Exercise 6.4

MNIST with fully connected networks and grid/random search

```
import numpy as np
import matplotlib.pyplot as plt
import os
import tensorflow as tf
from tabulate import tabulate
from tensorflow.keras import layers
```

```
#For verifying GPU is used when run locally
sess = tf.compat.v1.Session(config=tf.compat.v1.ConfigProto(log_device_placement=True)
```

```
Device mapping:
/job:localhost/replica:0/task:0/device:GPU:0 -> device: 0, name: NVIDIA GeForce GT
```

The MNIST data base of handwritten numbers is directly available through KERAS. The following codeblocks download and preprocess the data.

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

```
x_train = x_train / 255.0
y_train = y_train
x_test = x_test / 255.0
y_test = y_test

x_valid = x_test[8000:]
y_valid = y_test[8000:]
x_test = x_test[:8000]
y_test = y_test[:8000]

# Hint: convert integer RGB values (0-255) to float values (0-1)
```

```
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_valid.shape[0], "validation samples")
print(x_test.shape[0], "test samples")
```

```
x_train shape: (60000, 28, 28)
60000 train samples
2000 validation samples
8000 test samples
```

In this exercise, a fully connected neural network is used to predict the handwritten numbers.

To do this, we reformat the pictures with 28x28 pixels into a vector with a length of 28x28=784.

```
# reshape the image matrices to vectors
x_train = x_train.reshape(-1, 28**2)
x_valid = x_valid.reshape(-1, 28**2)
x_test = x_test.reshape(-1, 28**2)
print("x_train shape:", x_train.shape)
```

```
x_train shape: (60000, 784)
```

We use "onehot" encoding of the classes. This means a "zero" is encoded as `[1,0,0,0,0,0,0,0,0,0]` and a "one" as `[0,1,0,0,0,0,0,0,0,0]` etc. This is done because our network will have ten output nodes with the output node with the largest value being the predicted number.

```
# convert class vectors to binary class matrices (10 numbers/classes)
y_train_onehot = tf.keras.utils.to_categorical(y_train, 10)
y_valid_onehot = tf.keras.utils.to_categorical(y_valid, 10)
y_test_onehot = tf.keras.utils.to_categorical(y_test, 10)
```

```
# define model here
model = tf.keras.models.Sequential([
    layers.Dense(128, input_dim=784, activation="relu"),
    layers.Dropout(0.3),
    layers.Dense(10),
    layers.Activation('softmax')])  # softmax actication to transform output into prob

print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               100480

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

 activation (Activation)     (None, 10)                0

=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
None
```

