CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Deep Learning

Deep Learning

Lecture: Using Keras

Ted Scully

Keras

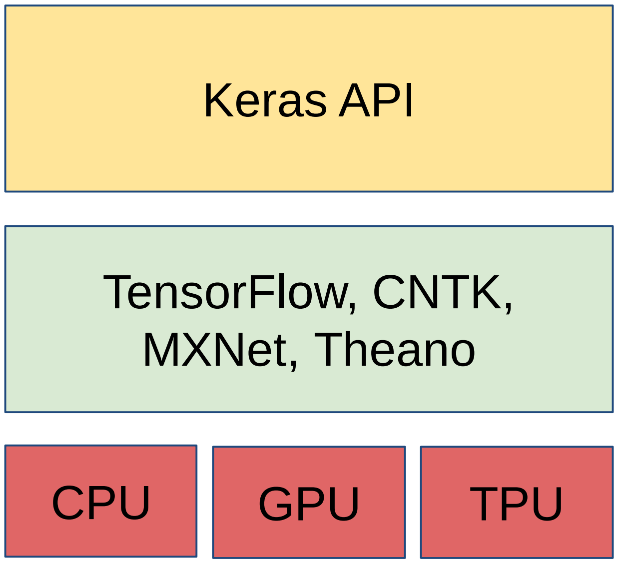
1. Keras is a **<u>high-level API</u>** for building neural networks, written in Python and capable of running on top of TensorFlow, CNTK, MXNet or Theano.

2. The objective of Keras is to facilitate fast prototyping and experimentation.

3. It is designed to be very easy to use and highly modular.

4. It supports both convolutional networks and recurrent networks, as well as combinations of the two.

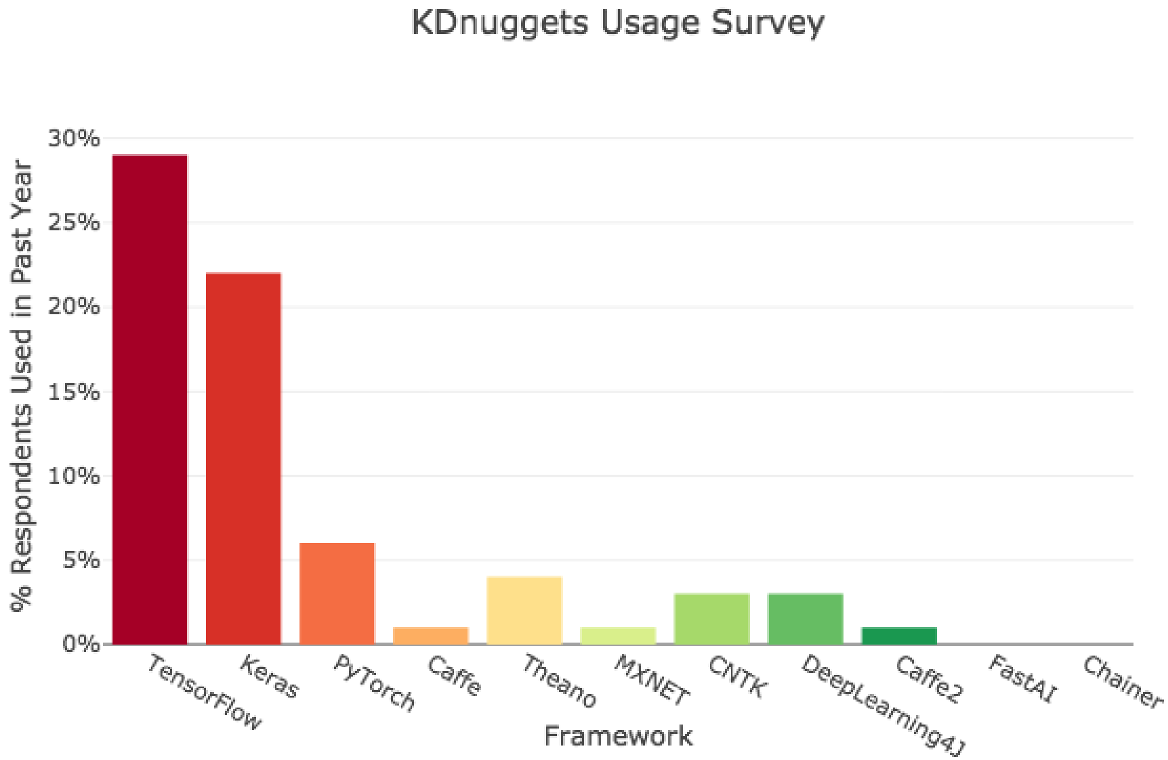
5. Runs on CPU, GPU or TPU.



Keras

Keras is a very popular API. Large adoption in both industry and academia.

What Analytics, Big Data, Data Science, Machine Learning software you used in the past 12 months for a real project?



https://www.kdnuggets.com/2017/05/poll-analytics-data-science-machine-learning-software-leaders.html

Keras – Sequential Model

1. The core data structure of Keras is a model, which is a way to organize layers.

A model is understood as a sequence or a graph of standalone, fully configurable modules that can be plugged together with as few restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions and regularization schemes are all standalone modules that you can combine to create new models.

2. The standard type of model is the Sequential model, a linear stack of layers.
3. You can create a Sequential model by passing a list of layer instances to the constructor:

```
import tensorflow as tf

# create instance of Sequential model
model = tf.keras.models.Sequential()
```

Keras – Dense Layer

1. The most common type of layer in Keras is a Dense layer (**tf.keras.layers.Dense**), which is a fully connected neural network layer.
2. In the code below we first add a fully connected layer of neurons containing 512 neurons each with a relu activation function (note the dense layer in this example assumes a flattened input shape).
3. We then add a fully connected Softmax layer.
4. You can continue to add as many layers as you want to your network.

```
import tensorflow as tf

# create instance of Sequential model
model = tf.keras.models.Sequential()

model = Sequential()
model.add( tf.keras.layers.Dense(300, activation=tf.nn.relu)
model.add(tf.keras.layers.Dense(10, activation=tf.nn.softmax))
```

Keras

1. Notice we can also create an instance of the Sequential model and pass a list of layers as parameters.
2. This is commonly used shortcut.

```
import tensorflow as tf

model = tf.keras.models.Sequential( [
  tf.keras.layers.Dense(300, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
] )
```

Input Shape

1. Your Keras model **needs to know what input shape** it should expect.

2. For this reason, the **first layer** in a Sequential model needs to receive information about its **input shape**.

    - You can pass an input_shape argument to the first layer in your model. This is just a tuple of integers that specify the shape (The presence of a None as a dimension indicates that any positive number may be expected along that dimension).

