In [ ]:
```python
# set the seed to get reproducible results
from black import out
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from tensorflow import keras
from tensorflow.keras.optimizers import RMSprop
from keras_tuner import Objective
from keras.layers import Flatten, Dense, Dropout
from keras import Sequential
from keras.initializers import RandomNormal, HeNormal
from keras.datasets import mnist
from keras.regularizers import l2, l1
from keras.optimizers import SGD
import keras_tuner as kt
import matplotlib.pyplot as plt
import random as python_random
import tensorflow as tf
import numpy as np
import os
from pytictoc import TicToc  # time difference

# visualization
import matplotlib.pyplot as plt
import seaborn as sns


def set_seed(seed):
    os.environ["PYTHONHASHSEED"] = str(seed)
    os.environ["TF_CUDNN_DETERMINISTIC"] = str(seed)

    # source: https://keras.io/getting_started/faq/#how-can-i-obtain-reproducible-res
    # source: https://github.com/keras-team/keras/issues/2743
    np.random.seed(seed)
    python_random.seed(seed)
    tf.random.set_seed(seed)
```

In [ ]:
```python
# Data preparation

#  load and normalize dataset
(train_x, train_y), (test_x, test_y) = mnist.load_data()

# vectorize image row-wise
# shape = (num_samples, num_features)
train_x = train_x.reshape(train_x.shape[0], -1)
test_x = test_x.reshape(test_x.shape[0], -1)

num_samples_training = train_x.shape[0]
num_features = train_x.shape[1]
num_classes = 10

# memory efficient
train_x = train_x.astype("float32")
test_x = test_x.astype("float32")

# min-max normalization
train_x = train_x / 255
test_x = test_x / 255

# manual one-hot encoding instead of using 'sparse_categorical_entropy' (now 'categor
# reason: https://stackoverflow.com/questions/49019383/keras-precision-and-recall-is-
train_y_1hot = keras.utils.to_categorical(train_y, num_classes)
test_y_1hot = keras.utils.to_categorical(test_y, num_classes)
```

In [ ]:
```python
# Helper functions
```

```
# build the MLP
# Notes:
# regularization can be used in the output layer too, although in most examples they
# dropout should not be used for input and output layers
def create_model(
    hidden_layer_nodes1=128,
    hidden_layer_nodes2=256,
    kernel_initializer=None,
    kernel_regularizer=None,
    is_dropout=False,
    optimizer="adam",
    metrics=["accuracy"],
):
    hidden_layer_options = {}
    output_layer_options = {}
    if kernel_initializer:
        hidden_layer_options["kernel_initializer"] = kernel_initializer
        output_layer_options["kernel_initializer"] = kernel_initializer
    if kernel_regularizer:
        hidden_layer_options["kernel_regularizer"] = kernel_regularizer

    model = Sequential()
    # 1st hidden layer
    model.add(Dense(hidden_layer_nodes1, input_shape=(num_features,), activation="rel

    if is_dropout:
        # source: https://machinelearningmastery.com/how-to-reduce-overfitting-with-d
        model.add(Dropout(0.3))

    # 2nd hidden layer
    model.add(Dense(hidden_layer_nodes2, activation="relu", **hidden_layer_options))

    if is_dropout:
        model.add(Dropout(0.3))

    # output
    model.add(Dense(num_classes, activation="softmax", **output_layer_options))
    # model.add(Dense(num_classes, activation="softmax", **hidden_layer_options))

    model.compile(optimizer=optimizer, loss="categorical_crossentropy", metrics=metri
    return model


def plot_weights(weight):
    plt.figure(constrained_layout=True)
    plt.subplot(131)
    sns.violinplot(y=weight[0], color="b")
    plt.xlabel("Hidden Layer 1")
    plt.subplot(132)
    sns.violinplot(y=weight[1], color="y")
    plt.xlabel("Hidden Layer 2")
    plt.subplot(133)
    sns.violinplot(y=weight[2], color="r")
    plt.xlabel("Output")


def filter_weights(model):
    weights = [
        model.get_weights()[0].flatten().reshape(-1, 1),  # hidden layer 1
        model.get_weights()[2].flatten().reshape(-1, 1),  # hidden layer 2
        model.get_weights()[4].flatten().reshape(-1, 1),  # output
    ]
    return weights
```

```python
def fitWrapper(batch_size, epochs):
    history = model.fit(
        train_x,
        train_y_1hot,
        batch_size=batch_size,
        epochs=epochs,
        validation_split=0.2,
        shuffle=False,
    )
    weights = filter_weights(model)
    return history, weights


# plot learning curves
def plot_history(history):
    epochs = len(history.history["accuracy"])
    x = np.arange(1, epochs + 1)
    plt.figure(constrained_layout=True)
    plt.subplot(211)
    plt.plot(x, history.history["accuracy"])
    plt.plot(x, history.history["val_accuracy"], color="green")
    # plt.xlabel("epochs")
    # plt.ylabel("accuracy")
    plt.legend(["train", "validation"], loc="upper left")

    plt.subplot(212)
    plt.plot(x, history.history["loss"])
    plt.plot(x, history.history["val_loss"], color="green")
    plt.xlabel("epochs")
    plt.ylabel("loss")
    plt.legend(["train", "validation"], loc="upper right")
```