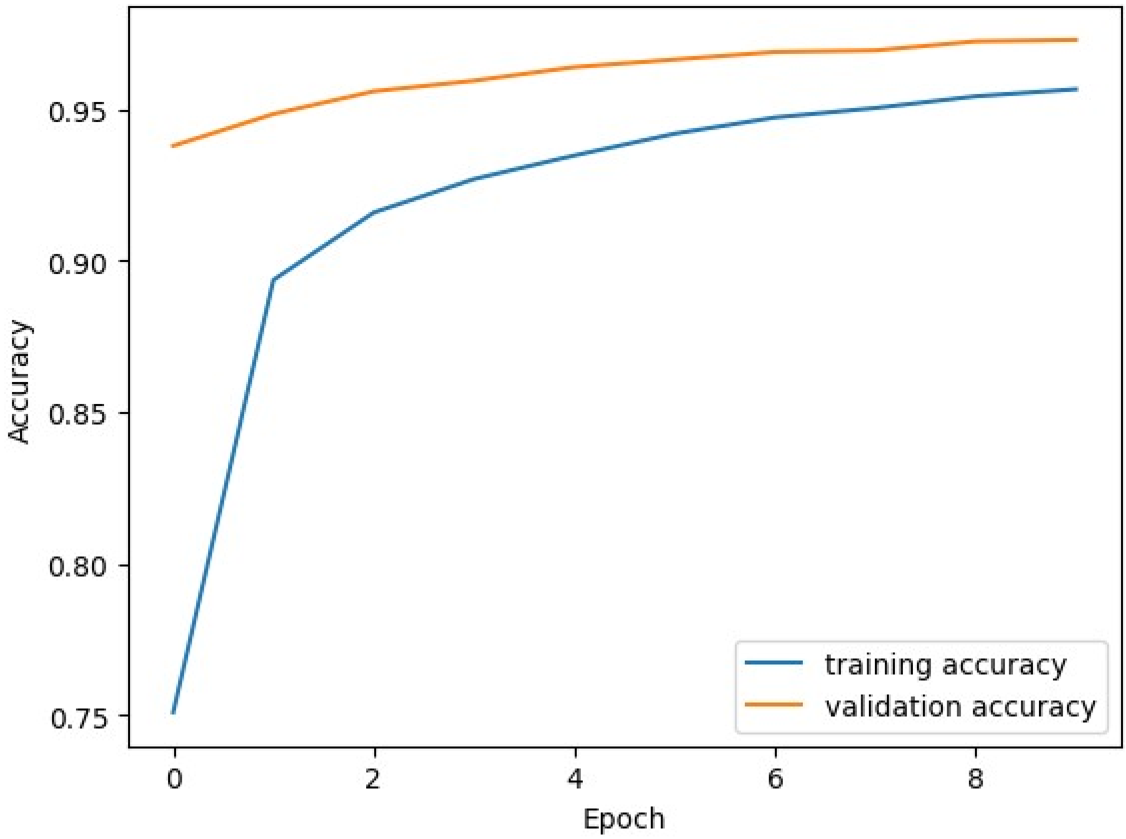
```
model.compile(
```

```
    loss='categorical_crossentropy',  # the recommended loss for a classification task
    optimizer="adam",
    metrics=['accuracy']) # we use accuracy to quanitfy to network performance.

# define callbacks for training
save_best = tf.keras.callbacks.ModelCheckpoint(
    "best_model_{}.weights.h5".format(model.name),
    save_best_only=True,
    monitor="val_accuracy",
    save_weights_only=True,
)

# Keras calculates training accuracy and loss during the training and with regularizat
# while the validation metrics are calculated at the end of each epoch.
# This callback calculates the training metrics the same way as for the validation
class CalculateMetrics(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        train_loss, train_acc = model.evaluate(x_train, y_train_onehot, verbose=0)
        logs["train_loss"] = train_loss
        logs["train_acc"] = train_acc

results = model.fit(
    x_train, y_train_onehot,
    validation_data=(x_valid, y_valid_onehot),

    batch_size=1000,
    epochs=10,
    callbacks=[
        save_best,
        CalculateMetrics(),
        tf.keras.callbacks.CSVLogger("history_{}.csv".format(model.name))
    ]
    )
```

```
Epoch 1/10
60/60 [==============================] - 5s 66ms/step - loss: 0.8654 - accuracy: 0
Epoch 2/10
60/60 [==============================] - 4s 62ms/step - loss: 0.3723 - accuracy: 0
Epoch 3/10
60/60 [==============================] - 4s 62ms/step - loss: 0.2976 - accuracy: 0
Epoch 4/10
60/60 [==============================] - 4s 63ms/step - loss: 0.2541 - accuracy: 0
Epoch 5/10
60/60 [==============================] - 4s 64ms/step - loss: 0.2253 - accuracy: 0
Epoch 6/10
60/60 [==============================] - 4s 64ms/step - loss: 0.2028 - accuracy: 0
Epoch 7/10
60/60 [==============================] - 4s 66ms/step - loss: 0.1855 - accuracy: 0
Epoch 8/10
60/60 [==============================] - 4s 67ms/step - loss: 0.1734 - accuracy: 0
Epoch 9/10
60/60 [==============================] - 4s 67ms/step - loss: 0.1593 - accuracy: 0
Epoch 10/10
60/60 [==============================] - 4s 66ms/step - loss: 0.1496 - accuracy: 0
```

```
# load best model
model.load_weights(f"best_model_{model.name}.weights.h5")

# ------------------------
# ---     Plotting     ---
# ------------------------

# plot training history
history = np.genfromtxt(f"history_{model.name}.csv", delimiter=",", names=True)
# add plots below



plt.figure()
plt.plot(history["epoch"], history["accuracy"], label="training accuracy")
plt.plot(history["epoch"], history["val_accuracy"], label="validation accuracy")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
```

```
Text(0, 0.5, 'Accuracy')
```



Note

Seems like we can train a bit more than 10 epochs for better result.

