Capstone_Project

November 23, 2020

1 Capstone Project

1.1 Image classifier for the SVHN dataset

1.1.1 Instructions

In this notebook, you will create a neural network that classifies real-world images digits. You will use concepts from throughout this course in building, training, testing, validating and saving your Tensorflow classifier model.

This project is peer-assessed. Within this notebook you will find instructions in each section for how to complete the project. Pay close attention to the instructions as the peer review will be carried out according to a grading rubric that checks key parts of the project instructions. Feel free to add extra cells into the notebook as required.

1.1.2 How to submit

When you have completed the Capstone project notebook, you will submit a pdf of the notebook for peer review. First ensure that the notebook has been fully executed from beginning to end, and all of the cell outputs are visible. This is important, as the grading rubric depends on the reviewer being able to view the outputs of your notebook. Save the notebook as a pdf (you could download the notebook with File -> Download .ipynb, open the notebook locally, and then File -> Download as -> PDF via LaTeX), and then submit this pdf for review.

1.1.3 Let's get started!

We'll start by running some imports, and loading the dataset. For this project you are free to make further imports throughout the notebook as you wish.

[1]:
```
import tensorflow as tf
from scipy.io import loadmat
```

For the capstone project, you will use the SVHN dataset. This is an image dataset of over 600,000 digit images in all, and is a harder dataset than MNIST as the numbers appear in the context of natural scene images. SVHN is obtained from house numbers in Google Street View images.

- Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu and A. Y. Ng. "Reading Digits in Natural Images with Unsupervised Feature Learning". NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2011.

The train and test datasets required for this project can be downloaded from here and here. Once unzipped, you will have two files: `train_32x32.mat` and `test_32x32.mat`. You should store these files in Drive for use in this Colab notebook.

Your goal is to develop an end-to-end workflow for building, training, validating, evaluating and saving a neural network that classifies a real-world image into one of ten classes.

[2]:
```
# Run this cell to connect to your Drive folder

from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
```

```
Mounted at /content/gdrive
```

[3]:
```
# Load the dataset from your Drive folder
directory = 'gdrive/MyDrive/TF_Cap'
train = loadmat(f'{directory}/train_32x32.mat')
test = loadmat(f'{directory}/test_32x32.mat')
```

Both `train` and `test` are dictionaries with keys `X` and `y` for the input images and labels respectively.

1.2 1. Inspect and preprocess the dataset

- Extract the training and testing images and labels separately from the train and test dictionaries loaded for you.
- Select a random sample of images and corresponding labels from the dataset (at least 10), and display them in a figure.
- Convert the training and test images to grayscale by taking the average across all colour channels for each pixel. *Hint: retain the channel dimension, which will now have size 1.*
- Select a random sample of the grayscale images and corresponding labels from the dataset (at least 10), and display them in a figure.

[8]:
```
X_train, X_test, y_train, y_test = train['X'], test['X'], train['y'], test['y']
X_test = np.moveaxis(X_test, -1,0)
X_train = np.moveaxis(X_train, -1,0)

print(X_train.shape)
print(X_test.shape)
```

```
(73257, 32, 32, 3)
(26032, 32, 32, 3)
```

[11]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
num_images = 15
indexes = np.random.random_integers(X_train.shape[0],size=num_images)
fig = plt.figure(figsize=(21,9))
```

