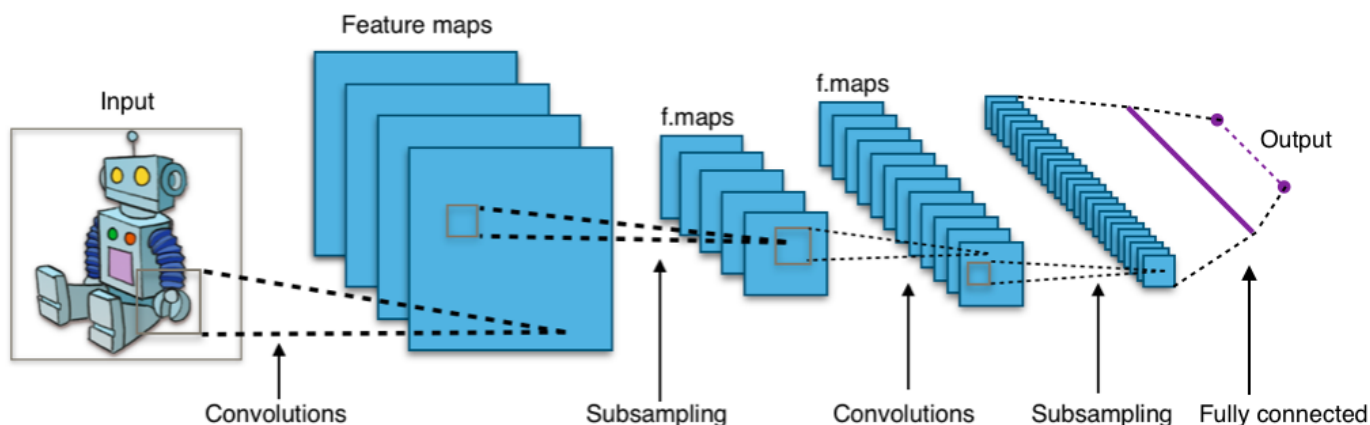


## ▼ Lab 7 - Deep Learning 2

The goal of this week's lab is to learn to use a widely-used neural network modules: convolutional neural networks (CNNs). We can use them to learn features from images and even text.



Images and text are common data modalities we encounter in classification tasks. While we can directly apply the linear or multi-layer modules we learned in the past few weeks to those modalities, there are neural network modules specifically designed for processing them, namely CNNs and RNNs.

This week we will walk through the basics of CNNs and RNNs.

- **Review:** Training and Multi-Layer Models (NNs)
- **Unit A:** Image Processing and Convolutions
- **Unit B:** Convolution Neural Networks (CNNs)

## ▼ Review

Last time we took a look at what's happening inside training when we call `model.fit`. We did this by implementing `model.fit` ourselves.

```
# For Tables
import pandas as pd
# For Visualization
import altair as alt
# For Scikit-Learn
```

```
import sklearn
from keras.wrappers.scikit_learn import KerasClassifier
# For Neural Networks
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense
```

We will also turn off warnings.

```
import warnings
warnings.filterwarnings('ignore')
```

As we have seen in past weeks we load our structured data in Pandas format.

```
df = pd.read_csv("https://srush.github.io/BT-AI/notebooks/circle.csv")
all_df = pd.read_csv("https://srush.github.io/BT-AI/notebooks/all_points.csv")
```

Next, we need to define a function that creates our model. This will determine the range or different feature shapes the model can learn.

### [TensorFlow Playground](#)

Depending on the complexity of the data we may select a model that is linear or one with multiple layers.

Here is what a linear model looks like.

```
def create_linear_model(learning_rate=1.0):
    # Makes it the same for everyone in class
    tf.random.set_seed(2)

    # Create model
    model = Sequential()
    model.add(Dense(1, activation="sigmoid"))

    # Compile model
    optimizer = tf.keras.optimizers.SGD(
        learning_rate=learning_rate
    )
    model.compile(loss="binary_crossentropy",
                  optimizer=optimizer,
                  metrics=["accuracy"])
```

```
return model
```

Here is what a more complex multi-layer model looks like.

```
def create_model(learning_rate=0.05):
    tf.random.set_seed(2)
    # create model
    model = Sequential()
    model.add(Dense(8, activation="relu"))
    model.add(Dense(8, activation="relu"))
    model.add(Dense(1, activation="sigmoid"))
    # Compile model
    optimizer = tf.keras.optimizers.SGD(
        learning_rate=learning_rate
    )

    model.compile(loss="binary_crossentropy",
                  optimizer=optimizer,
                  metrics=["accuracy"])
    return model
```

Generally, we will use "ReLU" for the inner layers and "sigmoid" for the final layer. The reasons for this are beyond the class, and mostly have to do with computational simplicity and standard practice.

