1. **Viewing Images**: Display a sample of images from each class to understand their visual characteristics. This will give you a sense of the variability within and across classes.
2. **Class Distribution**: Check if the dataset is balanced across different classes. This can be done by plotting the distribution of classes in your training, validation, and test sets.

1. **Viewing Images**: Display a sample of images from each class to understand their visual characteristics. This will give you a sense of the variability within and across classes.

```python
import matplotlib.pyplot as plt
import numpy as np

# Update to use the correct column name for classes
classes = np.unique(train_data['Class'])

# Number of samples to display from each class
num_samples = 5

# Plotting
fig, axes = plt.subplots(len(classes), num_samples, figsize=(num_samples*2, len(classes)*2))

for i, cls in enumerate(classes):
    # Selecting the first 'num_samples' images of class 'cls'
    class_images = train_data[train_data['Class'] == cls].iloc[:num_samples]
    for j in range(num_samples):
        # Load image from 'ImgPath' and reshape
        # You might need to adjust this part based on how you load images
        img = plt.imread(class_images.iloc[j]['ImgPath'])
        axes[i, j].imshow(img)
        axes[i, j].axis('off')
        if j == 0:
            axes[i, j].set_title(f"Class: {cls}")

plt.tight_layout()
plt.show()

```

The provided Python code is used to display a grid of images from a dataset, with each row in the grid corresponding to a different class of images. The code uses the `matplotlib` library for creating the grid and displaying the images, and the `numpy` library for handling the classes.

The variable `classes` is assigned the unique values in the 'Class' column of the `train_data` DataFrame, which represents the different classes of images in the dataset.

The variable `num_samples` is set to 5, indicating that five images from each class will be displayed.

The `plt.subplots` function is then used to create a grid of subplots with a number of rows equal to the number of classes and a number of columns equal to `num_samples`. The size of the figure is set to be proportional to the number of rows and columns.

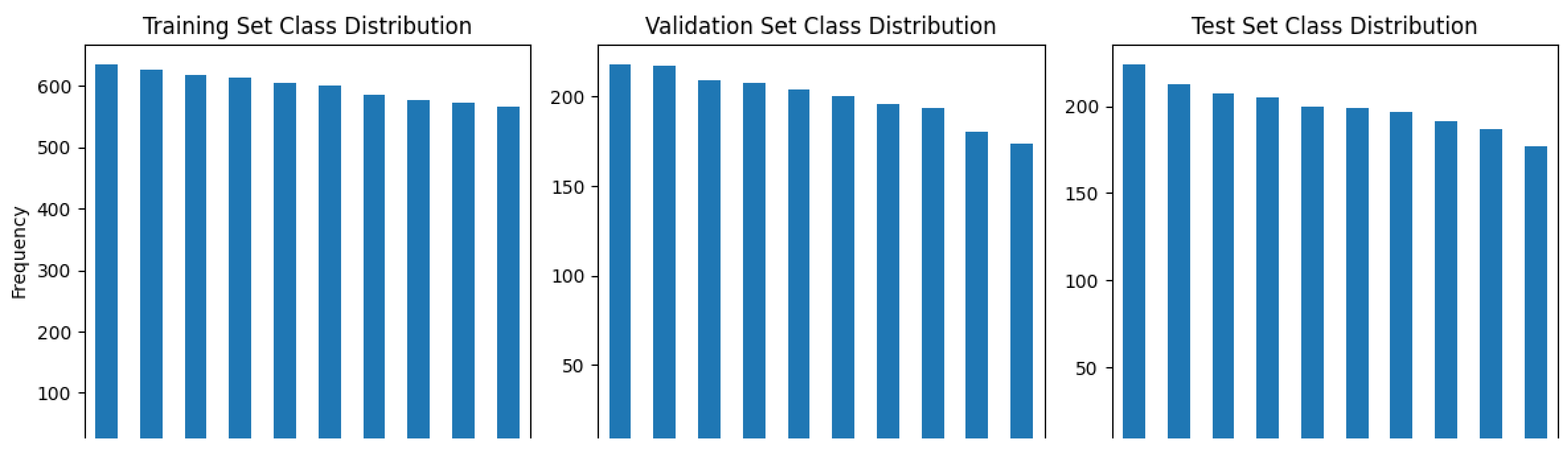
The code then enters a loop over the classes. For each class, it selects the first `num_samples` rows from `train_data` where the 'Class' column is equal to the current class. This subset of `train_data` is stored in the `class_images` DataFrame.