    - Some 2D layers, such as Dense, also support the specification of their input shape via the argument input_dim.

Keras

1. Notice in this example we specify that the first layer should expect a rank 1 array that contains 784 values (therefore we don’t consider the batch size or number of instances when creating the model).

```
import tensorflow as tf

model = tf.keras.models.Sequential( [
  tf.keras.layers.Dense(512, activation=tf.nn.relu, input_shape=(784,)),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
] )
```

Model Compilation

1. Before training a model, you need to set up the details around the loss function, the optimizer and the metrics you want to use. In Keras this is done via the **compile method**. It receives three **arguments**:

    - **A loss function**. This is the cost function that the model will try to minimize. It can be the string identifier of an existing loss function (such as categorical_crossentropy or mse) or it can be an instance of an objective function. Full list of loss functions available at tf.keras.losses

    - **An optimizer**. This could be the string identifier of an existing optimizer or an instance of the Optimizer class. You can find a list of the available optimizers at tf.keras.optimizers (SGD, Adam, RMSProp, etc).

    - **A list of metrics**. For any classification problem you will want to set this to metrics=['accuracy']. A metric could be the string identifier of an existing metric or a custom metric function. Again a full list of metrics is available at tf.keras.metrics.

Keras

1. To illustrate this process we are going to use the inbuilt mnist dataset. Remember the training data contains 60,000 28*28 pixels.
2. Notice we normalize the feature values of the mnist dataset.
3. The original shape of the training feature data is (60000, 28, 28).
4. We need to reshape this to be a 2D data structure. We reshape the training data so that it is now (60000, 784).
5. The training data now contains 60000 rows and 784 columns. Therefore, the input layer to the neural network will have 784 values.

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

print (x_train.shape)

# Reshape so that each individual row is an image
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

print (x_train.shape)
```

1. Notice when we compile our model we select the adam optimizer, accuracy is our metric and our loss function is **sparse_categorical_crossentropy** (note we use the sparse loss function because our labels are integer values and not one-hot-encoded).

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(512, activation=tf.nn.relu,
                                  input_shape=(784,)),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])
```

Training in Keras

1. The fit function in Keras facilitates the training of our model and allows us to train our model for a fixed number of iterations.

1. The main arguments are as follows:
   - x: A Tensor or NumPy array of training data.
   - y: A Tensor or NumPy array of target (label) data.
   - **batch_size**: Integer or None. Number of samples per gradient update. If unspecified, batch_size will default to 32.
   - **epochs**: Integer. Number of epochs to train the model. Remember an epoch is an iteration over the entire x and y data provided.
   - **validation_split**: A float value between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch.
   - **validation_data**: Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. validation_data will override validation_split. validation_data could be: - tuple (x_val, y_val)

Finally once the model has been trained we can then use the **<u>evaluate function</u>** to determine the **<u>loss value & metrics values</u>** for the model on the test data.

The evaluate function just takes in test features and test labels.

Notice in this example we train our model by providing training features and associated labels. We also specify 5 epochs.

Finally we evaluate the model using the test data and print out the results.

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(512, activation=tf.nn.relu,
                                    input_shape=(784,)),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

results = model.evaluate(x_test, y_test)

print (results)
```

Keras

```
(60000, 28, 28)
(60000, 784)
Epoch 1/5
60000/60000 [==============================] - 3s 46us/step - loss: 0.2000 - acc: 0.9416
Epoch 2/5
60000/60000 [==============================] - 3s 46us/step - loss: 0.0808 - acc: 0.9752
Epoch 3/5
60000/60000 [==============================] - 4s 65us/step - loss: 0.0527 - acc: 0.9831
Epoch 4/5
60000/60000 [==============================] - 5s 76us/step - loss: 0.0366 - acc: 0.9884
Epoch 5/5
60000/60000 [==============================] - 4s 74us/step - loss: 0.0279 - acc: 0.9910
10000/10000 [==============================] - 0s 31us/step

[0.06713101949844276, 0.9789]
```

1. Notice the last line prints out the **loss** on the test data and the **accuracy** on the test data.

```
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])


model.fit(x_train, y_train, epochs=5)

results = model.evaluate(x_test, y_test)

print (results)
```

Keras

In this example, we have specified a validation split of 0.1. We will see this reflected in the output from the training process.

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

model = tf.keras.models.Sequential([
   tf.keras.layers.Dense(512, activation=tf.nn.relu, input_shape=(784,)),
   tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
           loss='sparse_categorical_crossentropy',
           metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5, validation_split=0.1)
results = model.evaluate(x_test, y_test)

print (results)
```

Keras

In this example, we have specified a validation split of 0.1. We will see this reflected in the output from the training process.

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/5
54000/54000 [==============================] - 3s 62us/step - loss: 0.2161 - acc: 0.9362 - val_loss: 0.0962 - val_acc: 0.9743
Epoch 2/5
54000/54000 [==============================] - 3s 48us/step - loss: 0.0844 - acc: 0.9743 - val_loss: 0.0891 - val_acc: 0.9728
Epoch 3/5
54000/54000 [==============================] - 3s 47us/step - loss: 0.0569 - acc: 0.9832 - val_loss: 0.0798 - val_acc: 0.9752
Epoch 4/5
54000/54000 [==============================] - 3s 53us/step - loss: 0.0382 - acc: 0.9877 - val_loss: 0.0705 - val_acc: 0.9812
Epoch 5/5
54000/54000 [==============================] - 3s 48us/step - loss: 0.0287 - acc: 0.9906 - val_loss: 0.0738 - val_acc: 0.9793
10000/10000 [==============================] - 0s 20us/step
[0.07355271206417 82, 0.9777]
```

Predict Function

1. Rather than using the model.evaluate function we can use the **model.predict** function.

2. The model.predict function will output the probability scores for each class for each test instance. In this example the predict function would have produced an array 10000 * 10

3. We can obtain the label with the maximum probability using **np.argmax**.