```
for i,index in enumerate(indexes):
  fig.add_subplot(np.ceil(num_images/5)//1,5,i+1)
  plt.imshow(X_train[index])
```

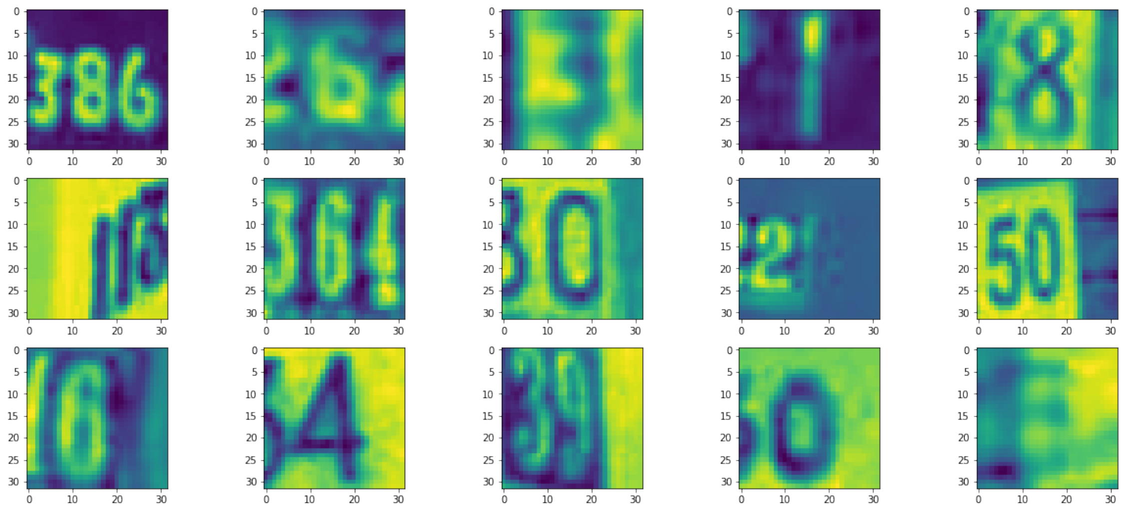
```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:5:
DeprecationWarning: This function is deprecated. Please call randint(1, 73257 +
1) instead
  """
```

[13]:
```
X_train_grey = X_train.mean(axis=3, keepdims=True)
X_test_grey = X_test.mean(axis=3, keepdims=True)

print(X_train_grey.shape)
print(X_test_grey.shape)
```

```
(73257, 32, 32, 1)
(26032, 32, 32, 1)
```

[15]:
```
num_images = 15
indexes = np.random.random_integers(X_train.shape[0],size=num_images)
fig = plt.figure(figsize=(21,9))

for i,index in enumerate(indexes):
  fig.add_subplot(np.ceil(num_images/5)//1,5,i+1)
  plt.imshow(X_train_grey[index,:,:,0])
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:2:
DeprecationWarning: This function is deprecated. Please call randint(1, 73257 +
1) instead
```



```
[16]: np.unique(y_train)
```

```
[16]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10], dtype=uint8)
```

```
[ ]:
```

1.3 2. MLP neural network classifier

- Build an MLP classifier model using the Sequential API. Your model should use only Flatten and Dense layers, with the final layer having a 10-way softmax output.
- You should design and build the model yourself. Feel free to experiment with different MLP architectures. *Hint: to achieve a reasonable accuracy you won't need to use more than 4 or 5 layers.*
- Print out the model summary (using the summary() method)
- Compile and train the model (we recommend a maximum of 30 epochs), making use of both training and validation sets during the training run.
- Your model should track at least one appropriate metric, and use at least two callbacks during training, one of which should be a ModelCheckpoint callback.
- As a guide, you should aim to achieve a final categorical cross entropy training loss of less than 1.0 (the validation loss might be higher).
- Plot the learning curves for loss vs epoch and accuracy vs epoch for both training and validation sets.
- Compute and display the loss and accuracy of the trained model on the test set.

```
[17]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPool2D, Dropout, BatchNormalization
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
```

```
[18]: def get_mlp_model(input_shape):
  model = Sequential([
    Flatten(input_shape=input_shape),
    Dense(64, activation='relu'),
```

```
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(16, activation='relu'),
    Dense(10, activation='softmax')
  ])
  model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['accuracy'])
  return model
```

[19]:
```
checkpoint = ModelCheckpoint(
    filepath='model_checkpoints/mlp_best',
    save_weights_only=True,
    save_best_only=True,
    monitor='val_loss',
    verbose=1
)

early_stop = EarlyStopping(
    monitor='val_loss',
    patience=5
)
```

[30]:
```
model = get_mlp_model((32,32,1))
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_10 (Dense)             (None, 64)                65600
_________________________________________________________________
dense_11 (Dense)             (None, 64)                4160
_________________________________________________________________
dense_12 (Dense)             (None, 32)                2080
_________________________________________________________________
dense_13 (Dense)             (None, 16)                528
_________________________________________________________________
dense_14 (Dense)             (None, 10)                170
=================================================================
Total params: 72,538
Trainable params: 72,538
Non-trainable params: 0
_________________________________________________________________
```

[21]:
```
X_train_grey = X_train_grey/255.0
X_test_grey = X_test_grey/255.0
```

```
y_test = y_test -1
y_train = y_train -1
```

[31]:
```
loss, accuracy = model.evaluate(X_test_grey, y_test)
print(f"Val/Test Loss: {loss}")
print(f'Val/Test Accuracy {accuracy}')
```