Validation accuracy is in this case higher, meaning we have an indication for underfitting. Network can be made more complex.

```
# evaluate performance

print("Model performance :")
headers = ["", "Loss", "Accuracy", "Test error rate [%]"]

table = [
    ["Train", *model.evaluate(x_train, y_train_onehot, verbose=0, batch_size=128), (1-
    ["Validation", *model.evaluate(x_valid, y_valid_onehot, verbose=0, batch_size=128)
    ["Test", *model.evaluate(x_test, y_test_onehot, verbose=0, batch_size=128), (1-mod
]

print(tabulate(table, headers=headers, tablefmt="orgtbl"))
```

```
Model performance :
|            |      Loss |   Accuracy |   Test error rate [%] |
|------------+-----------+------------+-----------------------|
| Train      | 0.105706  |   0.970317 |               2.96834 |
| Validation | 0.0822106 |   0.9765   |               2.35    |
| Test       | 0.126023  |   0.9635   |               3.65    |
```

You can compare your own results with a variety of different models: http://yann.lecun.com/exdb/mnist/ and https://en.wikipedia.org/wiki/MNIST_database

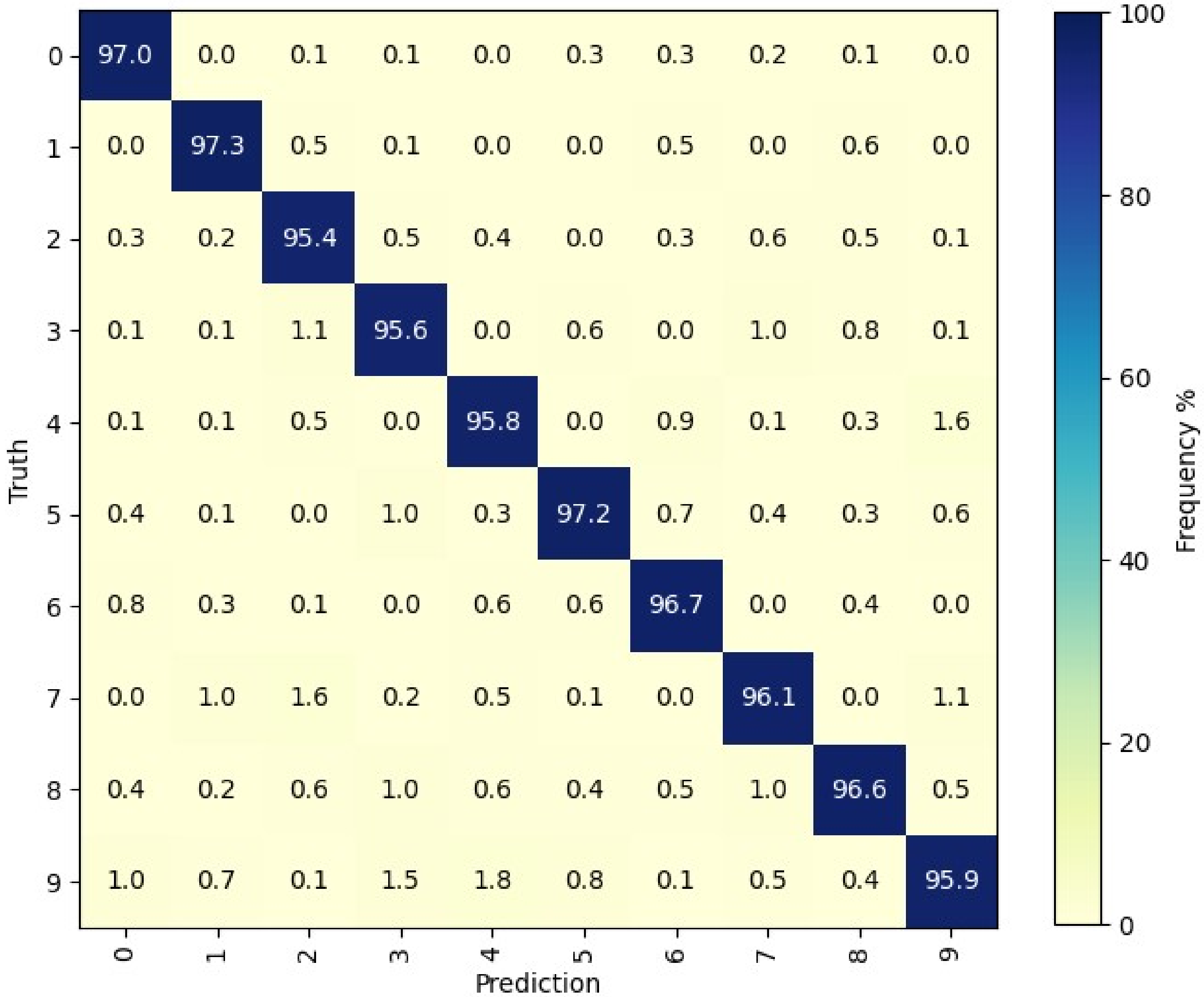
The following codeblocks define some helper functions for plotting. You don't need to touch them