```
import tensorflow as tf
import numpy as np

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(512, activation=tf.nn.relu,
input_shape=(784,)),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])


model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=3)
results = model.predict(x_test)

print (results[0])
print ("Predicted Class is ",np.argmax(results[0]))
```

Predict Function

1. Rather than using the model.evaluate function we can use the **model.predict** function.
2. The model.predict function will output the probability scores for each class for

```
import tensorflow as tf
import numpy as np

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(512, activation=tf.nn.relu,
```

```
Epoch 1/3
60000/60000 [==============================] - 5s 87us/step - loss: 0.2001 - acc: 0.9416
Epoch 2/3
60000/60000 [==============================] - 5s 79us/step - loss: 0.0803 - acc: 0.9753
Epoch 3/3
60000/60000 [==============================] - 5s 81us/step - loss: 0.0526 - acc: 0.9833
[1.3871913e-08 4.4194465e-08 6.4126448e-06 4.8022767e-04 8.7549195e-12
 3.3042408e-08 1.6389102e-11 9.9949980e-01 2.2796728e-06 1.1177561e-05]
Predicted Class is  7
```

Scikit Metrics

- It is worth noting that we can still using our metrics package from Scikit Learn to obtain more detail on individual results.
- For example in this code we easily generate a **confusion matrix** for the result produced by our neural network.

- Notice that we apply **np.argmax** to obtain the **index** with the highest probability for each row of the resultsProb data.

- The confusion matrix is passed the true labels and all the predicted integer labels from the neural network.

```
import tensorflow as tf
from sklearn.metrics import confusion_matrix
import numpy as np

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

model = tf.keras.models.Sequential([
   tf.keras.layers.Dense(512, activation=tf.nn.relu, input_shape=(784,)),
   tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',  metrics=['accuracy'])

model.fit(x_train, y_train, epochs=6)

resultsProb = model.predict(x_test)

# calculate predicted results from probabilities (horizontal axis)
results = np.argmax(resultsProb, axis =1)

print(confusion_matrix(y_test, results))
```

Scikit Metrics

```
Epoch 1/6
60000/60000 [==============================] - 5s 91us/step - loss: 0.2002 - acc: 0.9400
Epoch 2/6
60000/60000 [==============================] - 5s 88us/step - loss: 0.0803 - acc: 0.9756
Epoch 3/6
60000/60000 [==============================] - 5s 88us/step - loss: 0.0519 - acc: 0.9839
Epoch 4/6
60000/60000 [==============================] - 5s 88us/step - loss: 0.0362 - acc: 0.9886
Epoch 5/6
60000/60000 [==============================] - 5s 83us/step - loss: 0.0271 - acc: 0.9910
Epoch 6/6
60000/60000 [==============================] - 5s 81us/step - loss: 0.0198 - acc: 0.9935
[[ 973    1    0    0    1    0    1    1    2    1]
 [    0 1125    2    2    0    0    2    0    4    0]
 [    4    0 1010    4    2    0    1    5    5    1]
 [    0    0    4  995    0    2    0    3    3    3]
 [    1    1    5    0  959    0    2    2    2   10]
 [    2    0    0   12    1  867    4    1    5    0]
 [    7    3    0    1    3    1  942    0    1    0]
 [    2    3    7    3    0    0    0  998    9    6]
 [    0    0    4    4    4    2    1    1  957    1]
 [    3    2    0    2    7    2    0    2    6  985]]
```

Recap

So to recap the standard workflow involves:

- Creating a **sequential model**, which typically consists of **dense** layers
- **Compiling** the model you creating by specifying using the compile function, which allows you to specify the loss function, the optimizer and the metric(s) you want to use.
- The **fit** function then trains the neural network.
- You can then finally **evaluate** your model on a test set.

Next will look at some of the available layers in a little more detail.

- Dense layers
- Dropout
- Flatten

Dense Layers

1. As previously mentioned the Dense layer provides the regular densely-connected NN layer.

2. The following are most common parameters for the Dense Layer:
   - **units**: Specify the number of units(neurons) in this layer.
   - **activation**: Activation function to use. You can use a range of standard activation functions here such as ReLu, Sigmoid, Tanh, etc (see tf.keras.activations)
   - **use_bias**: Boolean, whether the layer uses a bias vector.
   - **kernel_initializer**: Initializer for the kernel weights matrix . The Initializations define the way to set the initial random weights of Keras layers (see tf.keras.initializers).
   - **bias_initializer**: Initializer for the bias vector (see next slide for more detail).
   - **kernel_regularizer**: Regularizer function applied to the kernel weights matrix bias_regularizer: Regularizer function applied to the bias vector (see tf.keras.regularizers).

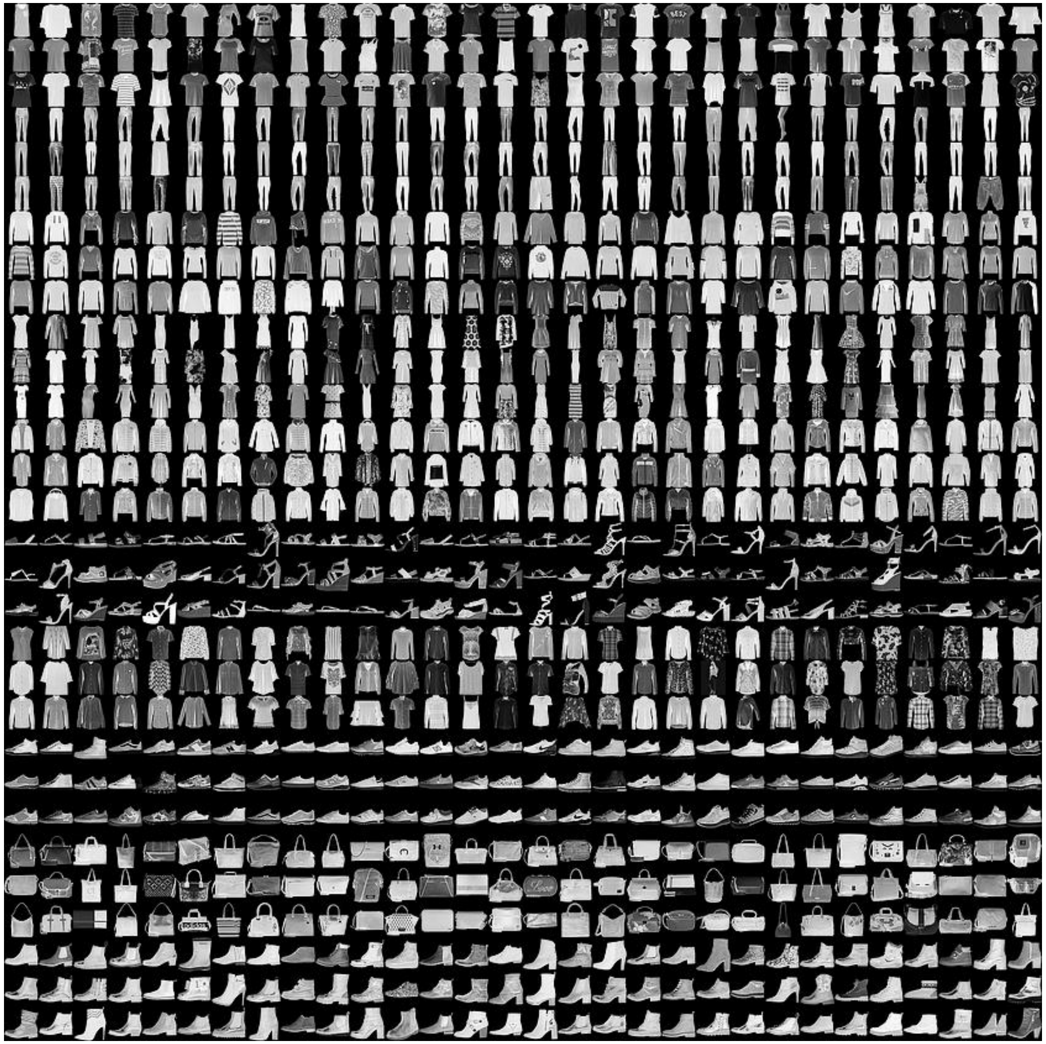
Dropout and Flatten

- As we have seen Dropout consists of randomly setting a fraction of input units to 0 at each update during training time, which helps prevent overfitting.
- The dropout layer applies Dropout to the input layer.
- Dropout is provided in [tf.keras.layers.Dropout]
- The main parameters are
  - **rate**: float between 0 and 1. Fraction of the input units to drop.
  - **seed**: A Python integer to use as random seed.

- The Flatten layer is a straight forward layer, it's only purpose is to flatten whatever input it receives into a flat 1D tensor.
  - It can be found at tf.keras.layers.Flatten().
  - If a flatten layer is the first layer of your network then you should specify the input_shape

Fashion MNist

1. Fashion-MNIST is a dataset that contains a training set of 60,000 examples and a test set of 10,000 examples.
2. Each example is a 28x28 grayscale image, associated with a label from 10 classes.
3. Fashion-MNIST serves as a replacement for the original MNIST dataset for benchmarking machine learning algorithms, which is considered too easy.



Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Keras

- The shape of the training data is (60000, 28, 28)

- The first layer in our network will **tf.keras.layers.Flatten**, which transforms the format of the images from a 2d-array (of 28 by 28 pixels), to a 1d-array of 28 * 28 = 784 pixels.

- 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, neural layers. The first Dense layer has 128 nodes (or neurons).
- The second (and last) layer is a 10-node softmax layer—this returns an array of 10 probability scores that sum to 1. Each node contains a score that indicates the probability that the current image belongs to one of the 10 classes.

- Notice we can call **model.predict** function and it will output ten probability scores for each class. We can obtain the label with the maximum probability using np.argmax