```
814/814 [==============================] - 2s 3ms/step - loss: 2.3119 -
accuracy: 0.1591
Val/Test Loss: 2.311924457550049
Val/Test Accuracy 0.15907344222068787
```

[32]:
```
history = model.fit(X_train_grey, y_train,
                    callbacks=[checkpoint, early_stop],
                    validation_data=(X_test_grey, y_test),
                    epochs=30,
                    batch_size=256,
                    verbose=2
                    )
```

```
Epoch 1/30

Epoch 00001: val_loss did not improve from 1.02200
287/287 - 1s - loss: 2.1799 - accuracy: 0.2065 - val_loss: 2.0099 -
val_accuracy: 0.2838
Epoch 2/30

Epoch 00002: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.8675 - accuracy: 0.3327 - val_loss: 1.6889 -
val_accuracy: 0.4150
Epoch 3/30

Epoch 00003: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.5481 - accuracy: 0.4651 - val_loss: 1.5510 -
val_accuracy: 0.4892
Epoch 4/30

Epoch 00004: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.4237 - accuracy: 0.5221 - val_loss: 1.4728 -
val_accuracy: 0.5257
Epoch 5/30

Epoch 00005: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.3398 - accuracy: 0.5617 - val_loss: 1.3945 -
val_accuracy: 0.5585
Epoch 6/30
```

```
Epoch 00006: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.2561 - accuracy: 0.5998 - val_loss: 1.3407 -
val_accuracy: 0.5857
Epoch 7/30

Epoch 00007: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.1937 - accuracy: 0.6251 - val_loss: 1.2989 -
val_accuracy: 0.6039
Epoch 8/30

Epoch 00008: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.1576 - accuracy: 0.6375 - val_loss: 1.2377 -
val_accuracy: 0.6242
Epoch 9/30

Epoch 00009: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.1279 - accuracy: 0.6492 - val_loss: 1.2611 -
val_accuracy: 0.6128
Epoch 10/30

Epoch 00010: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.0977 - accuracy: 0.6610 - val_loss: 1.2084 -
val_accuracy: 0.6334
Epoch 11/30

Epoch 00011: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.0699 - accuracy: 0.6688 - val_loss: 1.1887 -
val_accuracy: 0.6400
Epoch 12/30

Epoch 00012: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.0455 - accuracy: 0.6782 - val_loss: 1.1601 -
val_accuracy: 0.6517
Epoch 13/30

Epoch 00013: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.0400 - accuracy: 0.6788 - val_loss: 1.1321 -
val_accuracy: 0.6595
Epoch 14/30

Epoch 00014: val_loss did not improve from 1.02200
287/287 - 1s - loss: 1.0060 - accuracy: 0.6902 - val_loss: 1.1068 -
val_accuracy: 0.6668
Epoch 15/30

Epoch 00015: val_loss did not improve from 1.02200
287/287 - 1s - loss: 0.9902 - accuracy: 0.6965 - val_loss: 1.1096 -
val_accuracy: 0.6701
```

```
Epoch 16/30

Epoch 00016: val_loss did not improve from 1.02200
287/287 - 1s - loss: 0.9758 - accuracy: 0.7014 - val_loss: 1.1157 -
val_accuracy: 0.6635
Epoch 17/30

Epoch 00017: val_loss did not improve from 1.02200
287/287 - 1s - loss: 0.9661 - accuracy: 0.7045 - val_loss: 1.1156 -
val_accuracy: 0.6675
Epoch 18/30

Epoch 00018: val_loss did not improve from 1.02200
287/287 - 1s - loss: 0.9475 - accuracy: 0.7114 - val_loss: 1.0560 -
val_accuracy: 0.6868
Epoch 19/30

Epoch 00019: val_loss did not improve from 1.02200
287/287 - 1s - loss: 0.9331 - accuracy: 0.7147 - val_loss: 1.0786 -
val_accuracy: 0.6730
Epoch 20/30

Epoch 00020: val_loss did not improve from 1.02200
287/287 - 1s - loss: 0.9263 - accuracy: 0.7158 - val_loss: 1.0589 -
val_accuracy: 0.6848
Epoch 21/30

Epoch 00021: val_loss did not improve from 1.02200
287/287 - 1s - loss: 0.9068 - accuracy: 0.7238 - val_loss: 1.1273 -
val_accuracy: 0.6542
Epoch 22/30

Epoch 00022: val_loss did not improve from 1.02200
287/287 - 1s - loss: 0.8977 - accuracy: 0.7275 - val_loss: 1.0389 -
val_accuracy: 0.6953
Epoch 23/30