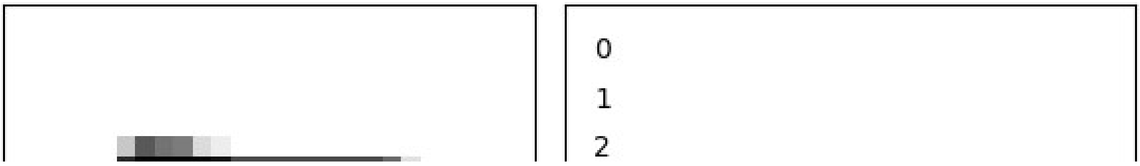
```
#@title
def plot_image(X, ax=None):
    """Plot an image X.

    Args:
        X (2D array): image, grayscale or RGB
        ax (None, optional): Description
    """
    if ax is None:
        ax = plt.gca()

    if (X.ndim == 2) or (X.shape[-1] == 1):
        ax.imshow(X.astype('uint8'), origin='upper', cmap=plt.cm.Greys)
    else:
        ax.imshow(X.astype('uint8'), origin='upper')

    ax.set(xticks=[], yticks=[])

def plot_prediction(Yp, X, y, classes=None, top_n=False):
    """Plot an image along with all or the top_n predictions.

    Args:
        Yp (1D array): predicted probabilities for each class
        X (2D array): image
        y (integer): true class label
```

```
        classes (1D array, optional): class names
        top_n (int, optional): number of top predictions to show
    """
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(6, 3.2))
    plt.subplots_adjust(left=0.02, right=0.98, bottom=0.15, top=0.98, wspace=0.02)
    plot_image(X, ax1)

    if top_n:
        n = top_n
        s = np.argsort(Yp)[-top_n:]
    else:
        n = len(Yp)
        s = np.arange(n)[::-1]

    patches = ax2.barh(np.arange(n), Yp[s], align='center')
    ax2.set(xlim=(0, 1), xlabel='Probability', yticks=[])

    for iy, patch in zip(s, patches):
        if iy == y:
            patch.set_facecolor('C1')  # color correct patch

    if classes is None:
        classes = np.arange(0, np.size(Yp))

    for i in range(n):
        ax2.text(0.05, i, classes[s][i], ha='left', va='center')

    plt.show()

def plot_confusion(yp, y, classes=None, fname=None):
    """Plot confusion matrix for given true and predicted class labels

    Args:
        yp (1D array): predicted class labels
        y (1D array): true class labels
        classes (1D array): class names
        fname (str, optional): filename for saving the plot
    """
    if classes is None:
        n = max(max(yp), max(y)) + 1
        classes = np.arange(n)
    else:
        n = len(classes)

    bins = np.linspace(-0.5, n - 0.5, n + 1)
    C = np.histogram2d(y, yp, bins=bins)[0]
    C = C / np.sum(C, axis=0) * 100

    fig = plt.figure(figsize=(8, 8))
    plt.imshow(C, interpolation='nearest', vmin=0, vmax=100, cmap=plt.cm.YlGnBu)
    plt.gca().set_aspect('equal')
    cbar = plt.colorbar(shrink=0.8)
    cbar.set_label('Frequency %')
    plt.xlabel('Prediction')
    plt.ylabel('Truth')
```