```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
train_images = train_images / 255.0
test_images = test_images / 255.0

print (train_images.shape, train_labels.shape)

model = tf.keras.Sequential([
   tf.layers.Flatten(input_shape=(28, 28)),
   tf.layers.Dense(128, activation=tf.nn.relu),
   tf.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
           loss='sparse_categorical_crossentropy',
           metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=20,  validation_split=0.1)
```

```
54000/54000 [==============================] - 7s 124us/sample - loss: 0.2237 - acc: 0.9157 -
val_loss: 0.3438 - val_acc: 0.8858
Epoch 13/20
54000/54000 [==============================] - 7s 122us/sample - loss: 0.2195 - acc: 0.9187 -
val_loss: 0.3269 - val_acc: 0.8860
Epoch 14/20
54000/54000 [==============================] - 7s 123us/sample - loss: 0.2101 - acc: 0.9222 -
val_loss: 0.3302 - val_acc: 0.8903
Epoch 15/20
54000/54000 [==============================] - 7s 137us/sample - loss: 0.2055 - acc: 0.9228 -
val_loss: 0.3414 - val_acc: 0.8895
Epoch 16/20
54000/54000 [==============================] - 7s 138us/sample - loss: 0.1996 - acc: 0.9263 -
val_loss: 0.3443 - val_acc: 0.8877
Epoch 17/20
54000/54000 [==============================] - 6s 117us/sample - loss: 0.1950 - acc: 0.9268 -
val_loss: 0.3312 - val_acc: 0.8865
Epoch 18/20
54000/54000 [==============================] - 6s 120us/sample - loss: 0.1882 - acc: 0.9305 -
val_loss: 0.3292 - val_acc: 0.8897
Epoch 19/20
54000/54000 [==============================] - 6s 118us/sample - loss: 0.1828 - acc: 0.9307 -
val_loss: 0.3562 - val_acc: 0.8852
Epoch 20/20
54000/54000 [==============================] - 7s 121us/sample - loss: 0.1773 - acc: 0.9336 -
val_loss: 0.3429 - val_acc: 0.8905