Once we have described the shape of our model we can turn it into a classifier and train it on data.

```
model = KerasClassifier(build_fn=create_model,
                        epochs=20,
                        batch_size=20,
                        verbose=0)
```

The neural network is used just like the classifiers from Week 4 and 5. The only difference is that we got to design its internal shape.

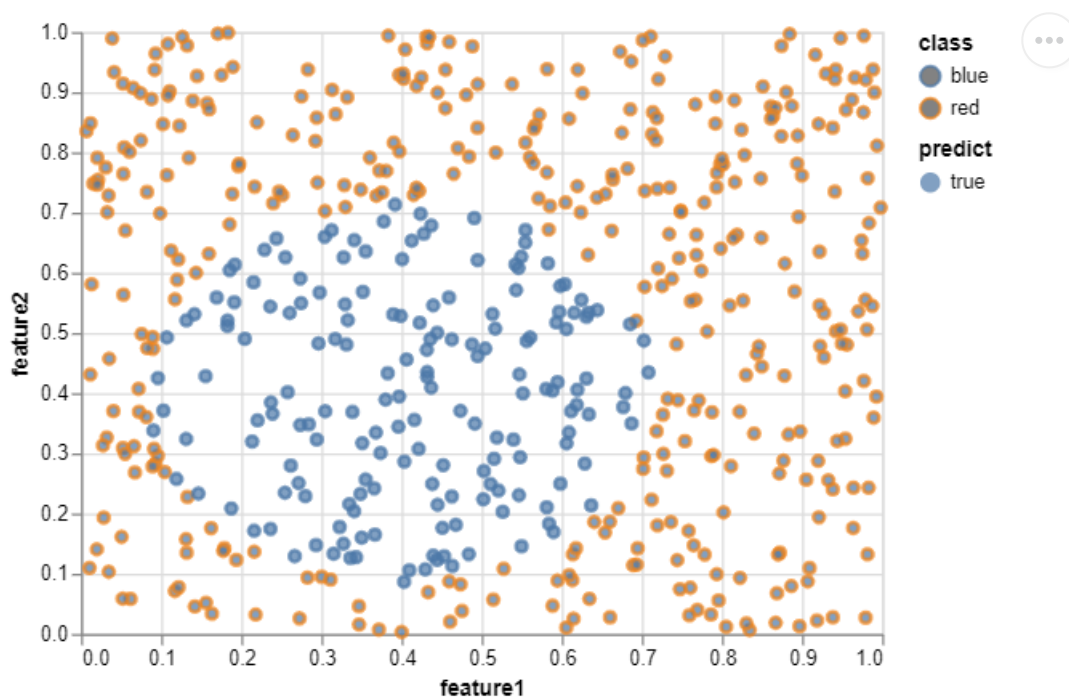
```
model.fit(x=df[["feature1", "feature2"]],
          y=(df["class"] == "red"))
df["predict"] = model.predict(df[["feature1", "feature2"]])
```

The output of the `model.fit` command tells us a lot of information about how the approach is doing.

In particular if it is working the `loss` should go down and the `accuracy` should go up. This implies that the model is learning to fit to the data that we provided it.

We can view how the model determines the separation of the data.

```
chart = (alt.Chart(df)
    .mark_point()
    .encode(
        x="feature1",
        y="feature2",
        color="class",
        fill="predict",
    ))
chart
```



## ▼ Review Exercise

Change the model above to have three inner layers with the first having size 10, the second size 5, and the third having size 5. How well does this do on our problem?

# FILLME

## ▼ Unit A

### ▼ Image Classification

Today's focus will be the problem of image classification. The goal is to take an image and predict the image class. So instead of predicting whether a point is red or blue we will now be predicting whether an image is a cat or a dog, or a house or a plane.

We are going to start with a famous simple image classification tasks known as MNist. This dataset consists of pictures of hand-written numbers. The goal is to look at the handwriting and determine what the number is.