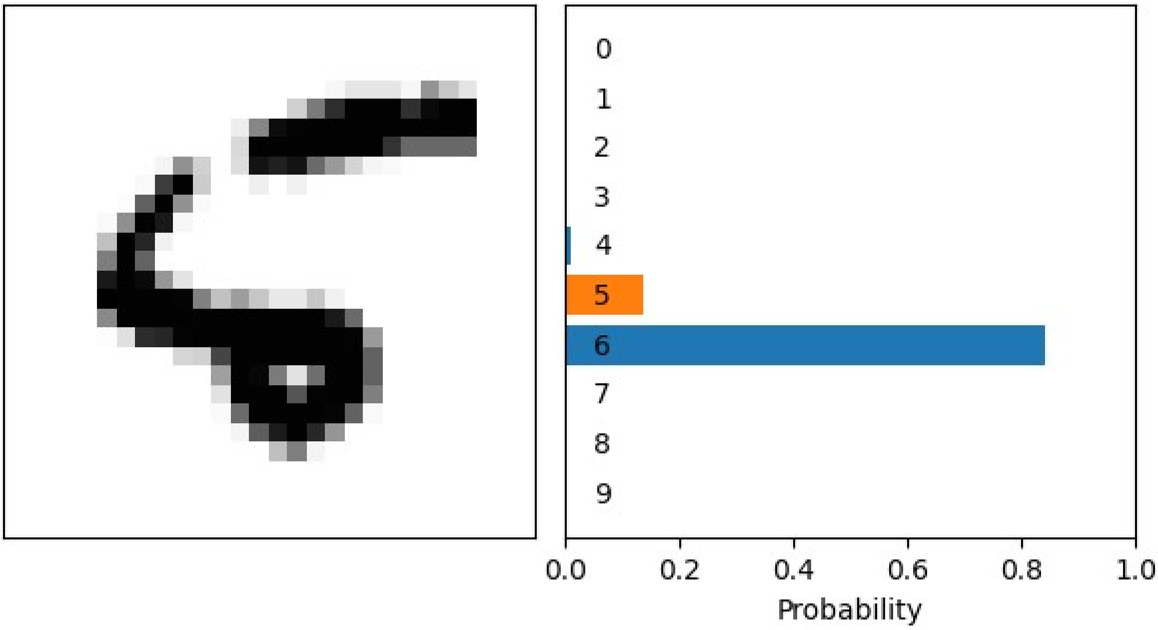
```
    plt.ylabel('Truth')
    plt.xticks(range(n), classes, rotation='vertical')
    plt.yticks(range(n), classes)
    for x in range(n):
        for y in range(n):
            if np.isnan(C[x, y]):
                continue
            color = 'white' if x == y else 'black'
            plt.annotate('%.1f' % (C[x, y]), xy=(y, x), color=color, ha='center', va='

    plt.show()


# plot a few examples, loop over test dataset:
# get missidentified samples
output = model.predict(x_test, batch_size=128)
labels = np.argmax(y_test_onehot, axis=1)
predictions = np.argmax(output, axis=1)
plot_confusion(predictions, labels)

for i in range(10): # loop over first 10 test samples
    plot_prediction(output[i],
                    255 * np.reshape(x_test[i], (28, 28)), # we need to reshape the da
                    labels[i])
```

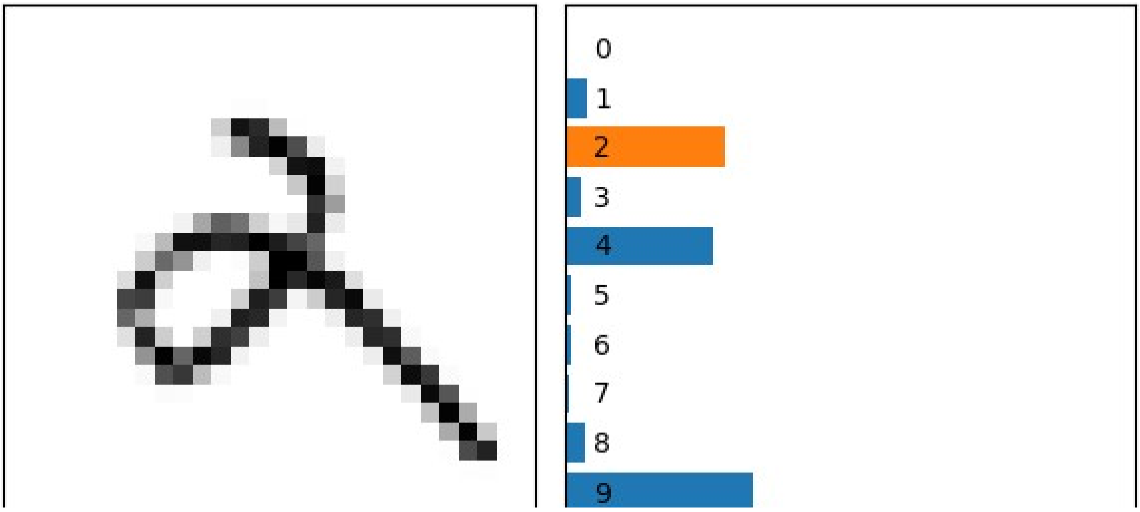
```
63/63 [==============================] - 0s 2ms/step
```