```

Visualizing Accuracy and Loss in Keras

1. The <u>fit function</u> returns a <u>**History**</u> object.

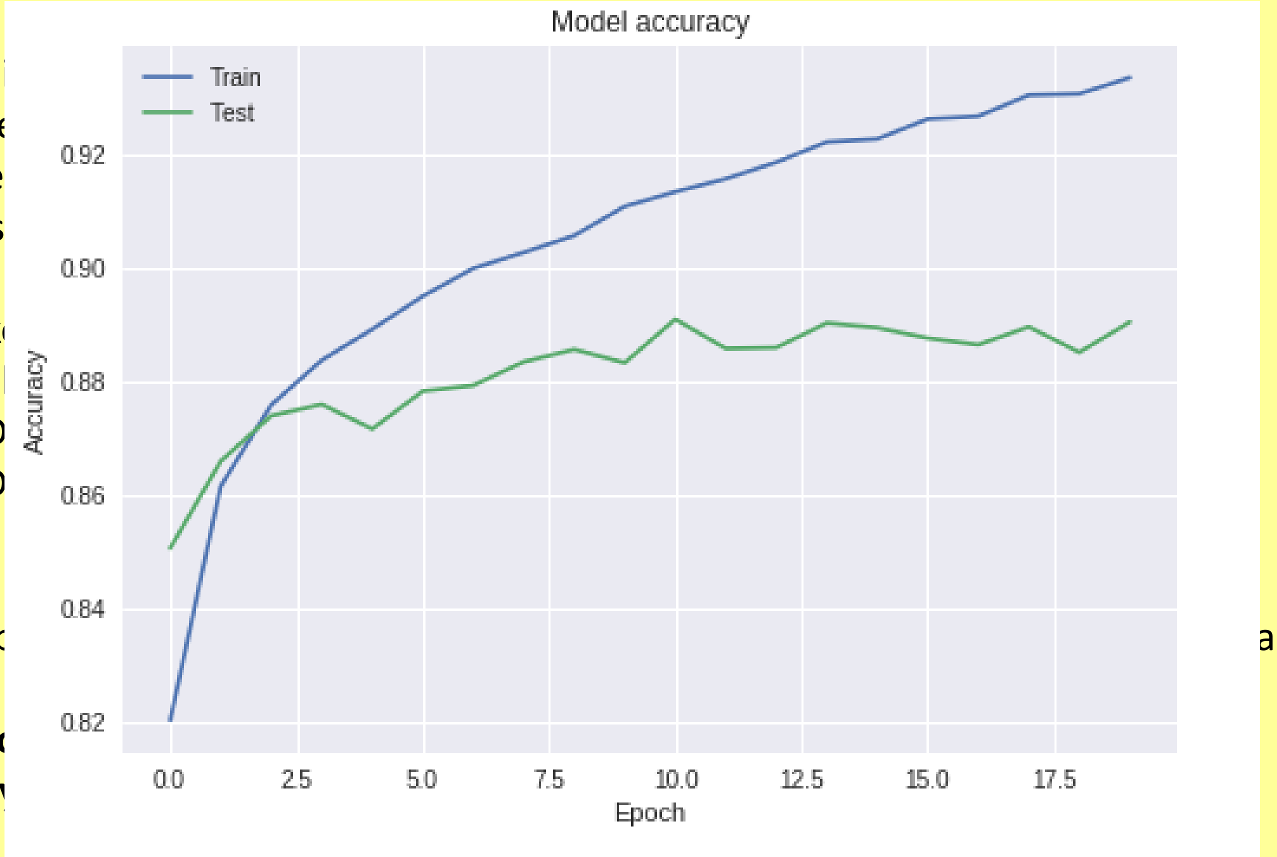
2. It records training metrics for each epoch. This includes the <u>**loss**</u> and the <u>**accuracy**</u> for the training set as well as the <u>**loss**</u> and <u>**accuracy**</u> for the validation dataset, if one is set.

3. History is a dictionary data structure. You can easily see the data available to you by printing out print(history.history.keys())

```
dict_keys(['loss', 'acc', 'val_loss', 'val_acc'])
```

```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
train_images = train_images / 255.0
test_images = test_images / 255.0

model = tf.keras.Sequential([
   tf.layers.Flatten(input_shape=(28, 28)),
   tf.layers.Dense(128, activation=tf.nn.relu),
   tf.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=2,  validation_split=0.1)
print(history.history.keys())

import matplotlib.pyplot as plt
plt.plot(history.history['acc']) # Plot training & validation accuracy values
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

```
import tensorflow as tf

fashion_mni
(train_image
train_image
test_images

model = tf.k
    tf.layers.F
    tf.layers.D
    tf.layers.D
])

model.comp                                                    accuracy'])

history = mo
print(histor
```




```
import matplotlib.pyplot as plt
plt.plot(history.history['acc']) # Plot training & validation accuracy values
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

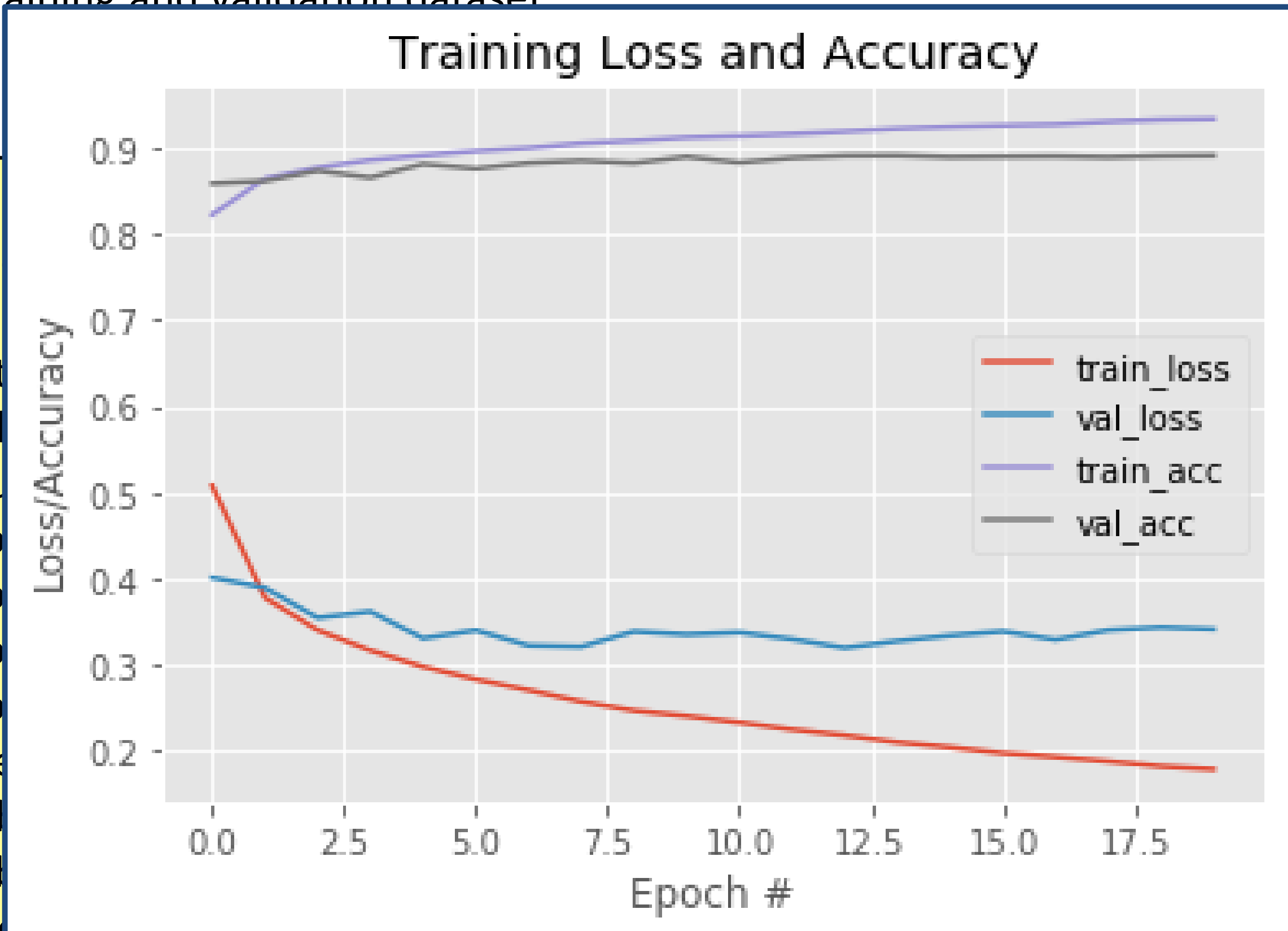
Visualizing Accuracy and Loss in Keras

1. However most of the time it is very useful to visualize the loss and accuracy for both the training and validation dataset.