Inside this loop, there is another loop over the range `num_samples`. For each iteration, it reads the image file specified in the 'ImgPath' column of the `class_images` DataFrame, displays the image in the corresponding subplot using the `imshow` function, and turns off the axis. If the current sample is the first one (i.e., `j` is 0), it also sets the title of the subplot to the name of the class.

After both loops have finished, the `plt.tight_layout` function is called to adjust the spacing between the subplots, and `plt.show` is used to display the figure. The result is a grid of images where each row corresponds to a different class, and each column corresponds to a different sample from that class.

2. **Class Distribution**: Check if the dataset is balanced across different classes. This can be done by plotting the distribution of classes in your training, validation, and test sets.

```python
import matplotlib.pyplot as plt

# Count the occurrence of each class in each dataset
train_class_counts = train_data['Class'].value_counts()
val_class_counts = val_data['Class'].value_counts()
test_class_counts = test_data['Class'].value_counts()

# Create a bar plot for each dataset
plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)
train_class_counts.plot(kind='bar')
plt.title('Training Set Class Distribution')
plt.xlabel('Class')
plt.ylabel('Frequency')

plt.subplot(1, 3, 2)
val_class_counts.plot(kind='bar')
plt.title('Validation Set Class Distribution')
plt.xlabel('Class')

plt.subplot(1, 3, 3)
test_class_counts.plot(kind='bar')
plt.title('Test Set Class Distribution')
plt.xlabel('Class')

plt.tight_layout()
plt.show()

```



Performance metrics like accuracy_score, precision, recall, F1 score, or ROC-AUC could be considered depending on the specific requirements and characteristics of the dataset, such as class imbalance or the importance of false positives versus false negatives.

Model development

As discussed in Week 4-5, the typical ML model development process consists of 4 steps, lets go through each and see how it is done.

1. **Determine your goals**: Performance metric and target value. Problem dependent.
2. **Setup the experiment**: Setup the test/validation data, visualisers and debuggers needed to determine bottlenecks in performance (overfitting/under-fitting, feature importance).
3. **Default Baseline Model**: Identify the components of end-to-end pipeline including - Baseline Models, cost functions, optimisation.
4. **Make incremental changes**: Repeatedly make incremental changes such as gathering new data, adjusting hyper-parameters, or changing algorithms, based on specific findings from your instrumentation.

So far we conducted the EDA and accordingly we can decide that a performance measure such as Accuracy is adequate for this task. This is justified by the observation that there is no label imbalance in our dataset and the task is multi-class classification. According to the CIFAR website others have achieved around 82% accuracy for the full dataset. Therefore we can set that as the target performance value. However this might be unrealistic given:

- We are only using a subset of data in this lab (10000/50000) to train the model.
- We are using a traditional neural network (not CNN).
- No data augmentation.

For a traditional NN a 50% accuracy target will be more realistic.

We have also identified the train/test/val splits and ready to do the experiments. In developing a neural network, we can use the learning curves to identify the next action to take. so let's write a simple function to plot the learning curve of a NN training process. This will be our diagnostic tool.

```
import matplotlib.pyplot as plt
def plot_learning_curve(train_loss, val_loss, train_metric, val_metric, metric_name='Accuracy'):
    plt.figure(figsize=(10,5))

    plt.subplot(1,2,1)
    plt.plot(train_loss, 'r--')
    plt.plot(val_loss, 'b--')
    plt.xlabel("epochs")
    plt.ylabel("Loss")
    plt.legend(['train', 'val'], loc='upper left')

    plt.subplot(1,2,2)
    plt.plot(train_metric, 'r--')
    plt.plot(val_metric, 'b--')
    plt.xlabel("epochs")
    plt.ylabel(metric_name)
    plt.legend(['train', 'val'], loc='upper left')

    plt.show()
```

The provided Python code defines a function `plot_learning_curve` that uses the `matplotlib` library to plot the learning curves of a model during training. The function takes five arguments: `train_loss`, `val_loss`, `train_metric`, `val_metric`, and `metric_name`.