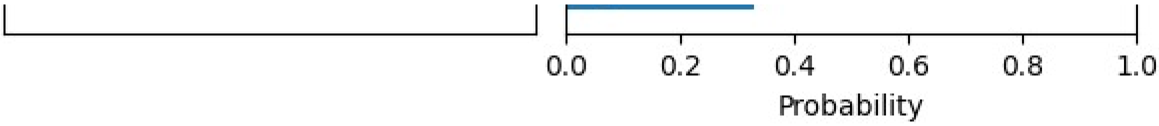


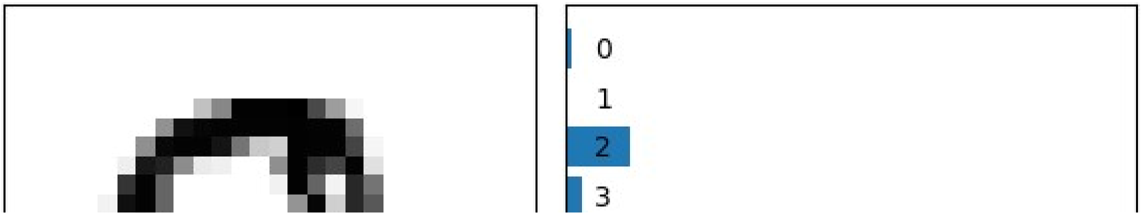



```
# now plot a few examples that were misclassified

indices_miss = np.nonzero(predictions != labels)[0]
x_missid = x_test[indices_miss]

for i in indices_miss[:10]:
    plot_prediction(output[i],
                    255 * np.reshape(x_test[i], (28, 28)), # we need to reshape the da
                    labels[i])

# or plot 32 of them in a joint plot
fig = plt.figure()
ax = fig.add_subplot(111)
plotdata = x_missid[:32]
plotdata = np.hstack(np.concatenate(np.reshape(plotdata, (4, 8, 28, 28)), axis=1))
ax.imshow(plotdata, cmap="gray")
```









Note

A shallow network seems to perform good on this task and making it deeper does not neccesarily make for a better model.

Grid search

Let's search the two hyperparameters dropout and number of nodes. You can start from the template below.

```
dropout_values = [0, 0.1, 0.3, 0.5, 0.8]
n_neurons_values = [16, 32, 64, 128, 256]
results_gridsearch = np.zeros((len(dropout_values), len(n_neurons_values), 2))
for iDrop, drop in enumerate(dropout_values):
  for iN, n_neurons in enumerate(n_neurons_values):
    model = tf.keras.models.Sequential([
        layers.Dense(n_neurons, input_dim=784, activation="relu"),
        layers.Dropout(drop),
        layers.Dense(10),
        layers.Activation('softmax')])  # softmax actication to transform output into

    model.compile(
        loss='categorical_crossentropy',  # the recommended loss for a classification
        optimizer="adam",
        metrics=['accuracy']) # we use accuracy to quanitfy to network performance.

    results = model.fit(
        x_train, y_train_onehot,
        validation_data=(x_valid, y_valid_onehot),
        batch_size=32,
        epochs=10,
        verbose=0
        )
    t = model.evaluate(x_test, y_test_onehot, verbose=0, batch_size=128)
    results_gridsearch[iDrop, iN] = t
    print(f"dropout = {drop:.2f}, {n_neurons} neurons -> accuracy {results_gridsearch[
```

```
dropout = 0.00, 16 neurons -> accuracy 0.915, error rate = 8.5%
dropout = 0.00, 32 neurons -> accuracy 0.919, error rate = 8.1%
dropout = 0.00, 64 neurons -> accuracy 0.919, error rate = 8.1%
```

```
dropout = 0.00, 128 neurons -> accuracy 0.914, error rate = 8.6%
dropout = 0.00, 256 neurons -> accuracy 0.915, error rate = 8.5%
dropout = 0.10, 16 neurons -> accuracy 0.918, error rate = 8.2%
dropout = 0.10, 32 neurons -> accuracy 0.921, error rate = 7.9%
dropout = 0.10, 64 neurons -> accuracy 0.921, error rate = 7.9%
dropout = 0.10, 128 neurons -> accuracy 0.919, error rate = 8.1%
dropout = 0.10, 256 neurons -> accuracy 0.915, error rate = 8.5%
dropout = 0.30, 16 neurons -> accuracy 0.913, error rate = 8.7%
dropout = 0.30, 32 neurons -> accuracy 0.918, error rate = 8.2%
dropout = 0.30, 64 neurons -> accuracy 0.917, error rate = 8.3%
dropout = 0.30, 128 neurons -> accuracy 0.919, error rate = 8.1%
dropout = 0.30, 256 neurons -> accuracy 0.914, error rate = 8.6%
dropout = 0.50, 16 neurons -> accuracy 0.905, error rate = 9.5%
dropout = 0.50, 32 neurons -> accuracy 0.916, error rate = 8.4%
dropout = 0.50, 64 neurons -> accuracy 0.919, error rate = 8.1%
dropout = 0.50, 128 neurons -> accuracy 0.919, error rate = 8.1%
dropout = 0.50, 256 neurons -> accuracy 0.918, error rate = 8.2%
dropout = 0.80, 16 neurons -> accuracy 0.881, error rate = 11.9%
dropout = 0.80, 32 neurons -> accuracy 0.899, error rate = 10.1%
dropout = 0.80, 64 neurons -> accuracy 0.909, error rate = 9.1%
dropout = 0.80, 128 neurons -> accuracy 0.914, error rate = 8.6%
dropout = 0.80, 256 neurons -> accuracy 0.918, error rate = 8.2%
```