```
# … code same as previous slide.

import matplotlib.pyplot as plt
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 20), history.history["loss"], label="train_loss")
plt.plot(np.arange(0, 20), history.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 20), history.history["acc"], label="train_acc")
plt.plot(np.arange(0, 20), history.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
```

Visualizing Accuracy and Loss in Keras

1. However most of the time it is very useful to visualize the loss and accuracy for both the training and validation dataset

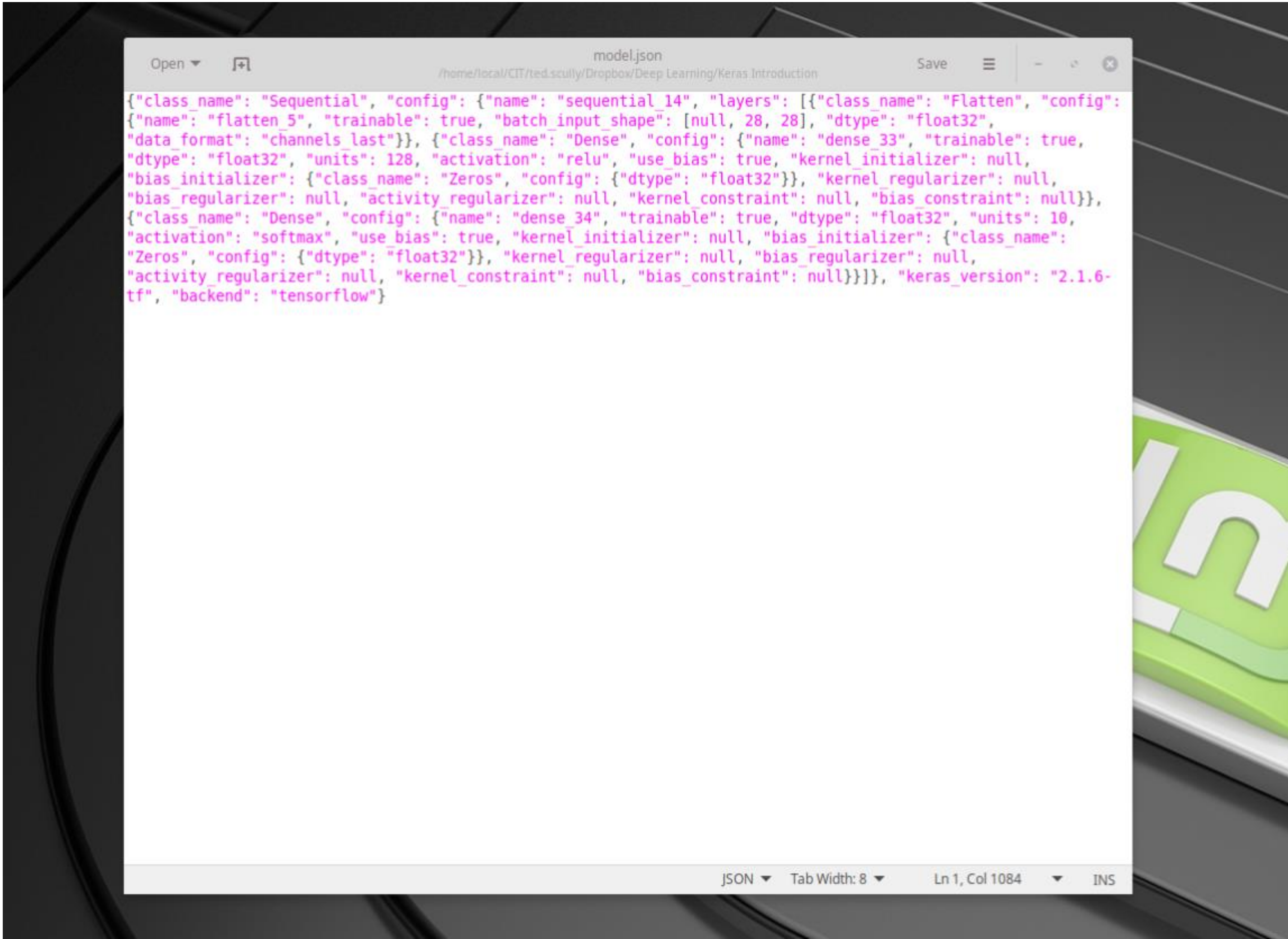



```
# … co
import
plt.styl
plt.figu
plt.plo
plt.plo
plt.plo
plt.plo
plt.title
plt.xlal
plt.ylal
plt.lege ()
```

Saving and Loading Keras Models to Disk

1. When saving models in Keras it separates the architecture of the network from the network weights.
2. It allows us to save and load the **weights** of a network to a **HDF5 file**
3. In contrast the **architecture** is saved to a **JSON file** and can then be loaded again from this file.
4. In the code below we will save the model we have worked on in the previous slides.

