NF264- Project 2: Digit recognizer

We are working for a small company that provides machine learning solutions for its customers. The postal office needs an AI system to automatically deliver mail. As a part of the system, they need a computer program that recognises handwritten digits. We are providing this program and as machine learning experts, we write the code that produces a classifier and this report that describes what we have done.

In [1]:
```python
import pandas as pd
import numpy as np
import plotly.express as px
import tensorflow as tf
from sklearnex import patch_sklearn
patch_sklearn()
```

```
C:\Users\elias\Miniconda3\envs\tf\lib\site-packages\tensorflow\python\framework\dtypes.py:523: FutureWarning:

Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be und
erstood as (type, (1,)) / '(1,)type'.

C:\Users\elias\Miniconda3\envs\tf\lib\site-packages\tensorflow\python\framework\dtypes.py:524: FutureWarning:

Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be und
erstood as (type, (1,)) / '(1,)type'.

C:\Users\elias\Miniconda3\envs\tf\lib\site-packages\tensorflow\python\framework\dtypes.py:525: FutureWarning:

Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be und
erstood as (type, (1,)) / '(1,)type'.

C:\Users\elias\Miniconda3\envs\tf\lib\site-packages\tensorflow\python\framework\dtypes.py:526: FutureWarning:

Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be und
erstood as (type, (1,)) / '(1,)type'.

C:\Users\elias\Miniconda3\envs\tf\lib\site-packages\tensorflow\python\framework\dtypes.py:527: FutureWarning:

Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be und
erstood as (type, (1,)) / '(1,)type'.

C:\Users\elias\Miniconda3\envs\tf\lib\site-packages\tensorflow\python\framework\dtypes.py:532: FutureWarning:

Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version of numpy, it will be und
erstood as (type, (1,)) / '(1,)type'.

Intel(R) Extension for Scikit-learn* enabled (https://github.com/intel/scikit-learn-intelex)
```

The Dataset

The MNIST dataset consist of 70000 images of handwritten digits. Each image consist of a 28x28 pixel images with a grayscale value between 0-255. They are given as a list of 70000 with each list having length 28x28 = 784. Which is confirmed by the shape printed below.

In [2]:
```
X = pd.read_csv('handwritten_digits_images.csv', header=None).to_numpy()
y = pd.read_csv('handwritten_digits_labels.csv', header=None).to_numpy()
print(X.shape)
```

```
(70000, 784)
```

Preprocessing steps

The labels of each images is represented as a digit between 0 and 9. We can make this label categorical, meaning they are all represented the same way as a bit array with 10 elements, where for example 4 is a 1 at the 5th index.

We also want to normalize the grayscale values from 0-255 to 0-1

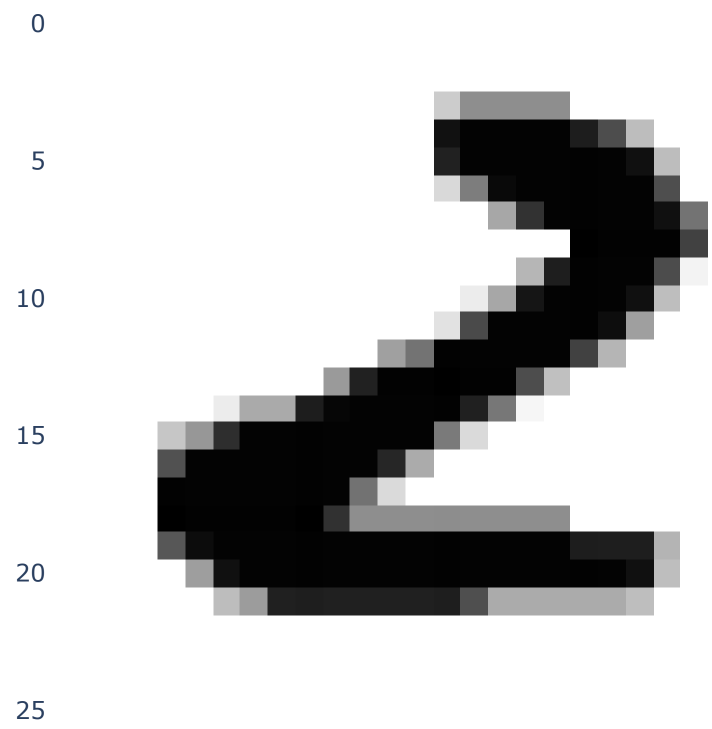
In [3]:
```
from keras.utils import to_categorical

y = to_categorical(y)

# Normalize to range 0-1
X = X.astype('float32')
X = X / 255.0
```

```
Using TensorFlow backend.
```

Here is an example of a image and its corresponding label.

In [4]:
```python
print(y[15000])

px.imshow(X[15000].reshape(28,28), color_continuous_scale=["white", "black"])
```

```
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
```



Splitting data

We split the data in 80% training data, 10% validation data used for evaluating and tuning hyperparameters, and 10% unseen test data which is used to choose the best model.

In [5]:
```
from sklearn.model_selection import train_test_split

X_train, X_val_test, y_train, y_val_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_val_test, y_val_test, test_size=0.5, random_state=42)

print('Train: X=%s, y=%s' % (X_train.shape, y_train.shape))
print('Test: X=%s, y=%s' % (X_test.shape, y_test.shape))
print('Validate: X=%s, y=%s' % (X_val.shape, y_val.shape))
```

```
Train: X=(56000, 784), y=(56000, 10)
Test: X=(7000, 784), y=(7000, 10)
Validate: X=(7000, 784), y=(7000, 10)
```

Candidate algorithms and choice of candidate hyperparameters (and why were the others left out)

We want to chose a classifier algorithm since a classifier utilizes some training data to understand how given input variables relate to a class. In this case, pictures of integers 0-9 are used as the training data. When the classifier is trained accurately, it can be used to detect integers for the Postal office. There are many candidates in this space.

Firstly, we choose K Nearest Neighbors Classifier since this is kind of a baseline model, and we have implemnted this model from scratch is previous courses so we are well aware of the algorithm.

Secondly, we want to test a descision tree classifier since we know that this is an effective Classifier from our previous Project, where we classified a dataset with 10 features. When we think about a writtin digit, there are probably some descisions thart could be made in a descision tree, such as if it has a single line in vertical direction it is a 1 or 7, or if it contains two circles it is a 8. From the pixel data we ecpect there will be some kind of denominator that could classify the image into a category of digits.

Lastly, we want to explore a Sequential Convolutional Neural Network Classifier since we are not as familier with this tool, and want to learn more about implemtning this Classifier. Neural Nets can be very powerfull if trained accuratly, so we want to explore if this could be a feasible solution for recognizing digits.

Chosen performance measure

When chosing perfomance measure there are several that could be used, i.e MSE and RMSE, but we want to use the accuracy in percentage (0-100%) on the test data for model selection, and the accuracy on validation data for model evaluation.

K Nearest Neighbors Classifier

The K-nearest neighbors (KNN) algorithm is a data classification method for estimating the probability that a data point will become a member of one or another group based on which group the data points are closest to it. A classification problem has a discrete value as its output. It is a type of supervised machine learning algorithm used to solve classification (and regression) problems. The algorithm is also called a lazy learning and non-parametic algorithm. This is because it is lazy and does not preform any training when you supply the training data. It just stores the data during the training time and does not perform any calculations. The algorithm does not build a model until a query is performed on the data set. It is considered a non-parametric methods because it does not make any assumptions about the underlying data distribution. It also involves classifying a data point by looking at the nearest annotated data point.

A advantage of using it, is that the training phase of K-nearest neighbor classification is much faster compared to other classification algorithms. There is no need to train a model for generalization, that is why KNN is known as the simple and instance-based learning algorithm. One disadvantage of using KNN is that the testing phase of K-nearest neighbor classification is slower and costlier in terms of time and memory. It requires large memory for storing the entire training dataset for prediction.