Epoch 00023: val_loss did not improve from 1.02200
287/287 - 1s - loss: 0.8844 - accuracy: 0.7328 - val_loss: 1.0611 -
val_accuracy: 0.6910
Epoch 24/30

Epoch 00024: val_loss improved from 1.02200 to 1.00193, saving model to
model_checkpoints/mlp_best
287/287 - 1s - loss: 0.8823 - accuracy: 0.7322 - val_loss: 1.0019 -
val_accuracy: 0.7009
Epoch 25/30
```

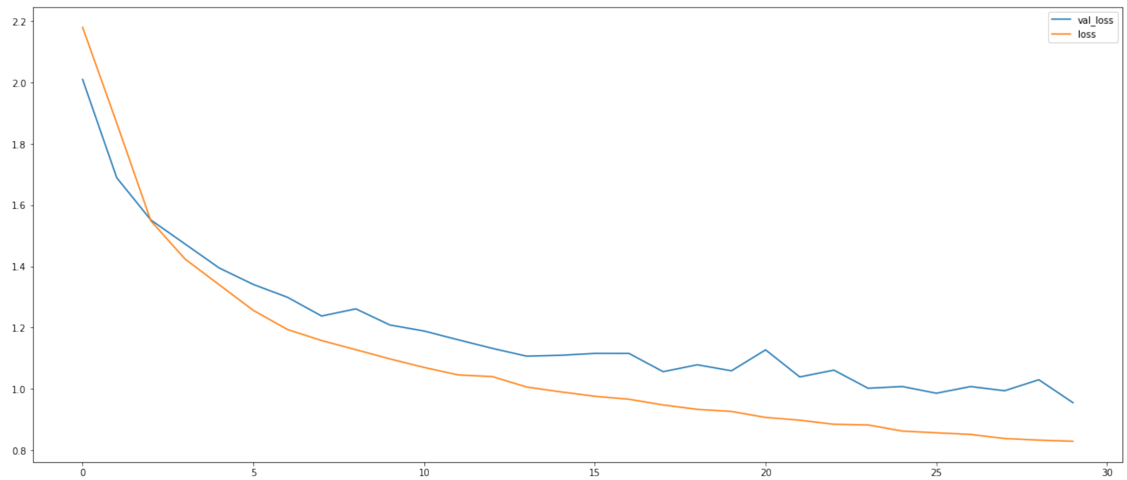
```
Epoch 00025: val_loss did not improve from 1.00193
287/287 - 1s - loss: 0.8623 - accuracy: 0.7390 - val_loss: 1.0076 -
val_accuracy: 0.7028
Epoch 26/30

Epoch 00026: val_loss improved from 1.00193 to 0.98571, saving model to
model_checkpoints/mlp_best
287/287 - 1s - loss: 0.8566 - accuracy: 0.7410 - val_loss: 0.9857 -
val_accuracy: 0.7084
Epoch 27/30

Epoch 00027: val_loss did not improve from 0.98571
287/287 - 1s - loss: 0.8512 - accuracy: 0.7417 - val_loss: 1.0074 -
val_accuracy: 0.7031
Epoch 28/30

Epoch 00028: val_loss did not improve from 0.98571
287/287 - 1s - loss: 0.8379 - accuracy: 0.7451 - val_loss: 0.9939 -
val_accuracy: 0.7087
Epoch 29/30

Epoch 00029: val_loss did not improve from 0.98571
287/287 - 1s - loss: 0.8326 - accuracy: 0.7473 - val_loss: 1.0297 -
val_accuracy: 0.6978
Epoch 30/30