Let's start with an image classification task. We will be using the [MNIST dataset](https://srush.github.io/BT-AI/notebooks/mnist_train.csv.gz), where the goal is to recognize handwritten digits.

```
df_train = pd.read_csv('https://srush.github.io/BT-AI/notebooks/mnist_train.csv.gz', compress
df_test = pd.read_csv('https://srush.github.io/BT-AI/notebooks/mnist_test.csv.gz', compressio
df_train[:100]
```

	class	1x1	1x2	1x3	1x4	1x5	1x6	1x7	1x8	1x9	1x10	1x11	1x12	1x13	1x14	1x
<b>0</b>	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
<b>1</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
<b>2</b>	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
<b>3</b>	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
<b>4</b>	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
<b>95</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
<b>96</b>	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
<b>97</b>	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
<b>98</b>	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
<b>99</b>	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

100 rows × 785 columns

This data is in the same format that we have been using so far.

The column `class` stores the class of each image, which is a number between 0 and 9.

```
df_train[:100]["class"].unique()

array([5, 0, 4, 1, 9, 2, 3, 6, 7, 8])
```

The rest of columns store the features. However there are many more features than before!

In particular the images are 28x28 pixels which means we have 784 features. To make later processing easier, we store the names of pixel value columns in a list `features`.

```
features = []
for i in range(1, 29):
    for j in range(1, 29):
        features.append(str(i) + "x" + str(j))
len(features)

784
```

These features are the intensity at each pixel : for instance, the column "3x4" stores the pixel value at the 3rd row and the 4th column. Since the size of each image is 28x28, there are 28 rows and 28 columns.

We can use pandas apply to graph these values for one image.

Convert feature to x, y, and value.

```
def position(row):
    y, x = row["index"].split("x")
    return {"x":int(x),
            "y":int(y),
            "val":row["val"]}
```

Draw a heat map showing the image.

```
def draw_image(i, shuffle=False):
    t = df_train[i:i+1].T.reset_index().rename(columns={i: "val"})
    out = t.loc[t["index"] != "class"].apply(position, axis=1, result_type="expand")

    label = df_train.loc[i]["class"]
    title = "Image of a " + str(label)
    if shuffle:
```

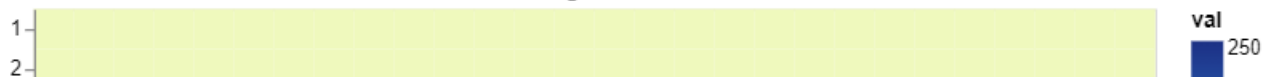
```
def shuffle():
    out["val"] = sklearn.utils.shuffle(out["val"], random_state=1234).reset_index()["val"]
    title = "Shuffled Image of a " + str(label)

    return (alt.Chart(out)
            .mark_rect()
            .properties(title=title)
            .encode(
                x="x:Q",
                y="y:Q",
                fill="val:Q",
                color="val:Q",
                tooltip=("x", "y", "val")
            ))
```