Note

Best combination is 32 or 64 neurons with 10% dropout.

Random seach

Now lets implement a random search. A random search allows us to scan more hyperparameters at once without more computing time. You can start from the template below.

```
num_trials = [10, 20, 50, 100, 200] # number of trials
accs_est = []
for N in num_trials:
  best_accs=[]
  for _ in range(3):
    search = {
    'batch_size': np.random.choice([16, 32, 64, 128, 256], N),
    'num_neurons': np.random.choice([8, 32, 128, 256, 512], N),
    'learn_rate': np.random.choice([1e-5,1e-4, 1e-3, 1e-2, 1e-1],N),
    'activation': np.random.choice(['relu', 'elu', 'sigmoid', 'tanh'], N),
    'dropout': np.random.choice([0.0, 0.1, 0.2, 0.3, 0.5, 0.6], N),
    'val_acc': np.zeros(N)
    }

    accs=[]
    for i in range(N):
      # you can access the current value of the hyperparameter with `search['batch_siz
      model = tf.keras.models.Sequential([
                    tf.keras.layers.Dense(search["num_neurons"][i], input_dim=78
```

```
                                  tf.keras.layers.Dense(search["num_neurons"][i], input_dim=78
                                  tf.keras.layers.Dropout(search["dropout"][i]),
                                  tf.keras.layers.Dense(10, activation='softmax')])
            opt=tf.keras.optimizers.Adam(search["learn_rate"][i])
            model.compile(loss='categorical_crossentropy',
            optimizer=opt,
            metrics=['accuracy'])
            results = model.fit(
                  x_train, y_train_onehot,
                  validation_data=(x_valid, y_valid_onehot),
                  batch_size=search["batch_size"][i],
                  epochs=10,
                  verbose=0
                )

            search['val_acc'][i] = model.evaluate(x_test, y_test_onehot, verbose=0, batch_si
            accs.append(search['val_acc'][i])
            print(f"iteration {(i)}:")
            for key in search:
              print(f"\t{key}: {search[key][i]}")
            print(f"\t-> accuracy {search['val_acc'][i]:.3f}, error rate = {100*(1-search['v
          best_accs.append(max(accs))
        accs_est.append((N, np.mean(best_accs), np.std(best_accs)))
print(accs_est)
```

```
print(accs_est)
```

    [(10, 0.9751666784286499, 0.006387113629864917), (20, 0.9797083338101705, 0.000831

[(10, 0.9751666784286499, 0.006387113629864917), (20, 0.9797083338101705, 0.0008312598422263999), (50, 0.9819583495457967, 0.0003118087330096271), (100, 0.9810000061988831, 0.0011501819915849983)]