Epoch 00030: val_loss improved from 0.98571 to 0.95490, saving model to
model_checkpoints/mlp_best
287/287 - 1s - loss: 0.8289 - accuracy: 0.7470 - val_loss: 0.9549 -
val_accuracy: 0.7217
```

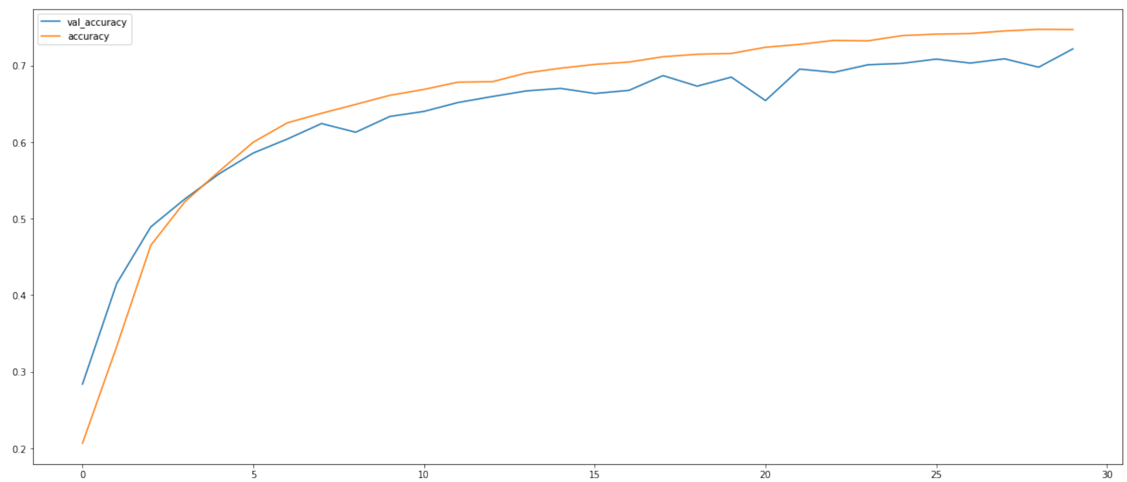
[33]:
```
loss, accuracy = model.evaluate(X_test_grey, y_test)
print(f"Val/Test Loss: {loss}")
print(f'Val/Test Accuracy {accuracy}')
```

```
814/814 [==============================] - 2s 2ms/step - loss: 0.9549 -
accuracy: 0.7217
Val/Test Loss: 0.9548984169960022
Val/Test Accuracy 0.7216886878013611
```

[34]:
```
import pandas as pd
df = pd.DataFrame(history.history)
df[['val_loss','loss']].plot(figsize=(21,9))
df[['val_accuracy','accuracy']].plot(figsize=(21,9))
```

[34]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f824ada59b0>
```





1.4 3. CNN neural network classifier

- Build a CNN classifier model using the Sequential API. Your model should use the Conv2D, MaxPool2D, BatchNormalization, Flatten, Dense and Dropout layers. The final layer should again have a 10-way softmax output.
- You should design and build the model yourself. Feel free to experiment with different CNN architectures. *Hint: to achieve a reasonable accuracy you won't need to use more than 2 or 3 convolutional layers and 2 fully connected layers.)*
- The CNN model should use fewer trainable parameters than your MLP model.
- Compile and train the model (we recommend a maximum of 30 epochs), making use of both training and validation sets during the training run.
- Your model should track at least one appropriate metric, and use at least two callbacks during training, one of which should be a ModelCheckpoint callback.
- You should aim to beat the MLP model performance with fewer parameters!

- Plot the learning curves for loss vs epoch and accuracy vs epoch for both training and validation sets.
- Compute and display the loss and accuracy of the trained model on the test set.

[35]:
```
def get_cnn_model(input_shape,rate):
  model = Sequential([
    Conv2D(16,3, activation='relu',padding='SAME',input_shape=input_shape),
    MaxPool2D(3),
    BatchNormalization(),
    Dropout(rate),
    Conv2D(8,3, activation='relu'),
    MaxPool2D(3),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(10, activation='softmax')
  ])
  model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
↪metrics=['accuracy'])
  return model
```

[36]:
```
model = get_cnn_model((32,32,1), 0.3)
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 16)        160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 10, 10, 16)        0
_________________________________________________________________
batch_normalization (BatchNo (None, 10, 10, 16)        64
_________________________________________________________________
dropout (Dropout)            (None, 10, 10, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 8, 8, 8)           1160
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 2, 2, 8)           0
_________________________________________________________________
flatten_3 (Flatten)          (None, 32)                0
_________________________________________________________________
dense_15 (Dense)             (None, 32)                1056
_________________________________________________________________
dense_16 (Dense)             (None, 10)                330
=================================================================
Total params: 2,770
Trainable params: 2,738
```

```
Non-trainable params: 32
_________________________________________________________________
```

[37]:
```
loss, accuracy = model.evaluate(X_test_grey, y_test)
print(f"Val/Test Loss: {loss}")
print(f'Val/Test Accuracy {accuracy}')
```

```
814/814 [==============================] - 2s 3ms/step - loss: 2.2939 -
accuracy: 0.1766
Val/Test Loss: 2.2938807010650635
Val/Test Accuracy 0.1765519380569458
```