Here are some example images.

```
im = draw_image(0)
im
```

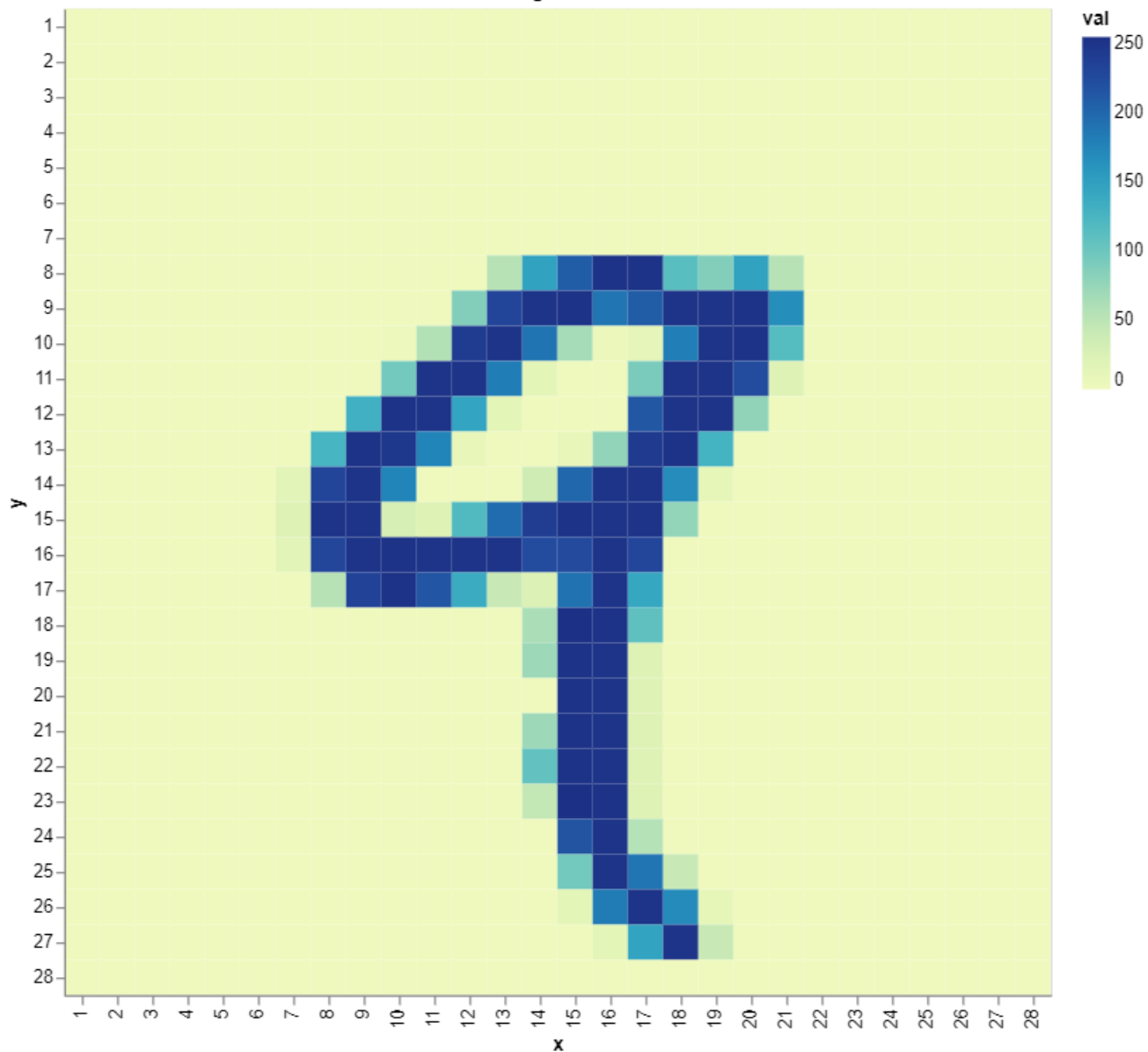
Image of a 5



```
im = draw_image(4)
```

```
im
```