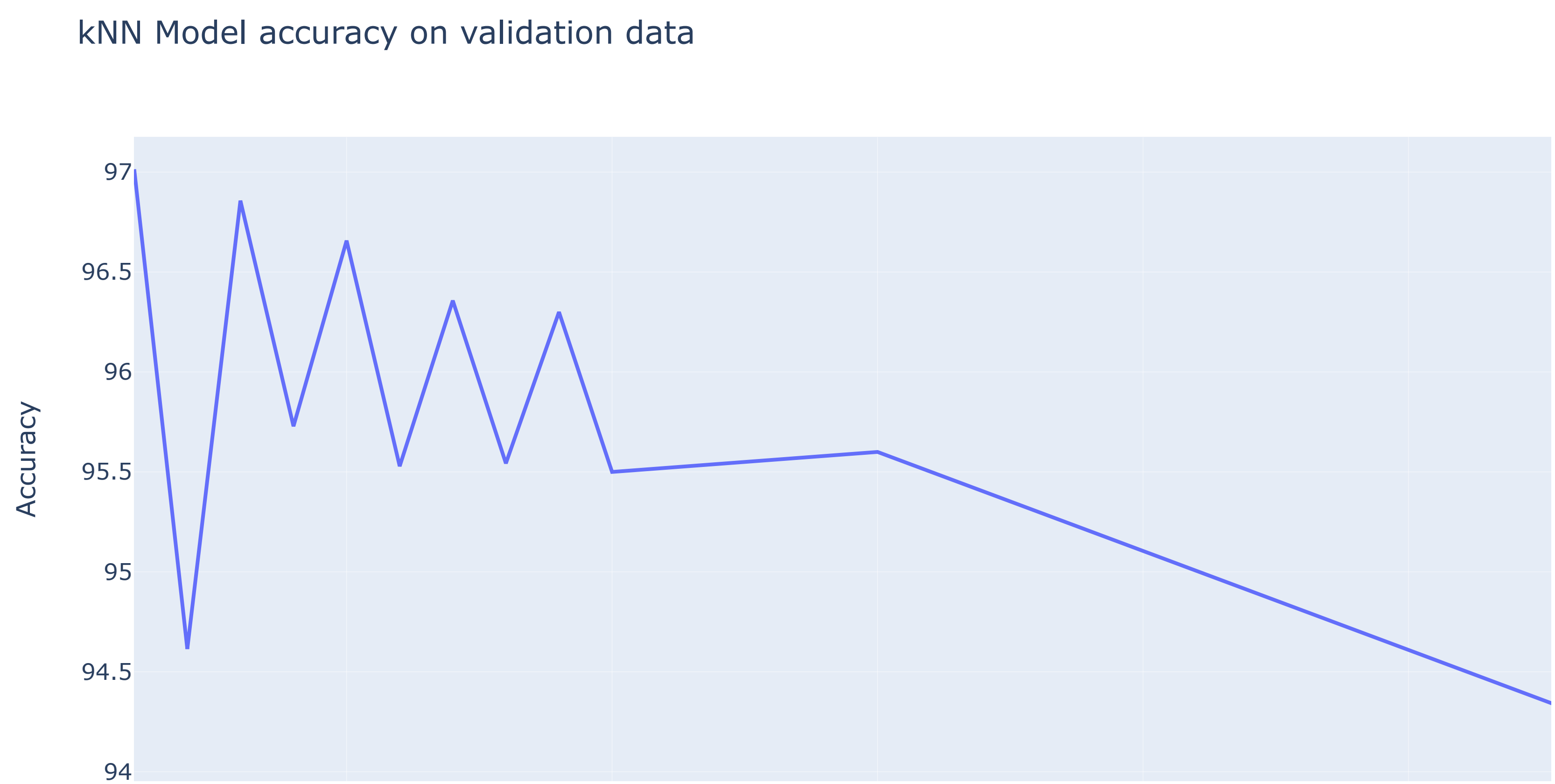
Hyperparameters

We tune the number of nearest neighbors k to asses for choosing the label of the each image.

In [6]:
```
from sklearn.neighbors import KNeighborsClassifier

kVals = [1, 2, 3, 4, 5, 6, 7, 8 , 9, 10, 15, 30]
accuracies = []

for k in kVals: # Testing many k hyperparametyers to optimize performance
        model = KNeighborsClassifier(algorithm='auto', n_neighbors=k)
        model.fit(X_train, y_train)

        score = model.score(X_val, y_val)
        print("k=%d, validation accuracy=%.2f%%" % (k, score * 100))
        accuracies.append([k, score * 100])
```

```
k=1, validation accuracy=97.01%
k=2, validation accuracy=94.61%
k=3, validation accuracy=96.86%
k=4, validation accuracy=95.73%
k=5, validation accuracy=96.66%
k=6, validation accuracy=95.53%
k=7, validation accuracy=96.36%
k=8, validation accuracy=95.54%
k=9, validation accuracy=96.30%
k=10, validation accuracy=95.50%
k=15, validation accuracy=95.60%
k=30, validation accuracy=94.11%
```

In [7]:
```
#Plotting data
df = pd.DataFrame(accuracies, columns = ['k', 'Accuracy'])
px.line(df, x="k", y = 'Accuracy', title="kNN Model accuracy on validation data")
```



In [8]:
```python
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score

model = KNeighborsClassifier(n_neighbors=1)
model.fit(X_train, y_train)
predictions = model.predict(X_test)

print("Evaluation of test data")
print(classification_report(y_test, predictions))
print("sklearn KNeighborsClassifier Test data accuracy: {:3.2f}%".format(accuracy_score(y_test, predictions)*
100))
```

```
Evaluation of test data
             precision    recall  f1-score   support

          0       0.98      0.99      0.99       685
          1       0.97      0.99      0.98       778
          2       0.98      0.97      0.98       671
          3       0.97      0.96      0.96       690
          4       0.98      0.97      0.98       733
          5       0.96      0.96      0.96       644
          6       0.98      0.98      0.98       729
          7       0.96      0.97      0.97       694
          8       0.99      0.94      0.96       670
          9       0.96      0.97      0.96       706

   micro avg       0.97      0.97      0.97      7000
   macro avg       0.97      0.97      0.97      7000
weighted avg       0.97      0.97      0.97      7000
 samples avg       0.97      0.97      0.97      7000

sklearn KNeighborsClassifier Test data accuracy: 97.14%
```

kNN Findings

We found best results with k=1 which gives us a accuracy on test of 97,14% on test data.

Decision Tree Classifier

Decision tree is a supervised machine learning algorithm that uses a set of rules to make decisions. A decision tree has a flowchart-like tree structure where an internal node represents feature, the branch represents a decision rule and each leaf node represents the outcome.

The most important feature is the capability of capturing descriptive decisionmaking knowledge from the supplied data. A decision tree can be generated from training sets. Decision tree classifier generates the actual prediction at the leaf nodes, more information can be stored at the leaf nodes. The algorithm is a distribution-free or non-parametric method, which does not depend upon probability distribution assumptions. It can handle high dimensional data with good accuracy.

Hyperparameters

As implemented in project 1 there are mainly 2 parameters to tweak. The impurity measure gini or entropy, and the max depth of the tree.