[38]:
```
checkpoint = ModelCheckpoint(
    filepath='model_checkpoints/cnn_best',
    save_weights_only=True,
    save_best_only=True,
    monitor='val_loss',
    verbose=1
)

early_stop = EarlyStopping(
    monitor='val_loss',
    patience=5
)
```

[39]:
```
history = model.fit(X_train_grey, y_train,
                    epochs=30,
                    verbose=2,
                    callbacks=[checkpoint, early_stop],
                    batch_size=256,
                    validation_data=(X_test_grey, y_test)
                    )
```

```
Epoch 1/30

Epoch 00001: val_loss improved from inf to 2.00878, saving model to
model_checkpoints/cnn_best
287/287 - 2s - loss: 1.9364 - accuracy: 0.3239 - val_loss: 2.0088 -
val_accuracy: 0.4173
Epoch 2/30

Epoch 00002: val_loss improved from 2.00878 to 1.22361, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 1.2656 - accuracy: 0.5821 - val_loss: 1.2236 -
val_accuracy: 0.6547
Epoch 3/30
```

```
Epoch 00003: val_loss improved from 1.22361 to 0.98619, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 1.0749 - accuracy: 0.6564 - val_loss: 0.9862 -
val_accuracy: 0.6893
Epoch 4/30

Epoch 00004: val_loss improved from 0.98619 to 0.92399, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 1.0012 - accuracy: 0.6854 - val_loss: 0.9240 -
val_accuracy: 0.7241
Epoch 5/30

Epoch 00005: val_loss improved from 0.92399 to 0.89057, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 0.9535 - accuracy: 0.7019 - val_loss: 0.8906 -
val_accuracy: 0.7321
Epoch 6/30

Epoch 00006: val_loss improved from 0.89057 to 0.87648, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 0.9187 - accuracy: 0.7149 - val_loss: 0.8765 -
val_accuracy: 0.7394
Epoch 7/30

Epoch 00007: val_loss improved from 0.87648 to 0.83100, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 0.8929 - accuracy: 0.7225 - val_loss: 0.8310 -
val_accuracy: 0.7563
Epoch 8/30

Epoch 00008: val_loss improved from 0.83100 to 0.82940, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 0.8748 - accuracy: 0.7287 - val_loss: 0.8294 -
val_accuracy: 0.7500
Epoch 9/30

Epoch 00009: val_loss did not improve from 0.82940
287/287 - 1s - loss: 0.8572 - accuracy: 0.7350 - val_loss: 0.9219 -
val_accuracy: 0.7178
Epoch 10/30

Epoch 00010: val_loss improved from 0.82940 to 0.82193, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 0.8432 - accuracy: 0.7381 - val_loss: 0.8219 -
val_accuracy: 0.7561
Epoch 11/30

Epoch 00011: val_loss improved from 0.82193 to 0.81182, saving model to
```

```
model_checkpoints/cnn_best
287/287 - 1s - loss: 0.8344 - accuracy: 0.7413 - val_loss: 0.8118 -
val_accuracy: 0.7565
Epoch 12/30

Epoch 00012: val_loss improved from 0.81182 to 0.78355, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 0.8279 - accuracy: 0.7437 - val_loss: 0.7836 -
val_accuracy: 0.7670
Epoch 13/30

Epoch 00013: val_loss improved from 0.78355 to 0.77730, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 0.8171 - accuracy: 0.7467 - val_loss: 0.7773 -
val_accuracy: 0.7734
Epoch 14/30

Epoch 00014: val_loss improved from 0.77730 to 0.76453, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 0.8093 - accuracy: 0.7480 - val_loss: 0.7645 -
val_accuracy: 0.7719
Epoch 15/30

Epoch 00015: val_loss improved from 0.76453 to 0.75593, saving model to
model_checkpoints/cnn_best
287/287 - 1s - loss: 0.8021 - accuracy: 0.7530 - val_loss: 0.7559 -
val_accuracy: 0.7778
Epoch 16/30

Epoch 00016: val_loss did not improve from 0.75593
287/287 - 1s - loss: 0.8020 - accuracy: 0.7512 - val_loss: 0.8223 -
val_accuracy: 0.7516
Epoch 17/30

Epoch 00017: val_loss did not improve from 0.75593
287/287 - 1s - loss: 0.7940 - accuracy: 0.7527 - val_loss: 0.7622 -
val_accuracy: 0.7720
Epoch 18/30

Epoch 00018: val_loss did not improve from 0.75593
287/287 - 1s - loss: 0.7897 - accuracy: 0.7546 - val_loss: 0.7994 -
val_accuracy: 0.7541
Epoch 19/30

Epoch 00019: val_loss did not improve from 0.75593
287/287 - 1s - loss: 0.7804 - accuracy: 0.7576 - val_loss: 0.7851 -
val_accuracy: 0.7600
Epoch 20/30
```