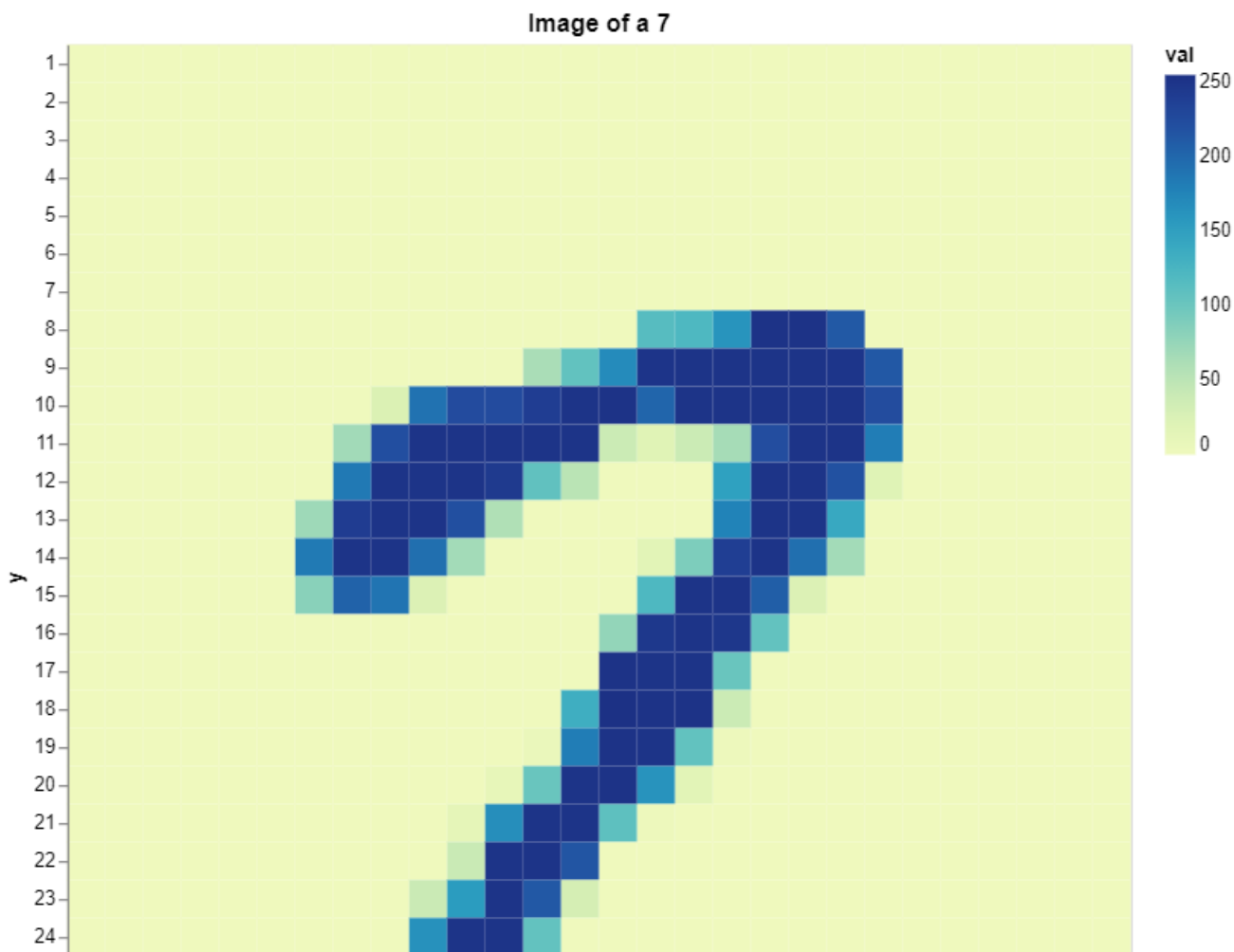
Image of a 9



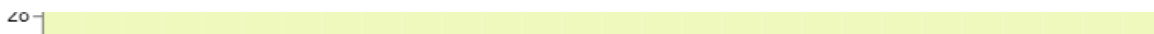
```
im = draw_image(15)
```

```
im
```





How can we solve this task? The challenge is that there are many different aspects that can make a digit look unique.



 **Student question: What are some features that you use to tell apart digits?**

We can use the NN classifier we learned last week, with a few modifications to change from binary classification to multi-class (10-way in this case) classification.

First, the final layer needs to output 10 different values. Also we need to switch `sigmoid` to a `softmax`. Lastly, we need to change the loss function to `sparse_categorical_crossentropy`.

The practical change is quite small, it should look very similar to what we have seen already

```
def create_model():
    # Makes it the same for everyone in class
    tf.random.set_seed(2)

    # Create model
    model = Sequential()
```

```

model.add(Dense(64, activation="relu"))
model.add(Dense(10, activation="softmax"))

# Compile model
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="adam",
              metrics=["accuracy"])

return model

```

While these terms are a bit technical, the main thing to know is that they team up to change the loss function from last week to score 10 different values instead of 2.

```

# Create model
model = KerasClassifier(build_fn=create_model,
                        epochs=2,
                        batch_size=20,
                        verbose=False)

# Fit model
model.fit(x=df_train[features].astype(float),
        y=df_train["class"])

<keras.callbacks.History at 0x7ff2b056a950>

```

Now that it is fit we can print a summary

```
print (model.model.summary())
```

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(20, 64)	50240
dense_4 (Dense)	(20, 10)	650
Total params: 50,890		
Trainable params: 50,890		
Non-trainable params: 0		
None		

And predict on test set

```

df_test["predict"] = model.predict(df_test[features])
correct = (df_test["predict"] == df_test["class"])
accuracy = correct.sum() / correct.size
print ("accuracy: ", accuracy)

```

accuracy: 0.9033

This simple MLP classifier was able to get 90% accuracy!