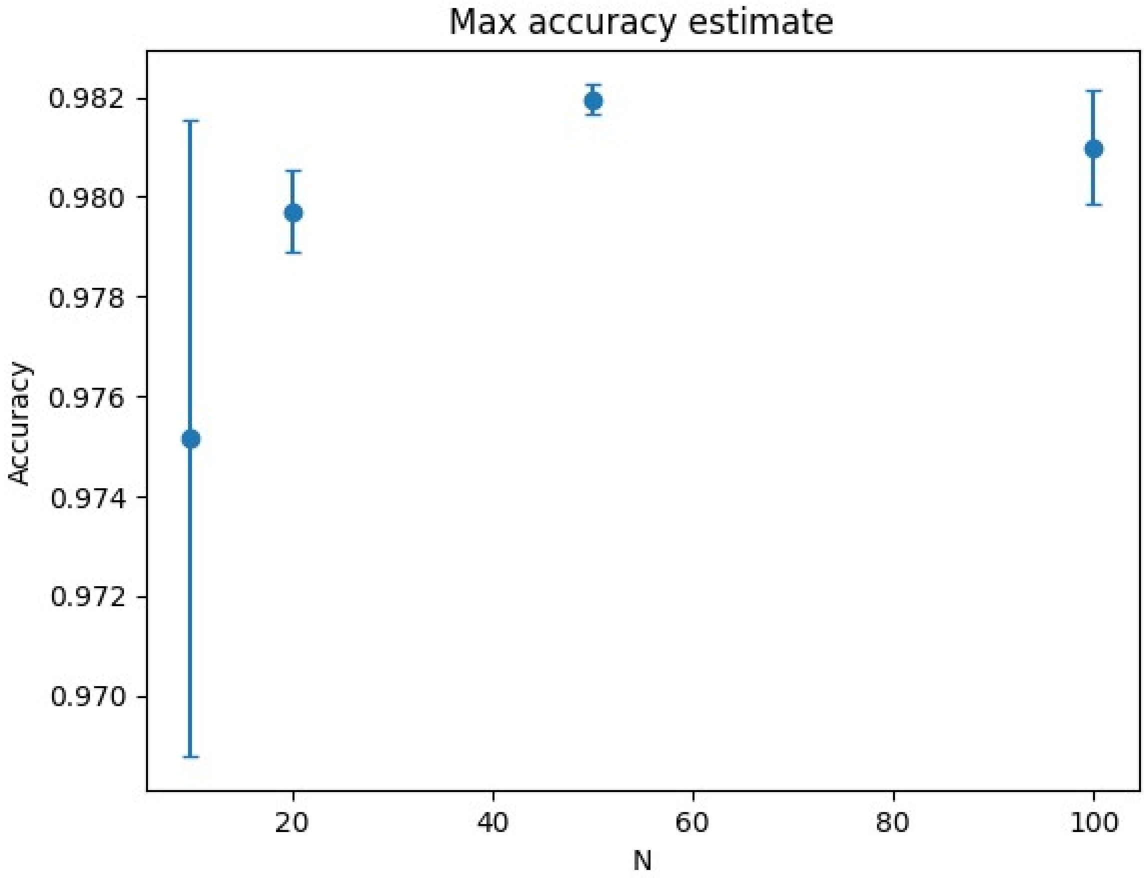
<matplotlib.image.AxesImage at 0x2505b68c5e0>

Note

So I thought we were supposed to plot accuracy and didn't realize my mistake until the code had run for ~10 hrs. If I instead wanted to evaluate loss I would instead have written `metrics=["loss"]` and also take `min` instead of `max`. My way is still a valid way of validating.

```
X=[10, 20, 50, 100]
Y=[0.9751666784286499, 0.9797083338101705, 0.9819583495457967, 0.9810000061988831]
std=[0.006387113629864917, 0.0008312598422263999, 0.0003118087330096271, 0.00115018199
plt.figure()
plt.errorbar(X,Y,std, linestyle="none", marker="o", capsize=3)
plt.xlabel("N")
plt.ylabel("Accuracy")
plt.title("Max accuracy estimate")
```

```
Text(0.5, 1.0, 'Max accuracy estimate')
```



From our independendent trials we see that 50 trials is sufficient to get a really good estimate of the best possible network. Note that for only 10 trials we still get a decent estimate. The errorbars represent one standard deviation but remember that we only use 3 values to estimate this.