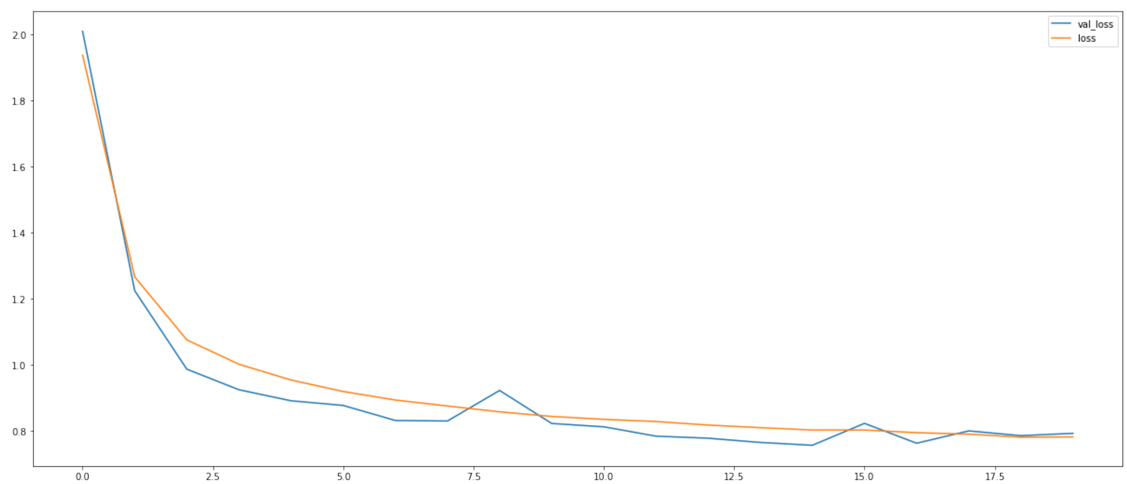
```
Epoch 00020: val_loss did not improve from 0.75593
287/287 - 1s - loss: 0.7813 - accuracy: 0.7569 - val_loss: 0.7921 -
val_accuracy: 0.7597
```

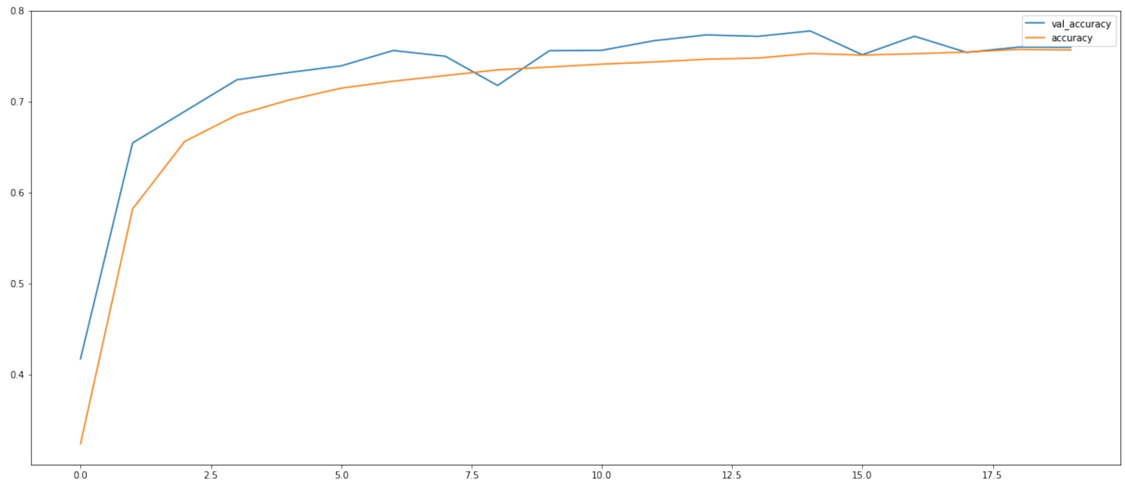
[40]:
```
loss, accuracy = model.evaluate(X_test_grey, y_test)
print(f"Val/Test Loss: {loss}")
print(f'Val/Test Accuracy {accuracy}')
```

```
814/814 [==============================] - 2s 3ms/step - loss: 0.7921 -
accuracy: 0.7597
Val/Test Loss: 0.792052686214447
Val/Test Accuracy 0.7597188353538513
```