In [ ]:

```python
# default network for different batch sizes
batches = [1, 256,num_samples_training]
for batch in batches:
    set_seed(1)  # get reproducible results
    print("\n\nBatch size: " + str(batch))
    # default optimizer adam
    model = create_model()
    weights = filter_weights(model)
    plot_weights(weights)

    t = TicToc()
    t.tic()
    history, weight = fitWrapper(batch_size=batch, epochs=100)
    t.toc()

    result = model.evaluate(test_x, test_y_1hot)
    print("Accuracy: " + str(result))
    plot_weights(weight)
    plot_history(history)
```

In [ ]:

```python
# rmsprop

rhos = [0.01, 0.99]
for rho in rhos:
    set_seed(1)

    model = create_model(optimizer=RMSprop(learning_rate=0.001, rho=rho))
    weights = filter_weights(model)
    plot_weights(weights)

    history, weights = fitWrapper(batch_size=256, epochs=100)

    weights = filter_weights(model)
```

```
    plot_weights(weights)
    plot_history(history)
    print("Evalute", model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
# sgd + weight initialization
set_seed(1)

model = create_model(
    optimizer=SGD(lr=0.01),
    kernel_initializer=RandomNormal(mean=10))
weights = filter_weights(model)
plot_weights(weights)

history, weights = fitWrapper(batch_size=256, epochs=100)

weights = filter_weights(model)
plot_weights(weights)
plot_history(history)
print("Evalute", model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
# l2 regularization model + rmsprop
alphas = [0.1, 0.01, 0.001]
for alpha in alphas:
    set_seed(1)

    model = create_model(
        optimizer=RMSprop(learning_rate=0.001, rho=0.99),
        kernel_regularizer=l2(alpha)
    )
    weights = filter_weights(model)
    plot_weights(weights)

    history, weights = fitWrapper(batch_size=256, epochs=100)

    weights = filter_weights(model)
    plot_weights(weights)
    plot_history(history)
    print("Evaluate", alpha, model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
# l2 regularization model + sgd
alphas = [0.1, 0.01, 0.001]
for alpha in alphas:
    set_seed(1)

    model = create_model(
        optimizer=SGD(lr=0.01),
        kernel_initializer=RandomNormal(10),
        kernel_regularizer=l2(alpha))
    weights = filter_weights(model)
    plot_weights(weights)

    history, weights = fitWrapper(batch_size=256, epochs=100)

    weights = filter_weights(model)
    plot_weights(weights)
    plot_history(history)
    print("Evaluate", alpha, model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
# l1-dropout regularization rmsprop
set_seed(1)

model = create_model(
    optimizer=RMSprop(learning_rate=0.001, rho=0.99),
    kernel_regularizer=l1(0.01),
```

```
    is_dropout=True,
)
weights = filter_weights(model)
plot_weights(weights)

history, weights = fitWrapper(batch_size=256, epochs=100)

weights = filter_weights(model)
plot_weights(weights)
plot_history(history)
print("Evalute", model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
# l1-dropout regularization sgd + initialization
set_seed(1)

model = create_model(
    optimizer=SGD(lr=0.01),
    kernel_initializer=RandomNormal(10),
    kernel_regularizer=l1(0.01),
    is_dropout=True,
)
weights = filter_weights(model)
plot_weights(weights)

history, weights = fitWrapper(batch_size=256, epochs=100)

weights = filter_weights(model)
plot_weights(weights)
plot_history(history)
print("Evalute", model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
#  Fine-tuning
from keras.callbacks import EarlyStopping

set_seed(1)

# custom metric functions
# source: https://github.com/keras-team/autokeras/issues/867#issuecomment-664794336
from keras import backend as K


def recall_m(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
    recall = true_positives / (possible_positives + K.epsilon())
    return recall


def precision_m(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
    precision = true_positives / (predicted_positives + K.epsilon())
    return precision


def f1_score(y_true, y_pred):
    precision = precision_m(y_true, y_pred)
    recall = recall_m(y_true, y_pred)
    return 2 * ((precision * recall) / (precision + recall + K.epsilon()))


def build_model(hp):
    hidden_layer_nodes1 = hp.Choice("hidden_layer_nodes1", values=[64, 128])
    hidden_layer_nodes2 = hp.Choice("hidden_layer_nodes2", values=[256, 512])
    learning_rate = hp.Choice("learning_rate", values=[0.1, 0.01, 0.001])
```

```
    l2_alpha = hp.Choice("l2_alpha", values=[0.1, 0.001, 0.000001])
    return create_model(
        hidden_layer_nodes1=hidden_layer_nodes1,
        hidden_layer_nodes2=hidden_layer_nodes2,
        kernel_regularizer=l2(l2_alpha),
        kernel_initializer=HeNormal(),
        optimizer=RMSprop(learning_rate=learning_rate),
        metrics=["accuracy", f1_score, recall_m, precision_m],
    )


build_model(kt.HyperParameters())
```