  **Student question: what is the size of the input to the model??**

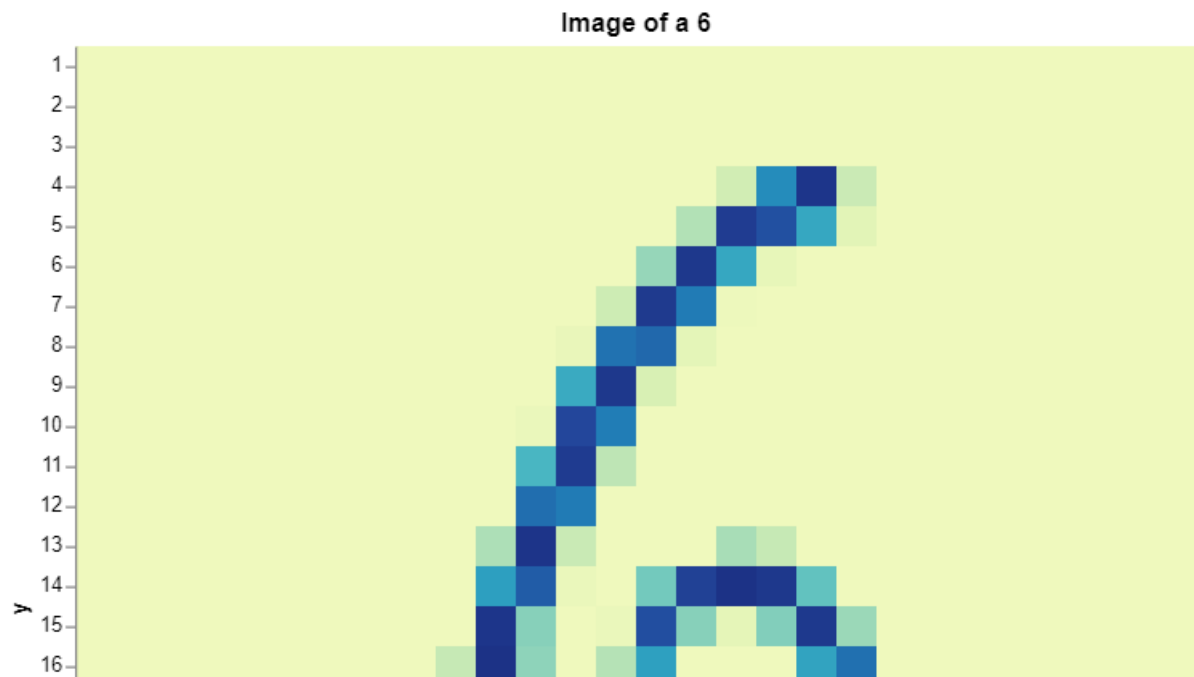
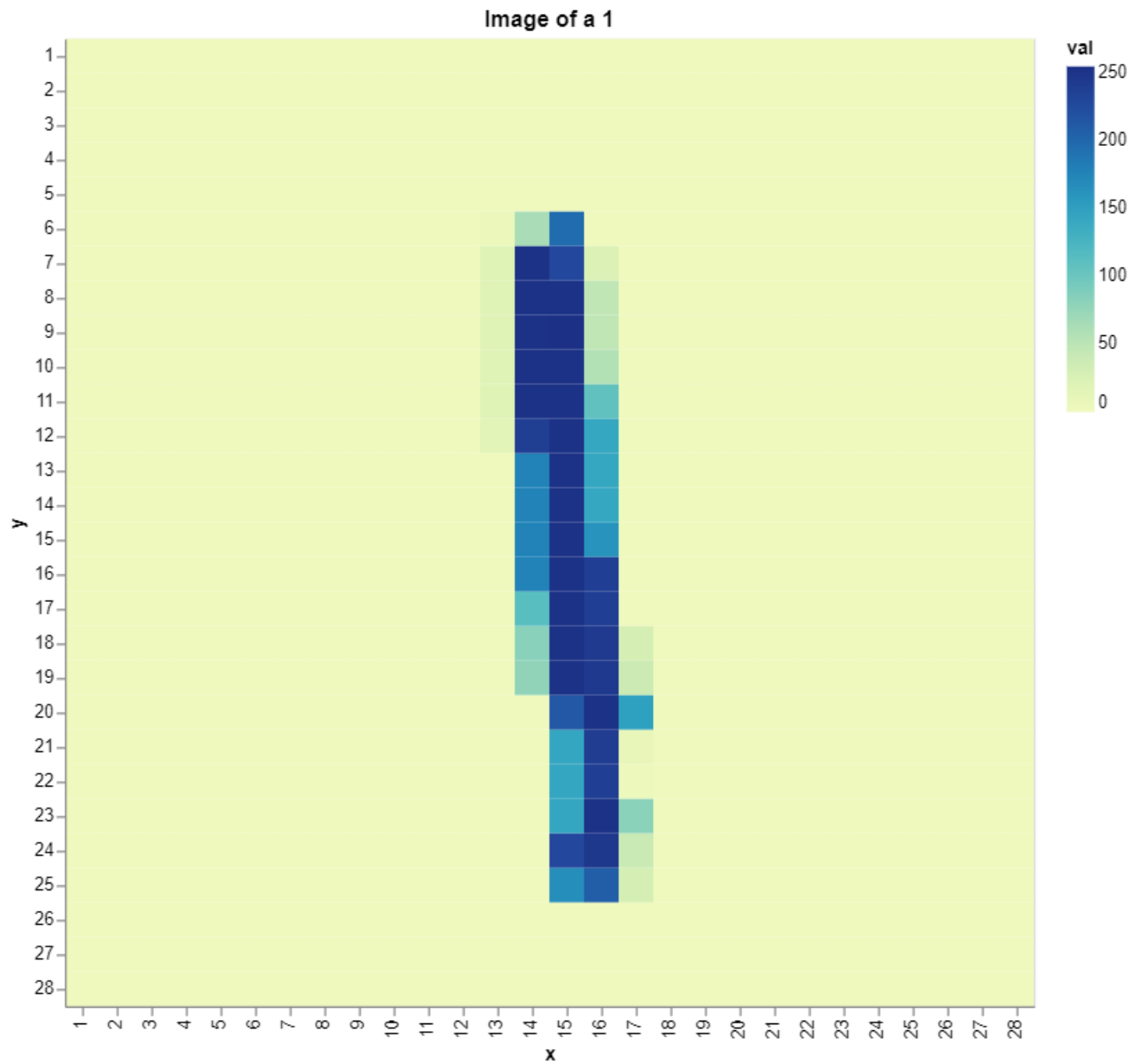
#     FILLME  
pass