Here's a step-by-step explanation of what the code does:

1. `plt.figure(figsize=(10,5))` : This line creates a new figure with a width of 10 inches and a height of 5 inches.
2. `plt.subplot(1,2,1)` : This line creates the first subplot in a 1x2 grid of subplots. This subplot will be used to plot the loss.
3. `plt.plot(train_loss, 'r--')` and `plt.plot(val_loss, 'b--')` : These lines plot the training and validation loss, respectively. The 'r--' and 'b--' arguments specify that the training loss should be plotted with a red dashed line and the validation loss with a blue dashed line.
4. `plt.xlabel("epochs")` and `plt.ylabel("Loss")` : These lines set the labels for the x-axis and y-axis of the first subplot.
5. `plt.legend(['train', 'val'], loc='upper left')` : This line adds a legend to the first subplot that indicates which line corresponds to the training loss and which to the validation loss.
6. `plt.subplot(1,2,2)` : This line creates the second subplot in the 1x2 grid. This subplot will be used to plot the metric.
7. `plt.plot(train_metric, 'r--')` and `plt.plot(val_metric, 'b--')` : These lines plot the training and validation metric, respectively.
8. `plt.xlabel("epochs")` and `plt.ylabel(metric_name)` : These lines set the labels for the x-axis and y-axis of the second subplot.
9. `plt.legend(['train', 'val'], loc='upper left')` : This line adds a legend to the second subplot.
10. `plt.show()` : This line displays the figure with the two subplots.

Base Model

Next we need to establish a base model. In this lab we focus on MLP, therefore let's setup a simple MLP. In this network let's set the hidden layer dimension to 256. This is a hyper parameter and we can tune it later.

```
INPUT_DIM = (32,32,3)
HIDDEN_LAYER_DIM = 256
OUTPUT_CLASSES = 10
```

Now we are ready to setup out NN model. We will use tensorflow to build our neural network and train. To be precise, we will be using Keras which is a high level API which is part of tensorflow. First we need to setup and use Keras. In AWS Sagemaker you can change the ipython notebook kernel to `conda\_tensorflow\_p36` which will load a kernel that include Keras and Tensorflow. Typically this is not installed by default in the local anaconda environment.

Setup Keras locally

Ignore this section if you are on AWS or colab

Anaconda does not come with the Neural Network and Deep Learning packages installed by default, therefore we are going to have to install them ourselves. To install, open Anaconda, and select the `Envrionments` tab from the left-hand bar. This tab lists all of the python and sci-kit learn packages that are installed on your machine, and all available packages to install.
drawing

Leave the `base(root)` environment selected. *(If you really want to keep the deep learning packages separate you can create a new environment. This makes it easier to clean-up these packages later, but will consume more disk-space. You will also need to ensure you are in the correct environment when you launch Jupyter).*

We need to install the package:

1. tensorflow

Tensorflow is an Open Source Software Library for Machine Intelligence. Keras is a high-level neural networks API specification (in tensorflow), implemented in Python and capable of running on top of either TensorFlow or Theano. We will be using Keras and Tensorflow as the backend throughout the lab. It was developed with a focus on enabling fast experimentation and it allows to go from idea to result with the least possible delay without worrying about the numerical details of floating point operations, tensor algebra and GPU programming.

To install the packages, select `All`, from the drop-down
drawing

Search for `tensorflow` and select the check-mark to note the package should be installed.
drawing

Finally click `Apply` at the bottom-right corner, and confirm the installation in the pop-up window. This will install all of the necessary packages, and make take some time.

To test your installation

```
import tensorflow as tf
AUTOTUNE = tf.data.experimental.AUTOTUNE

tf.__version__
```

```
'2.15.0'
```

You should see the version number as output. It should be greater than 2.0.0

Building the neural network requires configuring the layers of the model, then compiling the model and finally training the model.

Set up the layers

The basic building block of a neural network is the layer. Layers extract representations from the data fed into them. Hopefully, these representations are meaningful for the problem at hand.

Most neural networks consists of chaining together simple layers. Most layers, such as `tf.keras.layers.Dense`, have parameters that are learned during training.

A layer in the MLP is represented by `tf.keras.layers.Dense`. First lets define the dimensions of our neural network.

There are three ways to build a model in tensorflow:

- Functional API
- Sub-classing
- Sequential API

We will use the `Sequential` API to build models as it is the simplest.

```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=INPUT_DIM),
    tf.keras.layers.Dense(HIDDEN_LAYER_DIM, activation='sigmoid'),
    tf.keras.layers.Dense(OUTPUT_CLASSES)
])
```

The first layer in this network, `tf.keras.layers.Flatten`, transforms the format of the images from a two-dimensional array (of 32 by 32 pixels) to a one-dimensional array (of 32 * 32 * 3 = 3072 pixels). Think of this layer as unstacking rows of pixels in the image and lining them up. This layer has no parameters to learn; it only reformats the data.