Choose your HyperBand

Objective can be:

- `val_f1_score`
- `val_accuracy`

In [ ]:
```
# https://neptune.ai/blog/keras-tuner-tuning-hyperparameters-deep-learning-model
# we could use `val_f1`` ? But we will have bad results?
# `val_f1` isn't supported but you can define cusotm f1 function
# https://github.com/keras-team/autokeras/issues/867


tuner = kt.Hyperband(hypermodel=build_model, objective=Objective("val_f1_score", dire
tuner.search(
    train_x,
    train_y_1hot,
    validation_split=0.2,
    epochs=1000,
    callbacks=[EarlyStopping(patience=200, monitor="val_loss")],
)
tuner.results_summary()
```

In [ ]:
```
tuner = kt.Hyperband(hypermodel=build_model, objective="val_accuracy")
tuner.search(
    train_x,
    train_y_1hot,
    validation_split=0.2,
    epochs=1000,
    callbacks=[EarlyStopping(patience=200, monitor="val_loss")],
)
tuner.results_summary()
```

In [ ]:
```
# train with the best hyperparameters

best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
best_model = tuner.hypermodel.build(best_hps)

history = best_model.fit(train_x, train_y_1hot, epochs=50, validation_split=0.2)
loss_val, accuracy_val, f1_score_val, recall_val, precision_val = best_model.evaluate
    test_x, test_y_1hot
)

print("loss:" + str(loss_val))
print("accuracy:" + str(accuracy_val))
print("f1 score:" + str(f1_score_val))
print("recall:" + str(recall_val))
print("precision:" + str(precision_val))
```

```
# learning curves
plot_history(history)
```

In [ ]:

```
# this one is already trained by getting the best model and we can observe overfittin
# we deduce that by the starting training accuracy of 1.0
# it is better to create the model by the hyperparameters

# train, test the best model
best_model = tuner.get_best_models(num_models=1)[0]
history = best_model.fit(train_x, train_y_1hot, epochs=10, validation_split=0.2)
loss_val, accuracy_val, f1_score_val, recall_val, precision_val = best_model.evaluate
    test_x, test_y_1hot
)

print("loss:" + str(loss_val))
print("accuracy:" + str(accuracy_val))
print("f1 score:" + str(f1_score_val))
print("recall:" + str(recall_val))
print("precision:" + str(precision_val))

# learning curves
plot_history(history)
```

In [ ]:

```
# confusion matrix
from sklearn.metrics import confusion_matrix

y_pred = best_model.predict(test_x)
confusion_mat = confusion_matrix(test_y, y_pred.argmax(axis=1))
normed_conf = (confusion_mat.T / confusion_mat.astype(float).sum(axis=1)).T

fig, ax = plt.subplots(figsize=(10, 10))
sns.heatmap(normed_conf, annot=True, fmt=".2f")
plt.ylabel("Actual")
plt.xlabel("Predicted")
```