[41]:
```
df = pd.DataFrame(history.history)
df[['val_loss','loss']].plot(figsize=(21,9))
df[['val_accuracy','accuracy']].plot(figsize=(21,9))
```

[41]: `<matplotlib.axes._subplots.AxesSubplot at 0x7f824bd43cc0>`





1.5 4. Get model predictions

- Load the best weights for the MLP and CNN models that you saved during the training run.
- Randomly select 5 images and corresponding labels from the test set and display the images with their labels.
- Alongside the image and label, show each model's predictive distribution as a bar chart, and the final model prediction given by the label with maximum probability.

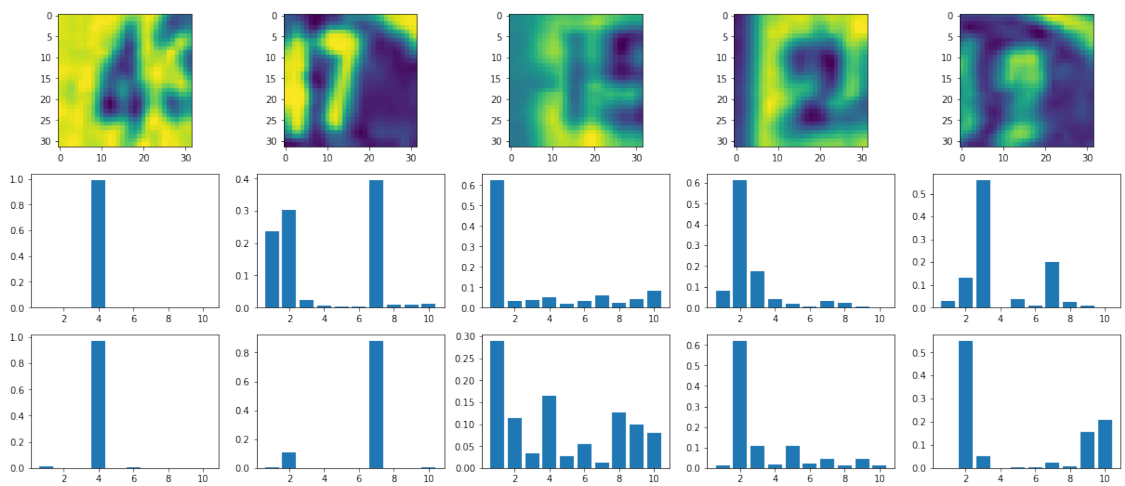
[42]:
```
input_shape = (32,32,1)
mlp = get_mlp_model(input_shape)
mlp.load_weights('model_checkpoints/mlp_best')

cnn = get_cnn_model(input_shape,0.3)
cnn.load_weights('model_checkpoints/cnn_best')
```

[42]:
```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at
0x7f82c2341e10>
```

[72]:
```
indexes = np.random.random_integers(X_test_grey.shape[0],size=5)
data = X_test_grey[indexes]
num = [i for i in range(1,11)]

labels = np.array([y_test[i][0] for i in indexes])
mlp_pred = mlp.predict(data)
cnn_pred = cnn.predict(data)

fig = plt.figure(figsize=(21,9))
for i,index in enumerate(indexes):
  plt.subplot(3,5,i+1)
  plt.imshow(data[i][:,:,0])

print(f"True labels: {labels+1}")
```

```python
print(f"MLP predictions: {np.argmax(mlp_pred,axis=1)+1}")
print(f"CNN predictions: {np.argmax(cnn_pred,axis=1)+1}")

for i,index in enumerate(indexes):
  plt.subplot(3,5,i+6)
  plt.bar(x=num, height=mlp_pred[i])

for i,index in enumerate(indexes):
  plt.subplot(3,5,i+11)
  plt.bar(x=num, height=cnn_pred[i])
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1:
DeprecationWarning: This function is deprecated. Please call randint(1, 26032 +
1) instead
  """Entry point for launching an IPython kernel.

True labels: [4 7 1 2 2]
MLP predictions: [4 7 1 2 3]
CNN predictions: [4 7 1 2 2]
```



[ ]:

[ ]: