Capstone_Project

October 20, 2022

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')
```

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

[3]:
```
! ls
```

```
gdrive  sample_data
```

[4]:
```
%cd gdrive/MyDrive/
```

```
/content/gdrive/MyDrive
```

[5]:
```
! ls
```

```
'Colab Notebooks'   'Week 2 Programming Assignment.ipynb'
 test_32x32.mat     'Week 3 Programming Assignment.ipynb'
 train_32x32.mat    'Week 4 Programming Assignment.ipynb'
```

[6]:
```
%cd ..
```

```
/content/gdrive
```

[7]:
```
%cd ..
```

```
/content
```

[8]:
```
# Load the dataset from your Drive folder

train = loadmat('gdrive/MyDrive/train_32x32.mat')
test = loadmat('gdrive/MyDrive/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.

[9]:
```
train_x = train['X']
train_y = train['y']

test_x = test['X']
test_y = test['y']

train_x.shape
```

[9]:
```
(32, 32, 3, 73257)
```

[10]:
```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 10, figsize=(10, 1))
indices = np.array(np.random.rand(10)*train_x.shape[-1], dtype=int)
for i in range(10):
    ax[i].set_axis_off()
    ax[i].imshow(train_x[:,:,:,indices[i]])
    ax[i].set_title(train_y[indices[i]][0] if train_y[indices[i]][0] != 10 else␣
↪0)
```

[11]:
```
train_x = np.mean(train['X'], axis=2, keepdims=True)
train_x /= np.max(train_x)
test_x = np.mean(test['X'], axis=2, keepdims=True)
test_x /= np.max(test_x)

train_y = np.array(np.mean(train['y'], axis=1), dtype=int)
test_y = np.array(np.mean(test['y'], axis=1), dtype=int)
```

```
train_x.shape
```

[11]: (32, 32, 1, 73257)

[12]:
```
fig, ax = plt.subplots(1, 10, figsize=(10, 1))
indices = np.array(np.random.rand(10)*train_x.shape[-1], dtype=int)
for i in range(10):
    ax[i].set_axis_off()
    ax[i].imshow(train_x[:,:,0,indices[i]], cmap='gray')
    ax[i].set_title(train_y[indices[i]] if train_y[indices[i]] != 10 else 0)
```

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.

[13]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
```

[14]:
```
model = Sequential([
    Flatten(input_shape=train_x.shape[:-1], name='input'),
    Dense(100, activation='relu', name='layer1'),
    Dense(50, activation='relu', name='layer2'),
    Dense(20, activation='relu', name='layer3'),
    Dense(10, activation='softmax', name='output')
])
```

[15]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (Flatten)             (None, 1024)              0

 layer1 (Dense)              (None, 100)               102500

 layer2 (Dense)              (None, 50)                5050

 layer3 (Dense)              (None, 20)                1020

 output (Dense)              (None, 10)                210

=================================================================
Total params: 108,780
Trainable params: 108,780
Non-trainable params: 0
_________________________________________________________________
```

[16]:
```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
↪metrics=['accuracy'])
```

[17]:
```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

def get_checkpoint_best_only(path):
    checkpoint_best_path = path+'/checkpoint'
    checkpoint_best = ModelCheckpoint(filepath=checkpoint_best_path,
                                      save_weights_only=True,
                                      save_freq='epoch',
                                      monitor='val_accuracy',
                                      save_best_only=True,
                                      verbose=1)
    return checkpoint_best

def get_early_stopping():
    return tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=5)
```

[18]:
```
## Comment: In order to use the "sparse_categorical_crossentropy" loss␣
↪function, the label for the
##          target '0' has to be swapped from '10' to '0'.

train_y[train_y == 10] = 0
test_y[test_y == 10] = 0
```

```
print(np.max(train_y))
print(np.max(test_y))
```

```
9
9
```

[19]:
```
## Comment: The data shape has to meet the keras input format convention.␣
 ↪Hence, the input tensor is
##          reshaped accordingly.

tf.transpose(train_x, [3, 0, 1, 2]).shape
```

[19]:
```
TensorShape([73257, 32, 32, 1])
```

[20]:
```
history = model.fit(tf.transpose(train_x, [3, 0, 1, 2]), train_y, epochs=30,␣
 ↪validation_split=0.2, batch_size=200,␣
 ↪callbacks=[get_checkpoint_best_only('MLP'), get_early_stopping()], verbose=0)
```

```
Epoch 1: val_accuracy improved from -inf to 0.18953, saving model to
MLP/checkpoint

Epoch 2: val_accuracy improved from 0.18953 to 0.38111, saving model to
MLP/checkpoint

Epoch 3: val_accuracy improved from 0.38111 to 0.44260, saving model to
MLP/checkpoint

Epoch 4: val_accuracy improved from 0.44260 to 0.50669, saving model to
MLP/checkpoint

Epoch 5: val_accuracy improved from 0.50669 to 0.57166, saving model to
MLP/checkpoint

Epoch 6: val_accuracy did not improve from 0.57166

Epoch 7: val_accuracy improved from 0.57166 to 0.59855, saving model to
MLP/checkpoint

Epoch 8: val_accuracy did not improve from 0.59855

Epoch 9: val_accuracy improved from 0.59855 to 0.64087, saving model to
MLP/checkpoint

Epoch 10: val_accuracy did not improve from 0.64087

Epoch 11: val_accuracy improved from 0.64087 to 0.65854, saving model to
```

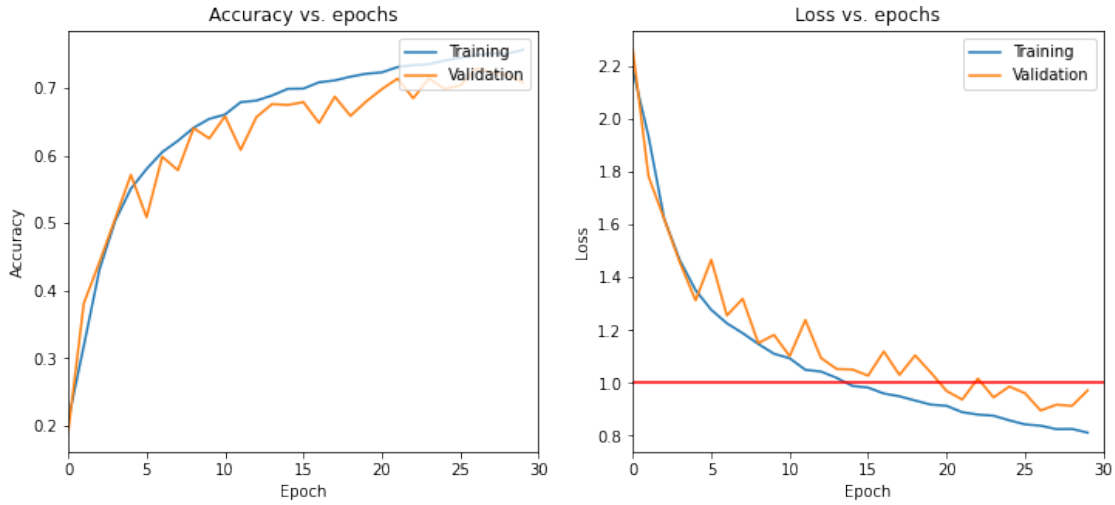
```
MLP/checkpoint

Epoch 12: val_accuracy did not improve from 0.65854

Epoch 13: val_accuracy did not improve from 0.65854

Epoch 14: val_accuracy improved from 0.65854 to 0.67663, saving model to
MLP/checkpoint

Epoch 15: val_accuracy did not improve from 0.67663

Epoch 16: val_accuracy improved from 0.67663 to 0.67950, saving model to
MLP/checkpoint

Epoch 17: val_accuracy did not improve from 0.67950

Epoch 18: val_accuracy improved from 0.67950 to 0.68762, saving model to
MLP/checkpoint

Epoch 19: val_accuracy did not improve from 0.68762

Epoch 20: val_accuracy did not improve from 0.68762

Epoch 21: val_accuracy improved from 0.68762 to 0.69874, saving model to
MLP/checkpoint

Epoch 22: val_accuracy improved from 0.69874 to 0.71431, saving model to
MLP/checkpoint

Epoch 23: val_accuracy did not improve from 0.71431

Epoch 24: val_accuracy improved from 0.71431 to 0.71451, saving model to
MLP/checkpoint

Epoch 25: val_accuracy did not improve from 0.71451

Epoch 26: val_accuracy did not improve from 0.71451

Epoch 27: val_accuracy improved from 0.71451 to 0.72918, saving model to
MLP/checkpoint

Epoch 28: val_accuracy did not improve from 0.72918

Epoch 29: val_accuracy did not improve from 0.72918

Epoch 30: val_accuracy did not improve from 0.72918
```

[21]:
```
import pandas as pd

df = pd.DataFrame(history.history)
df.head()
```

[21]:

	loss	accuracy	val_loss	val_accuracy
0	2.186202	0.207013	2.262533	0.189530
1	1.934450	0.318198	1.782617	0.381108
2	1.622151	0.431618	1.621413	0.442602
3	1.465087	0.503387	1.456197	0.506689
4	1.351783	0.551079	1.313012	0.571663

[22]:
```
fig = plt.figure(figsize=(12, 5))

fig.add_subplot(121)

plt.xlim([0,30])
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Accuracy vs. epochs')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')

fig.add_subplot(122)

plt.xlim([0,30])
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.plot([0,50], [1,1], 'r-')

plt.show()
```



[23]:
```
test_loss, test_acc = model.evaluate(tf.transpose(test_x, [3, 0, 1, 2]),␣
↪test_y, verbose=2)
```

```
814/814 - 1s - loss: 1.0464 - accuracy: 0.6974 - 1s/epoch - 2ms/step
```

[24]:
```
print('Test loss = '+str(np.round(test_loss, 3)))
print('Test accuracy = '+str(np.round(test_acc*100, 1))+'%')
```

```
Test loss = 1.046
Test accuracy = 69.7%
```

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.

[25]:
```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, BatchNormalization,␣
↪Dropout
from tensorflow.keras import regularizers

CNN_model = Sequential([
        Conv2D(filters=20, kernel_regularizer=regularizers.l2(1E-5),␣
↪input_shape=train_x.shape[:-1],
               kernel_size=(3, 3), activation='relu', padding='same',␣
↪name='conv_1'),
        MaxPooling2D(pool_size=(3, 3), name='pool_1'),
        Conv2D(filters=10, kernel_regularizer=regularizers.l2(1E-5),
               kernel_size=(3, 3), activation='relu', padding='same',␣
↪name='conv_2'),
        MaxPooling2D(pool_size=(3, 3), name='pool_2'),
        Flatten(name='flatten'),
        Dense(units=100, kernel_regularizer=regularizers.l2(1E-5),␣
↪activation='relu', name='dense_1'),
        Dropout(0.25, name='dropout'),
        Dense(50, kernel_regularizer=regularizers.l2(1E-5), activation="relu",␣
↪name='dense_2'),
        BatchNormalization(name='bacth_normalization'),
        Dense(10, kernel_regularizer=regularizers.l2(1E-5),␣
↪activation="softmax", name='dense_3')
])
```

[26]:
```
CNN_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv_1 (Conv2D)              (None, 32, 32, 20)        200

 pool_1 (MaxPooling2D)        (None, 10, 10, 20)        0

 conv_2 (Conv2D)              (None, 10, 10, 10)        1810

 pool_2 (MaxPooling2D)        (None, 3, 3, 10)          0

 flatten (Flatten)            (None, 90)                0

 dense_1 (Dense)              (None, 100)               9100

 dropout (Dropout)            (None, 100)               0

 dense_2 (Dense)              (None, 50)                5050
```

```
 bacth_normalization (BatchN  (None, 50)               200
 ormalization)

 dense_3 (Dense)             (None, 10)                510

=================================================================
Total params: 16,870
Trainable params: 16,770
Non-trainable params: 100
_________________________________________________________________
```

[27]:
```
CNN_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
↪metrics=['accuracy'])
```

[28]:
```
CNN_history = CNN_model.fit(tf.transpose(train_x, [3, 0, 1, 2]), train_y,␣
↪epochs=30, validation_split=0.2, batch_size=200,␣
↪callbacks=[get_checkpoint_best_only('CNN'), get_early_stopping()], verbose=0)
```

```
Epoch 1: val_accuracy improved from -inf to 0.50171, saving model to
CNN/checkpoint

Epoch 2: val_accuracy improved from 0.50171 to 0.71963, saving model to
CNN/checkpoint

Epoch 3: val_accuracy improved from 0.71963 to 0.77314, saving model to
CNN/checkpoint

Epoch 4: val_accuracy improved from 0.77314 to 0.77498, saving model to
CNN/checkpoint

Epoch 5: val_accuracy did not improve from 0.77498

Epoch 6: val_accuracy improved from 0.77498 to 0.80856, saving model to
CNN/checkpoint

Epoch 7: val_accuracy did not improve from 0.80856

Epoch 8: val_accuracy did not improve from 0.80856

Epoch 9: val_accuracy improved from 0.80856 to 0.81456, saving model to
CNN/checkpoint

Epoch 10: val_accuracy improved from 0.81456 to 0.82016, saving model to
CNN/checkpoint

Epoch 11: val_accuracy did not improve from 0.82016
```

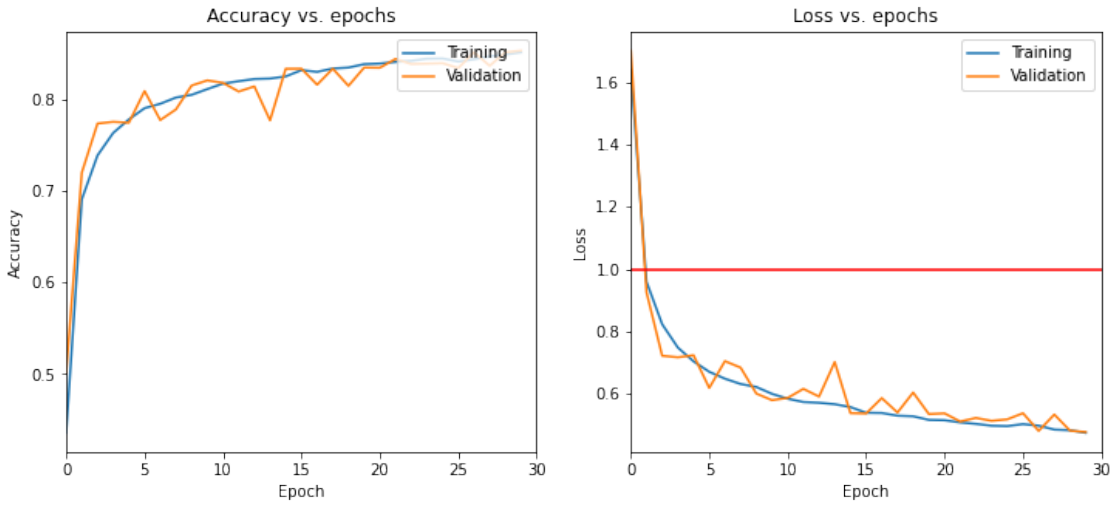
```
Epoch 12: val_accuracy did not improve from 0.82016

Epoch 13: val_accuracy did not improve from 0.82016

Epoch 14: val_accuracy did not improve from 0.82016

Epoch 15: val_accuracy improved from 0.82016 to 0.83286, saving model to
CNN/checkpoint

Epoch 16: val_accuracy improved from 0.83286 to 0.83292, saving model to
CNN/checkpoint

Epoch 17: val_accuracy did not improve from 0.83292

Epoch 18: val_accuracy did not improve from 0.83292

Epoch 19: val_accuracy did not improve from 0.83292

Epoch 20: val_accuracy improved from 0.83292 to 0.83415, saving model to
CNN/checkpoint

Epoch 21: val_accuracy did not improve from 0.83415

Epoch 22: val_accuracy improved from 0.83415 to 0.84357, saving model to
CNN/checkpoint

Epoch 23: val_accuracy did not improve from 0.84357

Epoch 24: val_accuracy did not improve from 0.84357

Epoch 25: val_accuracy did not improve from 0.84357

Epoch 26: val_accuracy did not improve from 0.84357

Epoch 27: val_accuracy improved from 0.84357 to 0.85081, saving model to
CNN/checkpoint

Epoch 28: val_accuracy did not improve from 0.85081

Epoch 29: val_accuracy improved from 0.85081 to 0.85087, saving model to
CNN/checkpoint

Epoch 30: val_accuracy improved from 0.85087 to 0.85258, saving model to
CNN/checkpoint
```