```
import tensorflow as tf

# ….. Model development from previous slides


# serialize model to a JSON file
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)

# serialize the model weights to a HDF5 file
model.save_weights("model.h5")
```



model.json
/home/local/CIT/ted.scully/Dropbox/Deep Learning/Keras Introduction

{"class_name": "Sequential", "config": {"name": "sequential_14", "layers": [{"class_name": "Flatten", "config": {"name": "flatten_5", "trainable": true, "batch_input_shape": [null, 28, 28], "dtype": "float32", "data_format": "channels_last"}}, {"class_name": "Dense", "config": {"name": "dense_33", "trainable": true, "dtype": "float32", "units": 128, "activation": "relu", "use_bias": true, "kernel_initializer": null, "bias_initializer": {"class_name": "Zeros", "config": {"dtype": "float32"}}, "kernel_regularizer": null, "bias_regularizer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}}, {"class_name": "Dense", "config": {"name": "dense_34", "trainable": true, "dtype": "float32", "units": 10, "activation": "softmax", "use_bias": true, "kernel_initializer": null, "bias_initializer": {"class_name": "Zeros", "config": {"dtype": "float32"}}, "kernel_regularizer": null, "bias_regularizer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}}]}, "keras_version": "2.1.6-tf", "backend": "tensorflow"}

JSON Tab Width: 8 Ln 1, Col 1084 INS

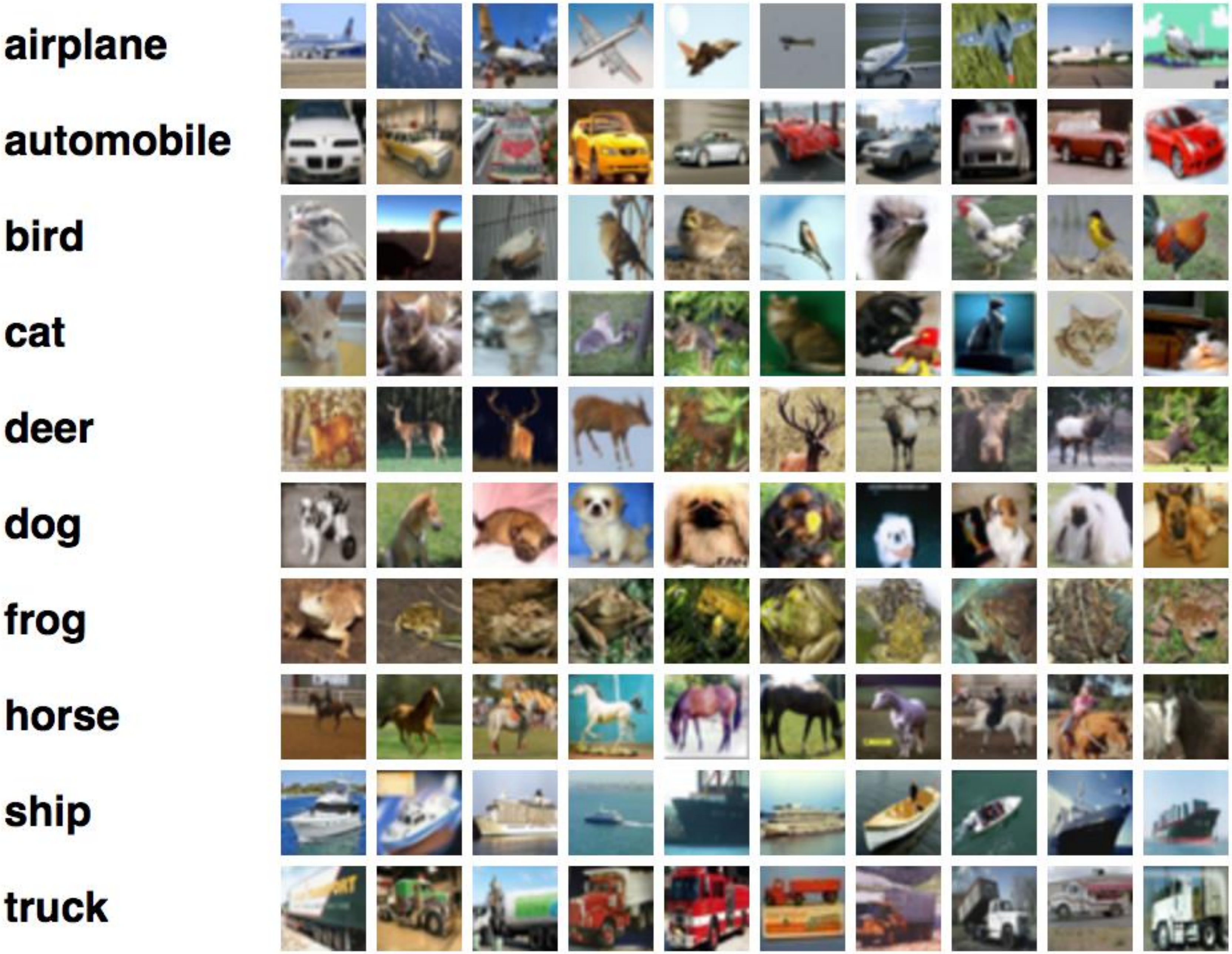
Saving and Loading Keras Models to Disk

1. Notice in the code below we must **first load the model** that was saved to the JSON file.
2. Once this is done we can load the weights to the new model and use the model as normal.

```
# load json file into keras model
model_file = open('model.json', 'r')
loaded_model_json = model_file.read()
model_file.close()
loaded_model = tf.keras.models.model_from_json(loaded_model_json)