After the pixels are flattened, the network consists of a sequence of two `tf.keras.layers.Dense` layers. These are densely connected, or fully connected (MLP), neural layers. The first Dense layer has 256 nodes (or neurons). The second (and last) layer returns a logits array with length of 10. Each node contains a score that indicates the current image belongs to one of the 10 classes.

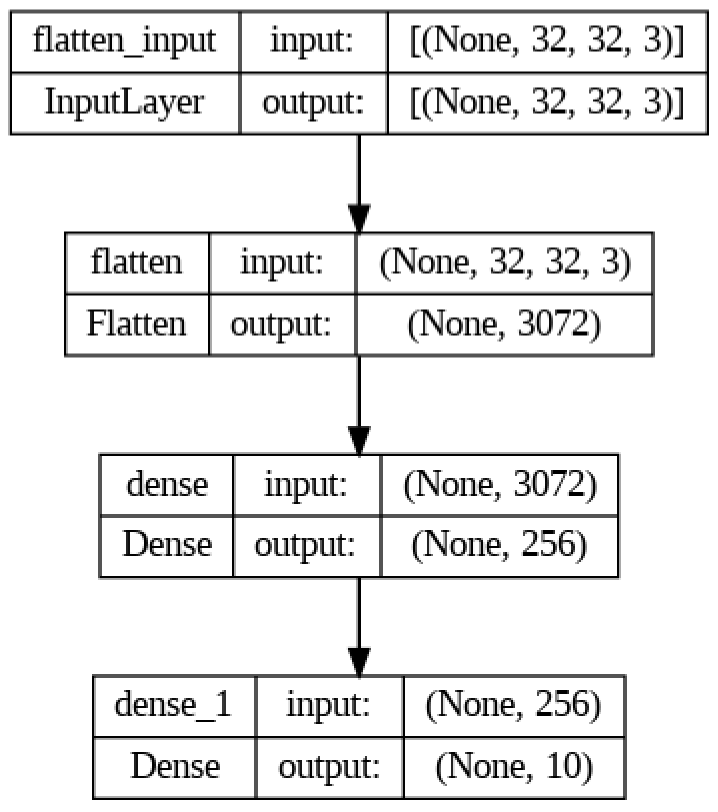
We can use `model.summary()` to print the model that was created.

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 flatten (Flatten)           (None, 3072)              0         
                                                                 
 dense (Dense)               (None, 256)               786688    
                                                                 
 dense_1 (Dense)             (None, 10)                2570      
                                                                 
=================================================================
Total params: 789258 (3.01 MB)
Trainable params: 789258 (3.01 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

The `tf.keras.utils.plot_model` shows the model as a figure

```
tf.keras.utils.plot_model(model, show_shapes=True)
```


flatten_input | input: | [(None, 32, 32, 3)]
InputLayer | output: | [(None, 32, 32, 3)]
flatten | input: | (None, 32, 32, 3)
Flatten | output: | (None, 3072)
dense | input: | (None, 3072)
Dense | output: | (None, 256)
dense_1 | input: | (None, 256)
Dense | output: | (None, 10)

> **Task: Read the tensorflow documentation on `Sequential` models url.**

Compile the model

Before the model is ready for training, it needs a few more settings. These are added during the model's compile step:

- **Loss function**: This measures how accurate the model is during training. You want to minimize this function to "steer" the model in the right direction.
- **Optimizer**: This is how the model is updated based on the data it sees and its loss function.
- **Metrics**: Used to monitor the training and testing steps. The following example uses accuracy, the fraction of the images that are correctly classified.

```
model.compile(optimizer='SGD',
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=['categorical_accuracy'])
```

> **Task: Read the tensorflow documentation on `losses` models url to identify other possible options for loss.**

> **Task: Read the tensorflow documentation on `metrics` models url to identify other possible options for metric.**

Train the model

Now we are ready to train the model. As are going to work with much larger and more complicated data set, that neural network are more suited towards, we need to write efficient code to load the data in batches to memory. This is done in keras using Image data generators.

To help we will define a loading function that takes the data frames we defined earlier:

We can use the `flow_from_dataframe` function in the keras data loader to load a set of images directly from a pandas data frame (the data frame should contain the image names (path) and the labels)

For some reason ImageDataGenerator need the labels to be in string format. Lets do that

```
train_data['Class'] = train_data['Class'].astype('str')
val_data['Class'] = val_data['Class'].astype('str')
```

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1./255, data_format='channels_last')
val_datagen = ImageDataGenerator(rescale=1./255, data_format='channels_last')

batch_size = 32

train_generator = train_datagen.flow_from_dataframe(
        dataframe=train_data,
        directory='./',
        x_col="ImgPath",
        y_col="Class",
        target_size=(32, 32),
        batch_size=batch_size,
        class_mode='categorical')

validation_generator = val_datagen.flow_from_dataframe(
        dataframe=val_data,
        directory='./',
        x_col="ImgPath",
        y_col="Class",
        target_size=(32, 32),
        batch_size=batch_size,
        class_mode='categorical')
```

```
Found 6000 validated image filenames belonging to 10 classes.
Found 2000 validated image filenames belonging to 10 classes.
```

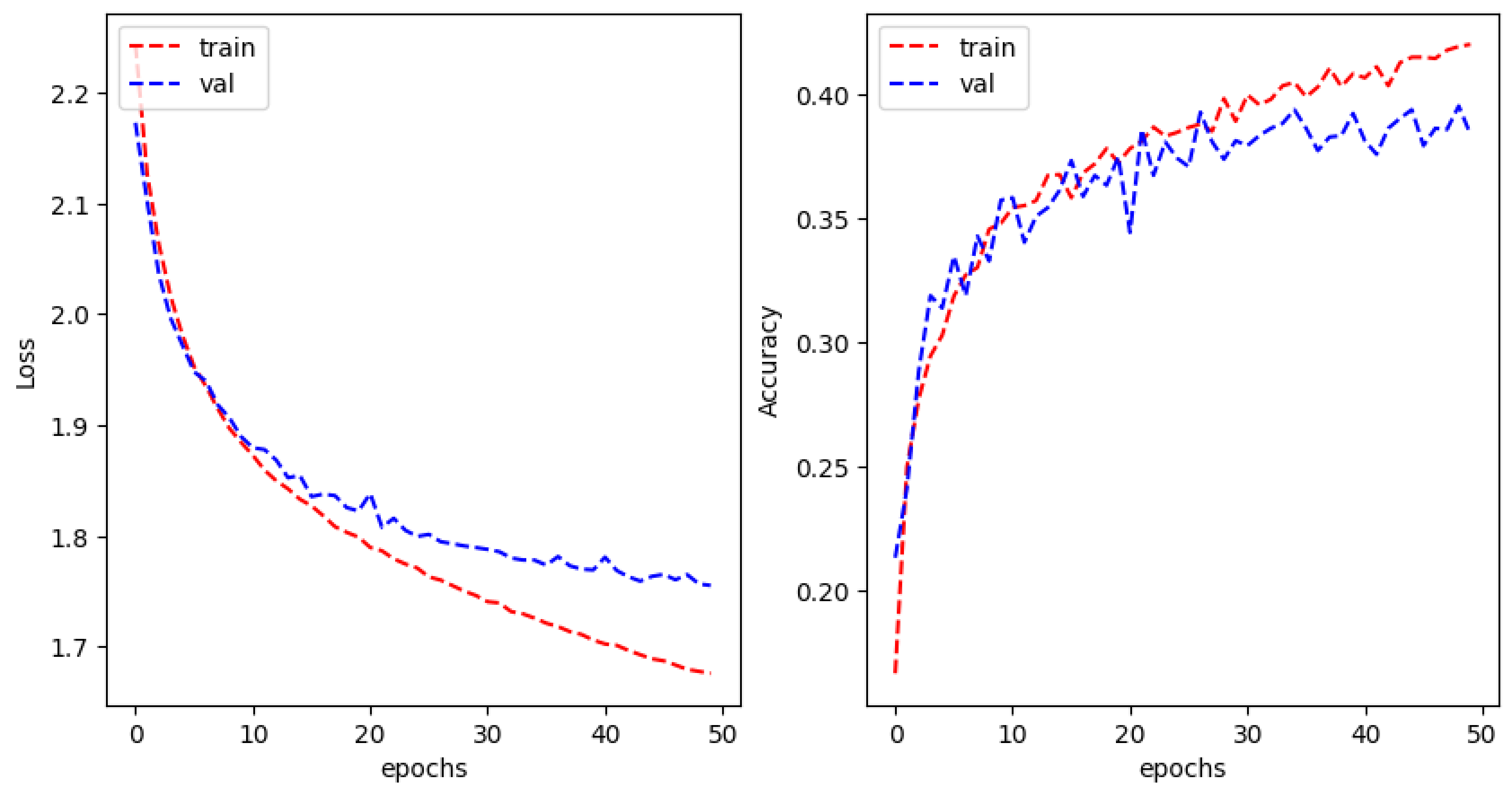
With this, the training the validation data can be loaded. You will need to supply the appropriate directory. Usually each pixels varies from 0-255, but it's highly recommended to normalize them in range of 0-1 to speed up the training process. The dataloader also do a simple normalisation on the pixel values directly.

Now we can use a simple model.fit_generator() in Keras to train the model.