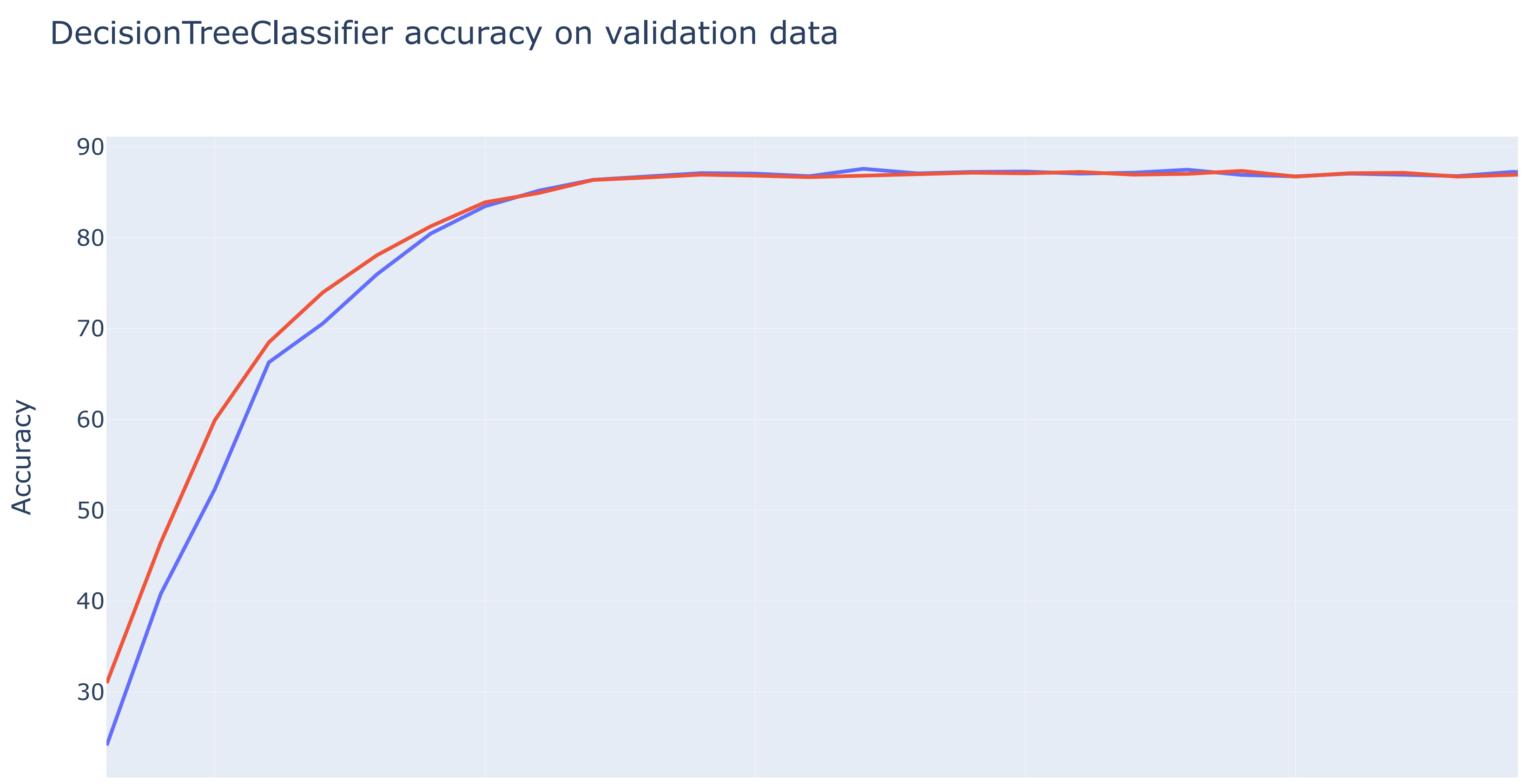
In [9]:
```
from sklearn.tree import DecisionTreeClassifier

criterion = ['gini','entropy']
max_depth = list(range(3, 31))

df = []
for d in max_depth:
    for c in criterion:
        dtc = DecisionTreeClassifier(criterion=c, max_depth=d)
        dtc.fit(X_train, y_train)
        acc = accuracy_score(y_val, dtc.predict(X_val))*100
        df.append([d,c,acc])
        print("Descicion tree with max depth: "+str(d) + ", impurity measure: "+c+", Accuracy: {:3.2f}%".form
at(acc))
```

```
Descicion tree with max depth: 3, impurity measure: gini, Accuracy: 24.13%
Descicion tree with max depth: 3, impurity measure: entropy, Accuracy: 30.99%
Descicion tree with max depth: 4, impurity measure: gini, Accuracy: 40.80%
Descicion tree with max depth: 4, impurity measure: entropy, Accuracy: 46.44%
Descicion tree with max depth: 5, impurity measure: gini, Accuracy: 52.33%
Descicion tree with max depth: 5, impurity measure: entropy, Accuracy: 59.91%
Descicion tree with max depth: 6, impurity measure: gini, Accuracy: 66.29%
Descicion tree with max depth: 6, impurity measure: entropy, Accuracy: 68.49%
Descicion tree with max depth: 7, impurity measure: gini, Accuracy: 70.57%
Descicion tree with max depth: 7, impurity measure: entropy, Accuracy: 73.99%
Descicion tree with max depth: 8, impurity measure: gini, Accuracy: 75.97%
Descicion tree with max depth: 8, impurity measure: entropy, Accuracy: 78.07%
Descicion tree with max depth: 9, impurity measure: gini, Accuracy: 80.47%
Descicion tree with max depth: 9, impurity measure: entropy, Accuracy: 81.27%
Descicion tree with max depth: 10, impurity measure: gini, Accuracy: 83.46%
Descicion tree with max depth: 10, impurity measure: entropy, Accuracy: 83.90%
Descicion tree with max depth: 11, impurity measure: gini, Accuracy: 85.19%
Descicion tree with max depth: 11, impurity measure: entropy, Accuracy: 84.93%
Descicion tree with max depth: 12, impurity measure: gini, Accuracy: 86.37%
Descicion tree with max depth: 12, impurity measure: entropy, Accuracy: 86.36%
Descicion tree with max depth: 13, impurity measure: gini, Accuracy: 86.74%
Descicion tree with max depth: 13, impurity measure: entropy, Accuracy: 86.63%
Descicion tree with max depth: 14, impurity measure: gini, Accuracy: 87.11%
Descicion tree with max depth: 14, impurity measure: entropy, Accuracy: 86.94%
Descicion tree with max depth: 15, impurity measure: gini, Accuracy: 87.06%
Descicion tree with max depth: 15, impurity measure: entropy, Accuracy: 86.83%
Descicion tree with max depth: 16, impurity measure: gini, Accuracy: 86.77%
Descicion tree with max depth: 16, impurity measure: entropy, Accuracy: 86.67%
Descicion tree with max depth: 17, impurity measure: gini, Accuracy: 87.59%
Descicion tree with max depth: 17, impurity measure: entropy, Accuracy: 86.83%
Descicion tree with max depth: 18, impurity measure: gini, Accuracy: 87.09%
Descicion tree with max depth: 18, impurity measure: entropy, Accuracy: 86.97%
Descicion tree with max depth: 19, impurity measure: gini, Accuracy: 87.24%
Descicion tree with max depth: 19, impurity measure: entropy, Accuracy: 87.16%
Descicion tree with max depth: 20, impurity measure: gini, Accuracy: 87.29%
Descicion tree with max depth: 20, impurity measure: entropy, Accuracy: 87.09%
Descicion tree with max depth: 21, impurity measure: gini, Accuracy: 87.06%
Descicion tree with max depth: 21, impurity measure: entropy, Accuracy: 87.24%
Descicion tree with max depth: 22, impurity measure: gini, Accuracy: 87.16%
Descicion tree with max depth: 22, impurity measure: entropy, Accuracy: 86.94%
Descicion tree with max depth: 23, impurity measure: gini, Accuracy: 87.49%
Descicion tree with max depth: 23, impurity measure: entropy, Accuracy: 87.03%
Descicion tree with max depth: 24, impurity measure: gini, Accuracy: 86.91%
```

```
Descicion tree with max depth: 24, impurity measure: entropy, Accuracy: 87.36%
Descicion tree with max depth: 25, impurity measure: gini, Accuracy: 86.76%
Descicion tree with max depth: 25, impurity measure: entropy, Accuracy: 86.73%
Descicion tree with max depth: 26, impurity measure: gini, Accuracy: 87.06%
Descicion tree with max depth: 26, impurity measure: entropy, Accuracy: 87.10%
Descicion tree with max depth: 27, impurity measure: gini, Accuracy: 86.93%
Descicion tree with max depth: 27, impurity measure: entropy, Accuracy: 87.14%
Descicion tree with max depth: 28, impurity measure: gini, Accuracy: 86.79%
Descicion tree with max depth: 28, impurity measure: entropy, Accuracy: 86.73%
Descicion tree with max depth: 29, impurity measure: gini, Accuracy: 87.24%
Descicion tree with max depth: 29, impurity measure: entropy, Accuracy: 86.90%
Descicion tree with max depth: 30, impurity measure: gini, Accuracy: 87.17%
Descicion tree with max depth: 30, impurity measure: entropy, Accuracy: 87.10%
```

In [10]:

```
#Plotting data
df = pd.DataFrame(df, columns = ['Depth', 'Impurity', 'Accuracy'])
px.line(df, x="Depth", y = 'Accuracy', color='Impurity', title="DecisionTreeClassifier accuracy on validation data")
```



We chose the best hyperparameters based on the graph above.

In [11]:
```
DecisionTree = DecisionTreeClassifier()
DecisionTree.fit(X_train, y_train)
predictions = DecisionTree.predict(X_test)

print("Evaluation of test data")
print(classification_report(y_test, predictions))

print("sklearn DecisionTreeClassifier Test data accuracy: {:3.2f}%".format(accuracy_score(y_test, predictions
)*100))
```

```
Evaluation of test data
              precision    recall  f1-score   support

           0       0.91      0.91      0.91       685
           1       0.94      0.96      0.95       778
           2       0.85      0.87      0.86       671
           3       0.85      0.86      0.85       690
           4       0.89      0.87      0.88       733
           5       0.81      0.80      0.81       644
           6       0.92      0.89      0.91       729
           7       0.90      0.91      0.90       694
           8       0.82      0.79      0.81       670
           9       0.83      0.83      0.83       706

   micro avg       0.87      0.87      0.87      7000
   macro avg       0.87      0.87      0.87      7000
weighted avg       0.87      0.87      0.87      7000
 samples avg       0.87      0.87      0.87      7000

sklearn DecisionTreeClassifier Test data accuracy: 87.26%
```

We found best results with impurity measure entropy and depth of 30 which gives us a accuracy on test of about 87% on test data. Given the kNN baseline performed 97% accuracy on test data, we can discard this model since it does not perform better.

Sequential Convolutional Neural Network Classifier

Convolutional networks (CNN) are a specialized type of neural networks that use convolution in place of general matrix multiplication in at least one of their layers. CNNs consists of an input layer, hidden layers and a output layer. The hidden layers are called hidden, because their inputs and outputs are masked by the activation function and final convolution. This usually means that it includes a layer that preforms a dot product of the convolution kernel with the layers input matrix.

CNNs use relatively little pre-processing compared to other image classification algorithms, which means that the networks learns to optimize the filters (or kernel) trough automated learning, while in traditional algorithms these filters are hand-engineered. CNNs take advantage of the hierarchial pattern in data. It assemble patterns of increasing complexity unsing smaller and simpler patterns embossed in their filters. Convolutional networks are a specialized type of neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

Hyperparameters

Layers of the model is described below

For activation function we use ReLU and the He weight, which is both commonly recognized as the best practice.

We are using a stochastic gradient descent optimizer with the model. The learning rate is set to 0.01 and a momentum is set to 0.9. Since we have a multiclass classificatin probelm it is beneficiary to use a categorical cross-entropy loss function. We evaluate this with the validation dataset accuracy.

In [12]:
```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.optimizers import Adam
from sklearn.model_selection import KFold
from keras.layers import BatchNormalization

print(tf.__version__)

if tf.test.gpu_device_name():
    print('GPU Device:{}'.format(tf.test.gpu_device_name()))
```

```
1.10.0
GPU Device:/device:GPU:0
```

We need to reshape the image data for the nural network to accept it as a single color channel.

In [13]:
```
X = X.reshape(X.shape[0], 28, 28, 1)
X_train, X_val_test, y_train, y_val_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_val_test, y_val_test, test_size=0.5, random_state=42)
```

Now let us define a model. The Sequential model has a frontend with pooling layers that extract features and a backend that actually makes the prediction. For the first part we can start with a 3x3 convolutional layer and 32 filters. Then we can use a max pooling layer which calculates the maximum value for each patch of the feature map. This can then be flattened to express different features for the classifier.

Since we know that the model should output the probabaility of the given image being one of the digits 0-9 we know that the output layer should be a probability distrubituon of 10 classes. To get the probability we use a softmax activation function. Before the output layer we use a dense layer with 100 nodes to be fitted by the flattened feature layer.

In [14]:
```
def neural_net_1(X_train, y_train, X_val, y_val, X_test, y_test, epochs = 5):
        model = Sequential()
        model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', input_shape=(28, 28, 
1)))
        model.add(MaxPooling2D((2, 2)))
        model.add(Flatten())
        model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
        model.add(Dense(10, activation='softmax')) # Output layer of 10 integers
        opt = SGD(lr=0.01, momentum=0.9)
        model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])

        model.fit(X_train, y_train, epochs=epochs, batch_size=64, validation_data=(X_val, y_val), verbose=1)
        _, acc = model.evaluate(X_test, y_test, verbose=1)
        print('Model accuracy on test data:  %.3f' % (acc * 100.0))

        return model

neural_net = neural_net_1(X_train, y_train, X_val, y_val, X_test, y_test, epochs=15)
```

```
Train on 56000 samples, validate on 7000 samples
Epoch 1/15
56000/56000 [==============================] - 10s 177us/step - loss: 0.2122 - acc: 0.9359 - val_loss: 0.1091
- val_acc: 0.9661
Epoch 2/15
56000/56000 [==============================] - 9s 154us/step - loss: 0.0800 - acc: 0.9769 - val_loss: 0.0777
- val_acc: 0.9750
Epoch 3/15
56000/56000 [==============================] - 9s 156us/step - loss: 0.0511 - acc: 0.9848 - val_loss: 0.0627
- val_acc: 0.9806
Epoch 4/15
56000/56000 [==============================] - 9s 154us/step - loss: 0.0383 - acc: 0.9889 - val_loss: 0.0581
- val_acc: 0.9836
Epoch 5/15
56000/56000 [==============================] - 9s 155us/step - loss: 0.0281 - acc: 0.9919 - val_loss: 0.0528
- val_acc: 0.9841
Epoch 6/15
56000/56000 [==============================] - 9s 154us/step - loss: 0.0225 - acc: 0.9934 - val_loss: 0.0518
- val_acc: 0.9834: 0s - loss: 0.0226 - ac - ETA: 0s - loss: 0.0225 - ac
Epoch 7/15
56000/56000 [==============================] - 9s 154us/step - loss: 0.0171 - acc: 0.9951 - val_loss: 0.0519
- val_acc: 0.9853
Epoch 8/15
56000/56000 [==============================] - 9s 155us/step - loss: 0.0135 - acc: 0.9965 - val_loss: 0.0514
- val_acc: 0.9846oss: 0.0134 - acc: 0
Epoch 9/15
56000/56000 [==============================] - 9s 156us/step - loss: 0.0101 - acc: 0.9976 - val_loss: 0.0513
- val_acc: 0.9850
Epoch 10/15
56000/56000 [==============================] - 9s 155us/step - loss: 0.0078 - acc: 0.9986 - val_loss: 0.0483
- val_acc: 0.9851
Epoch 11/15
56000/56000 [==============================] - 9s 155us/step - loss: 0.0065 - acc: 0.9988 - val_loss: 0.0533
- val_acc: 0.9841
Epoch 12/15
56000/56000 [==============================] - 9s 156us/step - loss: 0.0051 - acc: 0.9991 - val_loss: 0.0512
- val_acc: 0.9849
Epoch 13/15
56000/56000 [==============================] - 9s 156us/step - loss: 0.0038 - acc: 0.9995 - val_loss: 0.0512
- val_acc: 0.9859
Epoch 14/15
56000/56000 [==============================] - 9s 157us/step - loss: 0.0030 - acc: 0.9997 - val_loss: 0.0511
- val_acc: 0.98740. - ETA: 0s - loss: 0.0030 - acc: 0.99 - ETA: 0s - loss: 0.0030 - acc:
```

```
Epoch 15/15
56000/56000 [==============================] - 9s 157us/step - loss: 0.0025 - acc: 0.9999 - val_loss: 0.0549
- val_acc: 0.9864s - loss: 0.0016 - acc: 1. - ETA: - ETA: 5s - loss: 0.001
7000/7000 [==============================] - 1s 96us/step
Model accuracy on test data:  98.757
```

This model gives a very good score with an accuracy of 98.729% on test data. We could increse the depth of the forntend (feature extraction) layers to see if we get better results.

Increase depth

To increase the accuracy of feature extraction we add a double convolutional layer, with 64 filters. And as previous wee use a MxPooling layer to collect the features.

In [15]:
```
def neural_net_2(X_train, y_train, X_val, y_val, X_test, y_test, epochs = 5):
        model = Sequential()
        model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', input_shape=(28, 28,
1)))
        model.add(MaxPooling2D((2, 2)))
        model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform'))
        model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform'))
        model.add(MaxPooling2D((2, 2)))
        model.add(Flatten())
        model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
        model.add(Dense(10, activation='softmax'))
        opt = SGD(lr=0.01, momentum=0.9)
        model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])

        model.fit(X_train, y_train, epochs=epochs, batch_size=64, validation_data=(X_val, y_val), verbose=1)
        _, acc = model.evaluate(X_test, y_test, verbose=1)
        print('Model accuracy on test data:  %.3f' % (acc * 100.0))

        return model

neural_net = neural_net_2(X_train, y_train, X_val, y_val, X_test, y_test, epochs=15)
```

```
Train on 56000 samples, validate on 7000 samples
Epoch 1/15
56000/56000 [==============================] - 12s 223us/step - loss: 0.1550 - acc: 0.9513 - val_loss: 0.0747
- val_acc: 0.9743
Epoch 2/15
56000/56000 [==============================] - 12s 220us/step - loss: 0.0483 - acc: 0.9851 - val_loss: 0.0442
- val_acc: 0.9861s - l - ET - ETA: 2s - - ETA: 1s - loss: 0
Epoch 3/15
56000/56000 [==============================] - 12s 223us/step - loss: 0.0346 - acc: 0.9894 - val_loss: 0.0457
- val_acc: 0.9850
Epoch 4/15
56000/56000 [==============================] - 13s 224us/step - loss: 0.0257 - acc: 0.9916 - val_loss: 0.0406
- val_acc: 0.9879
Epoch 5/15
56000/56000 [==============================] - 13s 226us/step - loss: 0.0192 - acc: 0.9940 - val_loss: 0.0402
- val_acc: 0.9876
Epoch 6/15
56000/56000 [==============================] - 13s 227us/step - loss: 0.0149 - acc: 0.9954 - val_loss: 0.0422
- val_acc: 0.98760s - loss: 0.0150 - ac
Epoch 7/15
56000/56000 [==============================] - 13s 229us/step - loss: 0.0114 - acc: 0.9963 - val_loss: 0.0504
- val_acc: 0.9857: 0.0 - ETA: 0s - loss: 0.0114 - acc:
Epoch 8/15
56000/56000 [==============================] - 13s 232us/step - loss: 0.0088 - acc: 0.9974 - val_loss: 0.0519
- val_acc: 0.9860
Epoch 9/15
56000/56000 [==============================] - 13s 235us/step - loss: 0.0071 - acc: 0.9976 - val_loss: 0.0488
- val_acc: 0.9877
Epoch 10/15
56000/56000 [==============================] - 13s 239us/step - loss: 0.0063 - acc: 0.9980 - val_loss: 0.0456
- val_acc: 0.9899
Epoch 11/15
56000/56000 [==============================] - 14s 243us/step - loss: 0.0041 - acc: 0.9989 - val_loss: 0.0467
- val_acc: 0.9896
Epoch 12/15
56000/56000 [==============================] - 14s 245us/step - loss: 0.0035 - acc: 0.9990 - val_loss: 0.0469
- val_acc: 0.9901
Epoch 13/15
56000/56000 [==============================] - 14s 242us/step - loss: 0.0024 - acc: 0.9994 - val_loss: 0.0499
- val_acc: 0.9893
Epoch 14/15
56000/56000 [==============================] - 13s 230us/step - loss: 0.0027 - acc: 0.9992 - val_loss: 0.0439
- val_acc: 0.9909
```

```
Epoch 15/15
56000/56000 [==============================] - 13s 224us/step - loss: 0.0022 - acc: 0.9994 - val_loss: 0.0473
- val_acc: 0.9906 - loss: 0.0024 -  - ETA: 0s - loss: 0.0023
7000/7000 [==============================] - 1s 126us/step
Model accuracy on test data:  99.200
```

We see improved results with this Sequential model with an accuracy of 98.986% on test data, and therefore go forwards with cross calidating this model to see that it is not overfitted to the specific data split.

Cross Validation

In [16]:
```
def neural_net(X, y, epochs = 5, folds=3):
    scores = []
    hist = []
    kfold = KFold(folds, shuffle=True, random_state=1)

    for train_ix, val_ix in kfold.split(X):
        model = Sequential()
        model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', input_shape=(28, 28,
1)))
        model.add(MaxPooling2D((2, 2)))
        model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform'))
        model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform'))
        model.add(MaxPooling2D((2, 2)))
        model.add(Flatten())
        model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
        model.add(Dense(10, activation='softmax'))
        opt = SGD(lr=0.01, momentum=0.9)
        model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])

        trainX, trainY, valX, valY = X[train_ix], y[train_ix], X[val_ix], y[val_ix]
        model.fit(trainX, trainY, epochs=epochs, batch_size=32, validation_data=(valX, valY), verbose=1)

        _, acc = model.evaluate(valX, valY, verbose=1)
        print('> %.3f' % (acc * 100.0))

        scores.append(acc)
        hist.append(model)

    return scores, hist


final = neural_net(X,y, epochs=20, folds=5)
```

```
Train on 56000 samples, validate on 14000 samples
Epoch 1/20
56000/56000 [==============================] - 19s 343us/step - loss: 0.1349 - acc: 0.9584 - val_loss: 0.0679
- val_acc: 0.9789
Epoch 2/20
56000/56000 [==============================] - 19s 339us/step - loss: 0.0453 - acc: 0.9863 - val_loss: 0.0442
- val_acc: 0.9856
Epoch 3/20
56000/56000 [==============================] - 19s 347us/step - loss: 0.0305 - acc: 0.9901 - val_loss: 0.0351
- val_acc: 0.9894
Epoch 4/20
56000/56000 [==============================] - 20s 358us/step - loss: 0.0230 - acc: 0.9925 - val_loss: 0.0336
- val_acc: 0.9894
Epoch 5/20
56000/56000 [==============================] - 21s 367us/step - loss: 0.0174 - acc: 0.9943 - val_loss: 0.0364
- val_acc: 0.9888
Epoch 6/20
56000/56000 [==============================] - 21s 366us/step - loss: 0.0123 - acc: 0.9961 - val_loss: 0.0372
- val_acc: 0.9895
Epoch 7/20
56000/56000 [==============================] - 19s 345us/step - loss: 0.0095 - acc: 0.9972 - val_loss: 0.0349
- val_acc: 0.9900
Epoch 8/20
56000/56000 [==============================] - 20s 348us/step - loss: 0.0066 - acc: 0.9982 - val_loss: 0.0309
- val_acc: 0.9921
Epoch 9/20
56000/56000 [==============================] - 20s 358us/step - loss: 0.0067 - acc: 0.9978 - val_loss: 0.0369
- val_acc: 0.9909
Epoch 10/20
56000/56000 [==============================] - 21s 369us/step - loss: 0.0056 - acc: 0.9981 - val_loss: 0.0336
- val_acc: 0.9918
Epoch 11/20
56000/56000 [==============================] - 20s 354us/step - loss: 0.0045 - acc: 0.9986 - val_loss: 0.0343
- val_acc: 0.9916
Epoch 12/20
56000/56000 [==============================] - 19s 341us/step - loss: 0.0030 - acc: 0.9990 - val_loss: 0.0365
- val_acc: 0.9910
Epoch 13/20
56000/56000 [==============================] - 20s 348us/step - loss: 0.0030 - acc: 0.9991 - val_loss: 0.0427
- val_acc: 0.9896
Epoch 14/20
56000/56000 [==============================] - 20s 357us/step - loss: 0.0021 - acc: 0.9994 - val_loss: 0.0347
- val_acc: 0.9921
```

```
Epoch 15/20
56000/56000 [==============================] - 21s 367us/step - loss: 0.0014 - acc: 0.9997 - val_loss: 0.0399
- val_acc: 0.9914
Epoch 16/20
56000/56000 [==============================] - 21s 375us/step - loss: 0.0011 - acc: 0.9998 - val_loss: 0.0376
- val_acc: 0.9920
Epoch 17/20
56000/56000 [==============================] - 20s 350us/step - loss: 6.1121e-04 - acc: 0.9999 - val_loss: 0.
0341 - val_acc: 0.9928
Epoch 18/20
56000/56000 [==============================] - 20s 352us/step - loss: 3.5834e-04 - acc: 1.0000 - val_loss: 0.
0353 - val_acc: 0.9927
Epoch 19/20
56000/56000 [==============================] - 20s 361us/step - loss: 3.3564e-04 - acc: 1.0000 - val_loss: 0.
0358 - val_acc: 0.9927
Epoch 20/20
56000/56000 [==============================] - 21s 372us/step - loss: 3.2537e-04 - acc: 1.0000 - val_loss: 0.
0361 - val_acc: 0.9926
14000/14000 [==============================] - ETA:  - 2s 139us/step
> 99.264
Train on 56000 samples, validate on 14000 samples
Epoch 1/20
56000/56000 [==============================] - 21s 373us/step - loss: 0.1273 - acc: 0.9606 - val_loss: 0.0613
- val_acc: 0.9811
Epoch 2/20
56000/56000 [==============================] - 20s 348us/step - loss: 0.0465 - acc: 0.9851 - val_loss: 0.0493
- val_acc: 0.9872- ETA: 3s - loss: 0.04 - ETA: 0s - loss: 0.0466 - - ETA: 0s - loss: 0.0465 - acc: 0.
Epoch 3/20
56000/56000 [==============================] - 20s 353us/step - loss: 0.0311 - acc: 0.9901 - val_loss: 0.0471
- val_acc: 0.9871
Epoch 4/20
56000/56000 [==============================] - 20s 364us/step - loss: 0.0215 - acc: 0.9932 - val_loss: 0.0398
- val_acc: 0.9875
Epoch 5/20
56000/56000 [==============================] - 21s 377us/step - loss: 0.0156 - acc: 0.9947 - val_loss: 0.0504
- val_acc: 0.9864
Epoch 6/20
56000/56000 [==============================] - 21s 376us/step - loss: 0.0118 - acc: 0.9962 - val_loss: 0.0428
- val_acc: 0.9905
Epoch 7/20
56000/56000 [==============================] - 20s 352us/step - loss: 0.0091 - acc: 0.9972 - val_loss: 0.0433
- val_acc: 0.9896
Epoch 8/20
```

```
56000/56000 [==============================] - 19s 340us/step - loss: 0.0065 - acc: 0.9978 - val_loss: 0.0410
- val_acc: 0.9899
Epoch 9/20
56000/56000 [==============================] - 20s 349us/step - loss: 0.0062 - acc: 0.9978 - val_loss: 0.0451
- val_acc: 0.9909: 1s - loss: 0.0063 - acc - ETA: 1s - loss: 0.0 - ETA: 0s - loss: 0.0063 - acc:
Epoch 10/20
56000/56000 [==============================] - 20s 361us/step - loss: 0.0063 - acc: 0.9978 - val_loss: 0.0441
- val_acc: 0.9894
Epoch 11/20
56000/56000 [==============================] - 20s 365us/step - loss: 0.0040 - acc: 0.9987 - val_loss: 0.0515
- val_acc: 0.9889
Epoch 12/20
56000/56000 [==============================] - 20s 350us/step - loss: 0.0017 - acc: 0.9996 - val_loss: 0.0436
- val_acc: 0.9916
Epoch 13/20
56000/56000 [==============================] - 19s 341us/step - loss: 0.0015 - acc: 0.9995 - val_loss: 0.0485
- val_acc: 0.9911
Epoch 14/20
56000/56000 [==============================] - 20s 359us/step - loss: 5.8121e-04 - acc: 0.9998 - val_loss: 0.
0431 - val_acc: 0.9920
Epoch 15/20
56000/56000 [==============================] - 21s 372us/step - loss: 1.8772e-04 - acc: 1.0000 - val_loss: 0.
0461 - val_acc: 0.9921
Epoch 16/20
56000/56000 [==============================] - 20s 366us/step - loss: 8.2812e-05 - acc: 1.0000 - val_loss: 0.
0459 - val_acc: 0.9922- loss: 8.6647 - ETA: 1s - loss: 8.4059e-05 - acc: 1 - ETA: 0s - loss: 8.4463e-05 - ac
c: 1.00 - ETA: 0s - loss: 8.4252e-05 - acc: 1.0 - ETA: 0s - loss: 8.3852e-05 - acc - ETA: 0s - loss: 8.3311e-
05 - acc: 1.
Epoch 17/20
56000/56000 [==============================] - 20s 353us/step - loss: 4.1369e-05 - acc: 1.0000 - val_loss: 0.
0469 - val_acc: 0.9921 5s - loss: 4.4450e-0 - ETA: 4s - loss: 4.3960e-05 - ac - ETA: 3s - loss: 4.3116e-05 -
acc: 1.000 - ETA: 3s - loss: 4.2980e-05  - ETA: 0s - loss: 4.1878e-05 - acc: 1.000 - ETA: 0s - loss: 4.1781e-
Epoch 18/20
56000/56000 [==============================] - 20s 357us/step - loss: 3.3562e-05 - acc: 1.0000 - val_loss: 0.
0474 - val_acc: 0.9921- ETA: 0s - loss: 3.3913e-05 - acc: 1.00 - ETA: 0s - loss: 3.3773e-05 - acc:
Epoch 19/20
56000/56000 [==============================] - 20s 364us/step - loss: 2.9089e-05 - acc: 1.0000 - val_loss: 0.
0479 - val_acc: 0.9921
Epoch 20/20
56000/56000 [==============================] - 20s 363us/step - loss: 2.5812e-05 - acc: 1.0000 - val_loss: 0.
0485 - val_acc: 0.9920
14000/14000 [==============================] - 2s 134us/step
> 99.200
```

```
Train on 56000 samples, validate on 14000 samples
Epoch 1/20
56000/56000 [==============================] - 21s 370us/step - loss: 0.1361 - acc: 0.9575 - val_loss: 0.0772
- val_acc: 0.9753
Epoch 2/20
56000/56000 [==============================] - 21s 369us/step - loss: 0.0481 - acc: 0.9850 - val_loss: 0.0471
- val_acc: 0.9859
Epoch 3/20
56000/56000 [==============================] - 21s 380us/step - loss: 0.0309 - acc: 0.9905 - val_loss: 0.0423
- val_acc: 0.9869
Epoch 4/20
56000/56000 [==============================] - 21s 370us/step - loss: 0.0229 - acc: 0.9929 - val_loss: 0.0381
- val_acc: 0.9891
Epoch 5/20
56000/56000 [==============================] - 20s 353us/step - loss: 0.0175 - acc: 0.9943 - val_loss: 0.0366
- val_acc: 0.9896- ETA: 4s - loss: 0.0173 - acc: 0. - ETA: 4s - loss: 0.01 - ETA: 1s - loss:  - ETA: 0s - los
s: 0.0176 - ac
Epoch 6/20
56000/56000 [==============================] - 19s 348us/step - loss: 0.0141 - acc: 0.9957 - val_loss: 0.0428
- val_acc: 0.98870.9 - ETA: 0s - loss: 0.0142 - ac
Epoch 7/20
56000/56000 [==============================] - 20s 356us/step - loss: 0.0097 - acc: 0.9973 - val_loss: 0.0400
- val_acc: 0.9902
Epoch 8/20
56000/56000 [==============================] - 20s 361us/step - loss: 0.0074 - acc: 0.9978 - val_loss: 0.0417
- val_acc: 0.9908
Epoch 9/20
56000/56000 [==============================] - 21s 372us/step - loss: 0.0068 - acc: 0.9977 - val_loss: 0.0448
- val_acc: 0.9899
Epoch 10/20
56000/56000 [==============================] - 20s 363us/step - loss: 0.0051 - acc: 0.9984 - val_loss: 0.0414
- val_acc: 0.9912
Epoch 11/20
56000/56000 [==============================] - 19s 345us/step - loss: 0.0067 - acc: 0.9979 - val_loss: 0.0573
- val_acc: 0.9882
Epoch 12/20
56000/56000 [==============================] - 20s 356us/step - loss: 0.0045 - acc: 0.9986 - val_loss: 0.0433
- val_acc: 0.9903
Epoch 13/20
56000/56000 [==============================] - 20s 361us/step - loss: 0.0028 - acc: 0.9993 - val_loss: 0.0442
- val_acc: 0.9914
Epoch 14/20
56000/56000 [==============================] - 19s 343us/step - loss: 0.0022 - acc: 0.9995 - val_loss: 0.0471
```

```
- val_acc: 0.9904
Epoch 15/20
56000/56000 [==============================] - 19s 331us/step - loss: 0.0022 - acc: 0.9996 - val_loss: 0.0428
- val_acc: 0.9923
Epoch 16/20
56000/56000 [==============================] - 19s 332us/step - loss: 8.8951e-04 - acc: 0.9999 - val_loss: 0.
0435 - val_acc: 0.9924
Epoch 17/20
56000/56000 [==============================] - 19s 336us/step - loss: 7.0324e-04 - acc: 1.0000 - val_loss: 0.
0451 - val_acc: 0.9926
Epoch 18/20
56000/56000 [==============================] - 19s 336us/step - loss: 6.4789e-04 - acc: 1.0000 - val_loss: 0.
0462 - val_acc: 0.9925
Epoch 19/20
56000/56000 [==============================] - 18s 326us/step - loss: 6.2766e-04 - acc: 1.0000 - val_loss: 0.
0463 - val_acc: 0.9925
Epoch 20/20
56000/56000 [==============================] - 18s 327us/step - loss: 6.1460e-04 - acc: 1.0000 - val_loss: 0.
0464 - val_acc: 0.9924TA: 0s - loss: 6.3847e-04 -
14000/14000 [==============================] - 2s 116us/step
> 99.243
Train on 56000 samples, validate on 14000 samples
Epoch 1/20
56000/56000 [==============================] - 19s 339us/step - loss: 0.1359 - acc: 0.9585 - val_loss: 0.0601
- val_acc: 0.9808
Epoch 2/20
56000/56000 [==============================] - 19s 336us/step - loss: 0.0449 - acc: 0.9853 - val_loss: 0.0405
- val_acc: 0.9869
Epoch 3/20
56000/56000 [==============================] - 19s 339us/step - loss: 0.0288 - acc: 0.9910 - val_loss: 0.0368
- val_acc: 0.9889
Epoch 4/20
56000/56000 [==============================] - 18s 329us/step - loss: 0.0215 - acc: 0.9934 - val_loss: 0.0335
- val_acc: 0.9898
Epoch 5/20
56000/56000 [==============================] - 18s 328us/step - loss: 0.0156 - acc: 0.9952 - val_loss: 0.0351
- val_acc: 0.9889A: 4s - loss: 0.015 - ETA: 3s
Epoch 6/20
56000/56000 [==============================] - 19s 332us/step - loss: 0.0111 - acc: 0.9964 - val_loss: 0.0300
- val_acc: 0.9912
Epoch 7/20
56000/56000 [==============================] - 18s 326us/step - loss: 0.0085 - acc: 0.9973 - val_loss: 0.0300
- val_acc: 0.9926
```

```
Epoch 8/20
56000/56000 [==============================] - 18s 324us/step - loss: 0.0058 - acc: 0.9982 - val_loss: 0.0399
- val_acc: 0.9895loss: 0.0044 - acc: 0.9 - ETA:  - ETA: 0s - loss: 0.0054 - acc:
Epoch 9/20
56000/56000 [==============================] - 18s 326us/step - loss: 0.0060 - acc: 0.9980 - val_loss: 0.0384
- val_acc: 0.9900
Epoch 10/20
56000/56000 [==============================] - 18s 330us/step - loss: 0.0042 - acc: 0.9986 - val_loss: 0.0409
- val_acc: 0.9905
Epoch 11/20
56000/56000 [==============================] - 18s 328us/step - loss: 0.0047 - acc: 0.9986 - val_loss: 0.0394
- val_acc: 0.9914
Epoch 12/20
56000/56000 [==============================] - 18s 321us/step - loss: 0.0027 - acc: 0.9992 - val_loss: 0.0367
- val_acc: 0.9909
Epoch 13/20
56000/56000 [==============================] - 18s 323us/step - loss: 0.0011 - acc: 0.9996 - val_loss: 0.0347
- val_acc: 0.9927
Epoch 14/20
56000/56000 [==============================] - 18s 324us/step - loss: 3.4325e-04 - acc: 1.0000 - val_loss: 0.
0360 - val_acc: 0.9921oss: 3.2830e-04 - acc: 1. - ETA: 1s - loss: 3.2468e-04  - ETA: 1s - loss: 3.4
Epoch 15/20
56000/56000 [==============================] - 18s 326us/step - loss: 1.2294e-04 - acc: 1.0000 - val_loss: 0.
0356 - val_acc: 0.9926
Epoch 16/20
56000/56000 [==============================] - 18s 324us/step - loss: 6.3113e-05 - acc: 1.0000 - val_loss: 0.
0361 - val_acc: 0.9927
Epoch 17/20
56000/56000 [==============================] - 18s 326us/step - loss: 4.8737e-05 - acc: 1.0000 - val_loss: 0.
0369 - val_acc: 0.9927
Epoch 18/20
56000/56000 [==============================] - 18s 328us/step - loss: 4.1045e-05 - acc: 1.0000 - val_loss: 0.
0374 - val_acc: 0.9928
Epoch 19/20
56000/56000 [==============================] - 19s 333us/step - loss: 3.5967e-05 - acc: 1.0000 - val_loss: 0.
0377 - val_acc: 0.9927
Epoch 20/20
56000/56000 [==============================] - 19s 338us/step - loss: 3.1755e-05 - acc: 1.0000 - val_loss: 0.
0380 - val_acc: 0.9927
14000/14000 [==============================] - 2s 122us/step
> 99.271
Train on 56000 samples, validate on 14000 samples
Epoch 1/20
```

```
56000/56000 [==============================] - 19s 343us/step - loss: 0.1214 - acc: 0.9622 - val_loss: 0.0574
- val_acc: 0.9824
Epoch 2/20
56000/56000 [==============================] - 19s 342us/step - loss: 0.0409 - acc: 0.9870 - val_loss: 0.0599
- val_acc: 0.9829
Epoch 3/20
56000/56000 [==============================] - 20s 350us/step - loss: 0.0261 - acc: 0.9913 - val_loss: 0.0400
- val_acc: 0.9875 - loss: 0. - ETA: 3s -
Epoch 4/20
56000/56000 [==============================] - 19s 334us/step - loss: 0.0196 - acc: 0.9936 - val_loss: 0.0397
- val_acc: 0.9891 loss: 0.0187 - ac  - ETA: 2s - loss: 0.0200 - acc - ETA: 1s -
Epoch 5/20
56000/56000 [==============================] - 19s 333us/step - loss: 0.0137 - acc: 0.9958 - val_loss: 0.0464
- val_acc: 0.9874A: 7s - loss: 0.0 - E
Epoch 6/20
56000/56000 [==============================] - 19s 338us/step - loss: 0.0113 - acc: 0.9965 - val_loss: 0.0485
- val_acc: 0.9869
Epoch 7/20
56000/56000 [==============================] - 19s 344us/step - loss: 0.0077 - acc: 0.9976 - val_loss: 0.0393
- val_acc: 0.9890
Epoch 8/20
56000/56000 [==============================] - 19s 341us/step - loss: 0.0065 - acc: 0.9980 - val_loss: 0.0476
- val_acc: 0.9890
Epoch 9/20
56000/56000 [==============================] - 18s 324us/step - loss: 0.0059 - acc: 0.9982 - val_loss: 0.0412
- val_acc: 0.9905 - loss: 0
Epoch 10/20
56000/56000 [==============================] - 18s 326us/step - loss: 0.0034 - acc: 0.9989 - val_loss: 0.0438
- val_acc: 0.9903
Epoch 11/20
56000/56000 [==============================] - 19s 331us/step - loss: 0.0038 - acc: 0.9989 - val_loss: 0.0434
- val_acc: 0.9899
Epoch 12/20
56000/56000 [==============================] - 19s 337us/step - loss: 0.0027 - acc: 0.9993 - val_loss: 0.0443
- val_acc: 0.99060028 - acc: 0.
Epoch 13/20
56000/56000 [==============================] - 19s 342us/step - loss: 0.0011 - acc: 0.9998 - val_loss: 0.0465
- val_acc: 0.9906 - ETA: 5s - loss: 9.0621e-0 - ETA: 4s - loss: 9.3983e-04 - a - ETA: 1s - loss: 8.4815 - ET
A: 0s - loss: 0.0011 -
Epoch 14/20
56000/56000 [==============================] - 19s 347us/step - loss: 6.3919e-04 - acc: 0.9999 - val_loss: 0.
0468 - val_acc: 0.9909A: 8s - loss: 9.1085e-0 - ETA: 7s - loss: 9 - ETA: 6s - los - ETA: 5s - ETA: 3s - loss:
7.2650e
```

```
Epoch 15/20
56000/56000 [==============================] - 18s 329us/step - loss: 4.4072e-04 - acc: 1.0000 - val_loss: 0.
0467 - val_acc: 0.9914
Epoch 16/20
56000/56000 [==============================] - 18s 326us/step - loss: 3.7637e-04 - acc: 1.0000 - val_loss: 0.
0478 - val_acc: 0.9907
Epoch 17/20
56000/56000 [==============================] - 18s 330us/step - loss: 3.8011e-04 - acc: 1.0000 - val_loss: 0.
0487 - val_acc: 0.9911s - loss: 3.8033e-04 - acc: 1.000
Epoch 18/20
56000/56000 [==============================] - 19s 334us/step - loss: 3.3370e-04 - acc: 1.0000 - val_loss: 0.
0492 - val_acc: 0.9908
Epoch 19/20
56000/56000 [==============================] - 19s 341us/step - loss: 3.2529e-04 - acc: 1.0000 - val_loss: 0.
0496 - val_acc: 0.9911e-04 - acc - ETA: 0s - loss: 3.2933e-04 - acc: 1
Epoch 20/20
56000/56000 [==============================] - 20s 357us/step - loss: 3.2076e-04 - acc: 1.0000 - val_loss: 0.
0503 - val_acc: 0.9909
14000/14000 [==============================] - 2s 129us/step
> 99.093
```

In [17]:
```
print(final[0])
tests = list(range(1,len(final[0])+1))

fig = px.scatter(x = tests, y = final[0], labels=dict(x="K-folds test number:", y="Accuracy on validation dat
a"))

fig.update_layout(xaxis={'tickformat': ',d'})
fig.update_layout(yaxis_range=[0,1])
```

```
[0.9926428571428572, 0.992, 0.9924285714285714, 0.9927142857142857, 0.9909285714285714]
```

Results of cross validation

After running 5 folds and plotting the graph above we see that the data is not overfitted to the specific data-split and we can assume it is generilzed for unseen data. We now choose the best performing model and test it on unseen data.

In [18]:
```
best_model = final[1][final[0].index(max(final[0]))]

pred = best_model.predict_classes(X_test)
y_test_digits = [np.argmax(dig) for dig in y_test]
print(classification_report(y_test_digits, pred))

print("keras Sequential Test data accuracy: {:3.2f}%".format(accuracy_score(y_test_digits, pred)*100))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       685
           1       1.00      1.00      1.00       778
           2       1.00      1.00      1.00       671
           3       1.00      1.00      1.00       690
           4       1.00      1.00      1.00       733
           5       1.00      1.00      1.00       644
           6       1.00      1.00      1.00       729
           7       1.00      1.00      1.00       694
           8       1.00      1.00      1.00       670
           9       1.00      1.00      1.00       706

    accuracy                           1.00      7000
   macro avg       1.00      1.00      1.00      7000
weighted avg       1.00      1.00      1.00      7000

keras Sequential Test data accuracy: 99.89%
```

What is your final classifier and how does it work.

We choose the Sequential neural net model as our final classifier with accuracy of 99.89% on test data. The model details are described above.

How well it is expected to perform in production (on unseen data).Justify your estimate

We expect it to perform very accurately by classifying almost all (99%) of unsees digits.

Measures taken to avoid overfitting

We tested the final model with folding data dataset to ensure that the split does not affect model performance. We also clearly splitted the set into train, validation and test which were used for each their part of assessing, improving and testing the models.

Given more resources (time or computing resources), how would you improve your solution

More Epochs and K-folds if i had a stronger GPU. Also I had to use older versions of CUDA, CUDNN, Tenserflow and Keras to work with my GPU. Current versions of the library may be better optimized and produce better results.

Summary

Through this project, we have tested 3 different models for classifying handwritten numbers. This is a task that has many use cases, and this project was intended for a post office that wants to automate its tasks. Based on our results, we definitely see that this task is suitable for machine learning classification. Our best model guessed correctly in almost 100% of cases, and can classify thousands of numbers in just a few seconds. Such types of models are already used to automate repetitive tasks, and as we have proven, machine learning is a good tool for this.