# load weights into new model
loaded_model.load_weights("model.h5")

predictions = loaded_model.predict(test_images)
print (predictions.shape)

print (predictions[1], np.argmax(predictions[1]) , test_labels[1])
```

CIFAR 10 Dataset

- When it comes to computer vision and machine learning, the MNIST dataset is the classic definition of a “benchmark” dataset, one that is too easy to obtain high accuracy results on, and not representative of the images we’ll see in the real world.
- While the Fashion MNIST dataset is a little more challenging it is still not really reflective of real world image classification.
- For a more challenging benchmark dataset, we commonly we can use CIFAR-10, a collection of 60,000, 32×32 RGB images, thus implying that each image in the dataset is represented by $32 \times 32 \times 3 = 3{,}072$ integers.
- As the name suggests, CIFAR-10 consists of 10 classes, including airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.
- Each <u>class is evenly represented with 6,000 images </u>per class.
- When training and evaluating a machine learning model on CIFAR-10, it’s typical to use the predefined data splits by the authors and use 50,000 images for training and 10,000 for testing.


airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

CIFAR 10 Dataset

1. CIFAR-10 is substantially harder than the MNIST dataset.

2. The challenge comes from the dramatic variance in how objects appear. For example, we can no longer assume that an image containing a green pixel at a given (x, y)-coordinate is a frog. This pixel could be a background of a forest that contains a deer. Or it could be the color of a green car or truck.

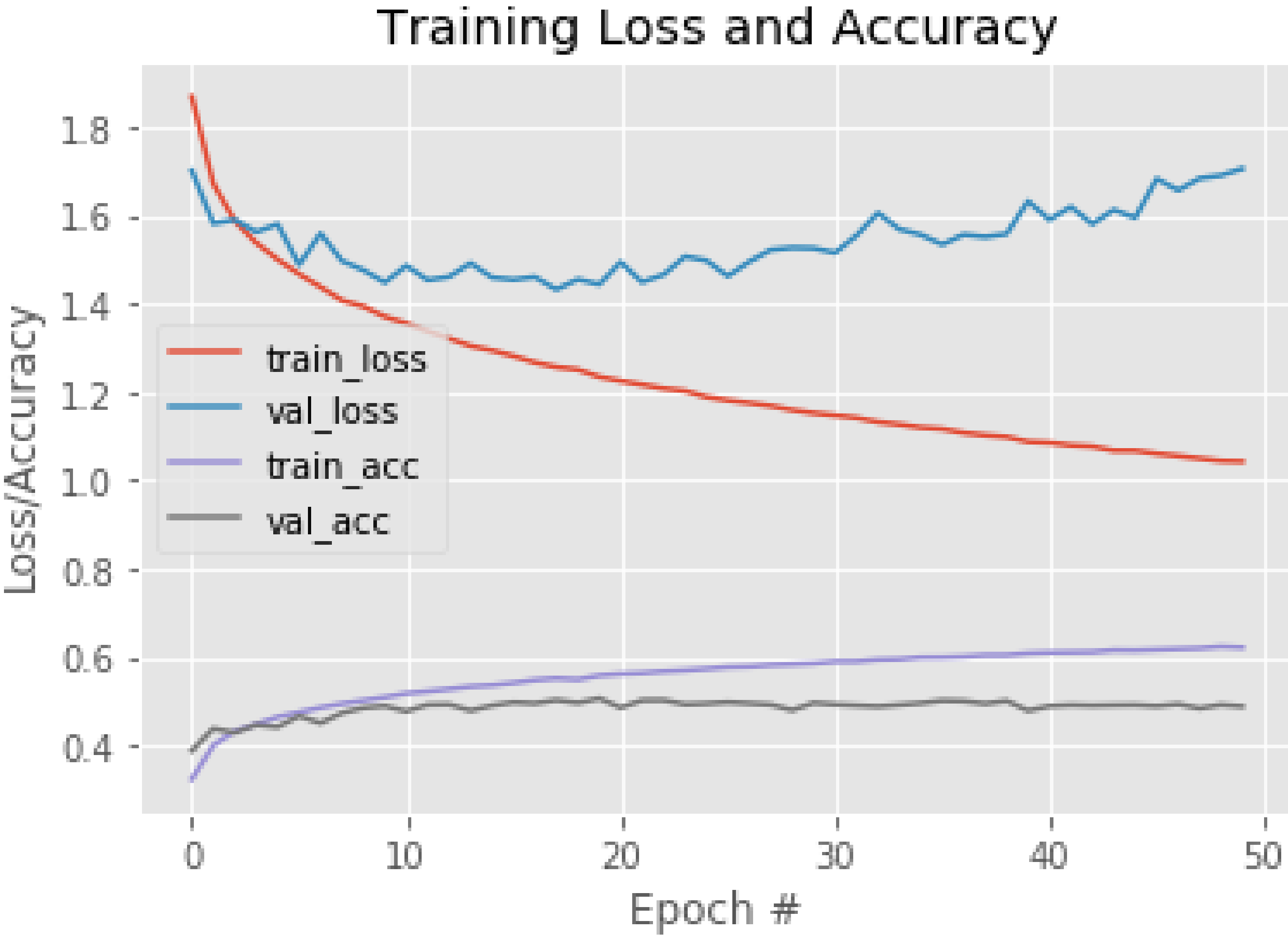
3. These assumptions are a stark contrast to the MNIST dataset, where the network can learn assumptions regarding the **spatial distribution of pixel intensities**. For example, the spatial distribution of foreground pixels of a 1 is substantially different than a 0 or a 5.

4. This type of variance exhibited in object appearance in CIFAR10 makes applying a series of fully-connected layers much more challenging.

5. As you'll see in the following code, standard fully-connected layer networks are not suited for this type of image classification.

```
import tensorflow as tf
import matplotlib.pyplot as plt

num_epochs = 50
cifar = tf.keras.datasets.cifar10
(x_train, y_train),(x_test, y_test) = cifar.load_data()
x_train, x_test = x_train / 255.0, x_test / 255


model = tf.keras.models.Sequential([
    tf.layers.Flatten(input_shape=(32, 32, 3)),
    tf.keras.layers.Dense(1024, activation=tf.nn.relu),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(x_train, y_train, epochs=num_epochs,  validation_data=(x_test, y_test))

plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, num_epochs), history.history["loss"], label="train_loss")
plt.plot(np.arange(0, num_epochs), history.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, num_epochs), history.history["acc"], label="train_acc")
plt.plot(np.arange(0, num_epochs), history.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend
```

1.


Training Loss and Accuracy
Loss/Accuracy
Epoch #
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0
10
20
30
40
50
train_loss
val_loss
train_acc
val_acc

CIFAR 10 Dataset

1. Clearly the network we have trained on the previous slide is badly overfitting on the training data.

2. We could certainly consider optimizing our hyperparameters further, in particular, experimenting with varying learning rates and increasing both the depth and the number of nodes in the network, but we would be fighting for meager gains.

3. The reality is that basic feedforward network with strictly fully-connected layers are not suitable for challenging image datasets. For that, we need a more advanced approach: Convolutional Neural, which we will be covering shorty.