```
history = model.fit_generator(train_generator, validation_data = validation_generator, epochs=50, verbose=0)
```

```
<ipython-input-17-243f5d5117d3>:1: UserWarning: `Model.fit_generator` is deprecated and will be removed in a future version. Please use `Model.fit`, which supports generat
  history = model.fit_generator(train_generator, validation_data = validation_generator, epochs=50, verbose=0)
```

```
plot_learning_curve(history.history['loss'], history.history['val_loss'],
                    history.history['categorical_accuracy'], history.history['val_categorical_accuracy'],
                    metric_name='Accuracy')
```



Now you know how to setup a neural network and train it in tensorflow.

Incremental changes

Next step is doing the incremental updates to improve the performance. Observe the loss curves and act accordingly. Some options are.

If the model is under fitting:

1. Increase the number of neurones in the hidden layer
2. Increase the number of hidden layers

If the model is over fitting:

1. Add regulatization: Lasso/Ridge penalty
2. Add dropout
3. Reduce number of neurones in each layer or number of layers.

Some of the steps are done in the LectureQandA code for week 9 module - neural networks. You can get the basic syntax from there and implement accordingly.

> ☞ **Task: Tune the neural network model to get the best performance possible.**

```
import tensorflow as tf

INPUT_DIM = (32,32,3)
HIDDEN_LAYER_DIM = 512  # Increased from 256 to 512
OUTPUT_CLASSES = 10

# Updated model with an additional hidden layer and ReLU activation
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=INPUT_DIM),
    tf.keras.layers.Dense(HIDDEN_LAYER_DIM, activation='relu'),  # Changed to ReLU
    tf.keras.layers.Dense(HIDDEN_LAYER_DIM, activation='relu'),  # Added another hidden layer
    tf.keras.layers.Dropout(0.5),  # Added Dropout
    tf.keras.layers.Dense(OUTPUT_CLASSES)
])

# Updated optimizer to Adam
model.compile(optimizer='adam',  # Changed to Adam optimizer
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=['categorical_accuracy'])

# Assuming 'train_generator' and 'validation_generator' are already defined as before

# Fit the model with potentially more epochs, and use the updated fit method
history = model.fit(train_generator, validation_data=validation_generator, epochs=30, verbose=1)

# Function 'plot_learning_curve' remains the same as before

# Call to plot the learning curve
plot_learning_curve(history.history['loss'], history.history['val_loss'],
                    history.history['categorical_accuracy'], history.history['val_categorical_ac
                    metric_name='Accuracy')


```

Testing the final model

Now lets test our final model on the test data.

First we will create a data generator to get the test data

```
test_data['Class'] = test_data['Class'].astype('str')

test_datagen = ImageDataGenerator(rescale=1./255, data_format='channels_last')

batch_size = 1

test_generator = test_datagen.flow_from_dataframe(
        dataframe=test_data,
        directory='./',
        x_col="ImgPath",
        y_col="Class",
        target_size=(32, 32),
        batch_size=batch_size,
        class_mode='categorical')
```

```
Found 2000 validated image filenames belonging to 10 classes.
```

```
model.evaluate(test_generator)
```

```
2000/2000 [==============================] - 6s 3ms/step - loss: 1.6500 - categorical_accuracy: 0.4160
[1.6499924659729004, 0.41600000858306885]
```

Ploting some results

Lets visualize the output of our network.

```
label_names = {'airplane' : 0, 'automobile' : 1, 'bird' : 2, 'cat' : 3, 'deer' : 4, 'dog' : 5, 'frog' : 6, 'horse' : 7, 'ship' : 8, 'truck' : 9}
```

```
d_inv = {v: k for k, v in label_names.items()}
plt.figure(figsize=(16,4))
batches = 0
for x,y in test_generator:
        batches = batches + 1
        y_hat = model.predict(x, verbose=0)
        x = np.squeeze(x)
        if batches < 5:
            plt.subplot(1,5,batches)
            plt.imshow(x)
            plt.title("GT-{}, Pred-{}".format(d_inv[np.argmax(y[0])], d_inv[np.argmax(y_hat[0])]

        else:
            break

        plt.axis('off')
```