[29]:
```
df2 = pd.DataFrame(CNN_history.history)
df2.head()
```

[29]:

	loss	accuracy	val_loss	val_accuracy
0	1.639930	0.435850	1.700411	0.501706
1	0.959395	0.690521	0.925184	0.719629
2	0.822808	0.738350	0.721636	0.773137
3	0.747927	0.762887	0.716807	0.774980
4	0.703723	0.777630	0.723343	0.774024

[30]:
```
fig = plt.figure(figsize=(12, 5))

fig.add_subplot(121)

plt.xlim([0,30])
plt.plot(CNN_history.history['accuracy'])
plt.plot(CNN_history.history['val_accuracy'])
plt.title('Accuracy vs. epochs')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')

fig.add_subplot(122)

plt.xlim([0,30])
plt.plot(CNN_history.history['loss'])
plt.plot(CNN_history.history['val_loss'])
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.plot([0,50], [1,1], 'r-')

plt.show()
```



[31]:
```
CNN_test_loss, CNN_test_acc = CNN_model.evaluate(tf.transpose(test_x, [3, 0, 1,␣
↪2]), test_y, verbose=2)
```

```
814/814 - 7s - loss: 0.4942 - accuracy: 0.8506 - 7s/epoch - 8ms/step
```

[32]:
```
print('CNN test loss = '+str(np.round(CNN_test_loss, 3))+' < '+str(np.
↪round(test_loss, 3))+' = MLP test loss')
print('CNN test accuracy = '+str(np.round(CNN_test_acc*100, 1))+'%'+' >␣
↪'+str(np.round(test_acc*100, 1))+'% = MLP test accuracy')
```

```
CNN test loss = 0.494 < 1.046 = MLP test loss
CNN test accuracy = 85.1% > 69.7% = MLP test accuracy
```

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.

[35]:
```
model.load_weights('MLP/checkpoint')
CNN_model.load_weights('CNN/checkpoint')
```

[35]:
```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at
0x7f6d02ad2bd0>
```

[111]:
```
fig, ax = plt.subplots(1, 5, figsize=(5, 1))
indices = np.array(np.random.rand(5)*test_x.shape[-1], dtype=int)
for i in range(5):
    ax[i].set_axis_off()
    ax[i].imshow(test_x[:,:,0,indices[i]], cmap='gray')
    ax[i].set_title(test_y[indices[i]])
```

[112]:
```
predictions = np.zeros((len(indices), 10))
for j in range(5):
```

```
    predictions[j,:] = model.predict(tf.transpose(test_x, [3, 0, 1, 2])[np.
↪newaxis, indices[j], ...])
```

```
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 18ms/step
```

[113]:
```
CNN_predictions = np.zeros((len(indices), 10))
for j in range(5):
    CNN_predictions[j,:] = CNN_model.predict(tf.transpose(test_x, [3, 0, 1,␣
↪2])[np.newaxis, indices[j], ...])
```

```
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 18ms/step
```

[120]:
```
bar_width = 0.3
position_MLP = np.arange(10)-bar_width/2
position_CNN = np.arange(10)+bar_width/2

fig, ax = plt.subplots(2, 5, figsize=(15, 4))
for i in range(5):
    ax[0,i].set_axis_off()
    ax[0,i].imshow(test_x[:,:,0,indices[i]], cmap='gray')
    ax[0,i].set_title('True label: '+str(test_y[indices[i]]))

    if (i != 0):
        ax[1,i].set_yticks([])
        ax[1,i].set_yticklabels([])
    else:
        ax[1,i].set_ylabel('probability [%]')
    ax[1,i].set_xlabel('classes')
    ax[1,i].set_xlim([-bar_width, 9+bar_width])
    ax[1,i].set_xticks(np.arange(0, 10))
    ax[1,i].set_ylim([0, 100])
    ax[1,i].set_title('MLP pred.: '+str(np.argmax(predictions[i,:]))+', CNN␣
↪pred.: '+str(np.argmax(CNN_predictions[i,:])))
    ax[1,i].bar(position_MLP, predictions[i,:]*100, bar_width, color='blue',␣
↪label='MLP')
    ax[1,i].bar(position_CNN, CNN_predictions[i,:]*100, bar_width, color='red',␣
↪label='CNN')
    ax[1,i].legend()
```

```
fig.tight_layout()
plt.show()
```