It is hard to visualize the behavior of the classifier under such a high dimensionality. Instead we can look at some examples that the classifier gets wrong.

```
wrong = (df_test["predict"] != df_test["class"])
examples = df_test.loc[wrong]
num = 0
charts = alt.vconcat()
for idx, example in examples.iterrows():
    label = example["class"]
    predicted_label = example["predict"]
    charts &= draw_image(idx)
    num += 1
    if num > 10:
        break
```

charts





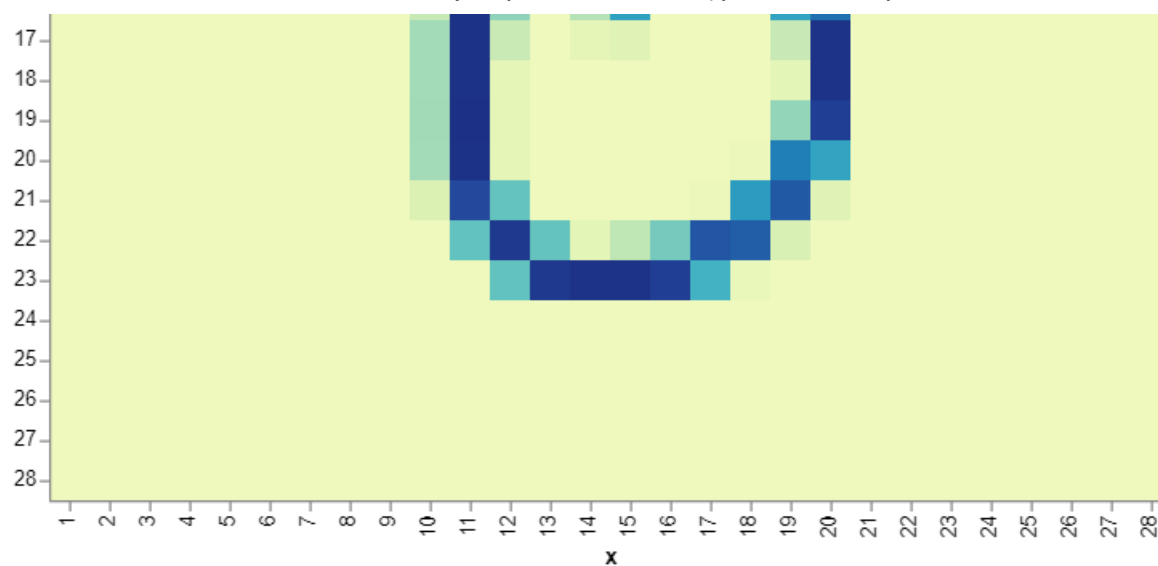


Image of a 9

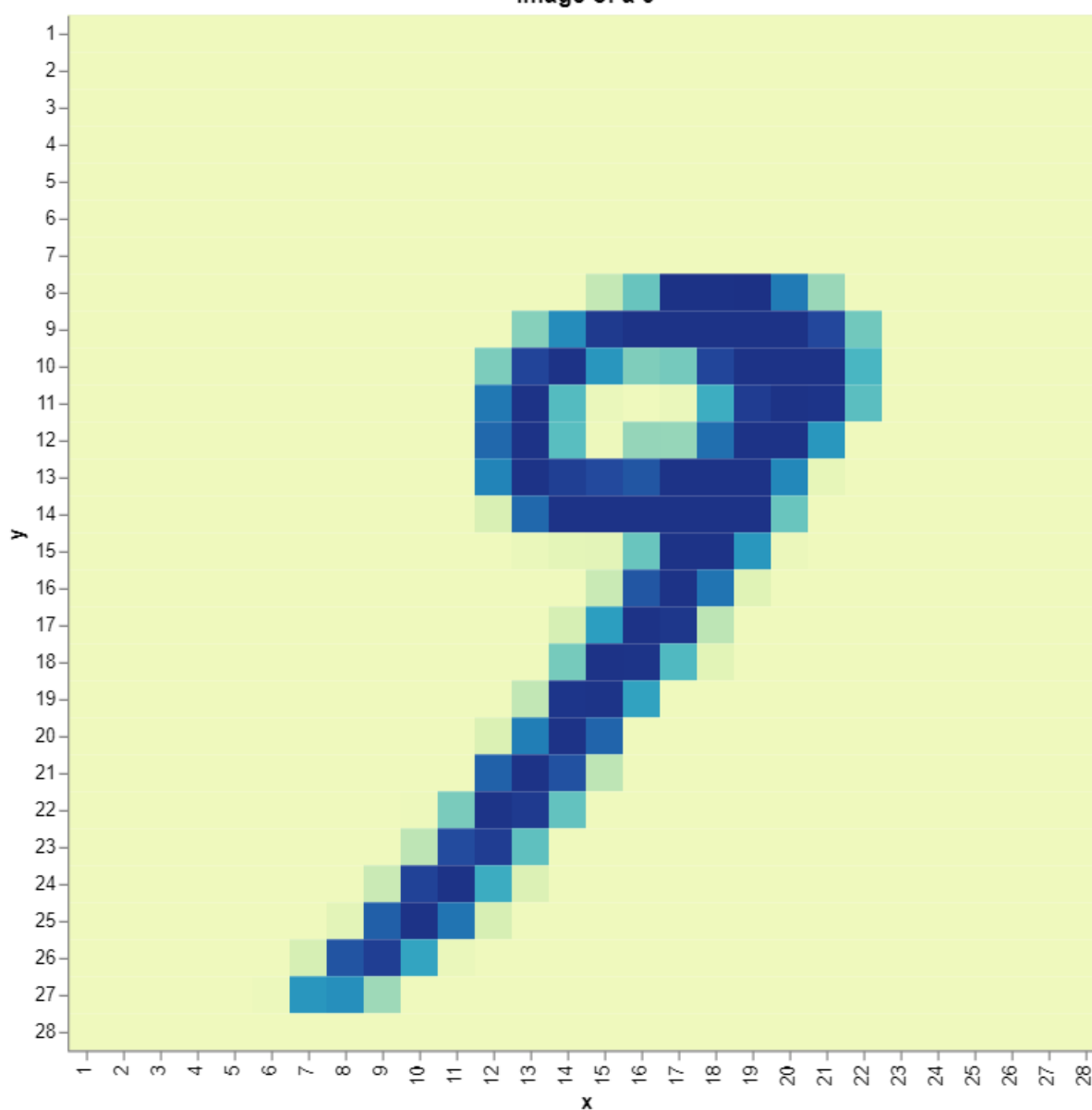
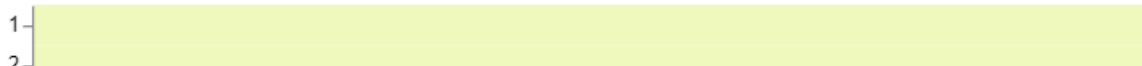


Image of a 7



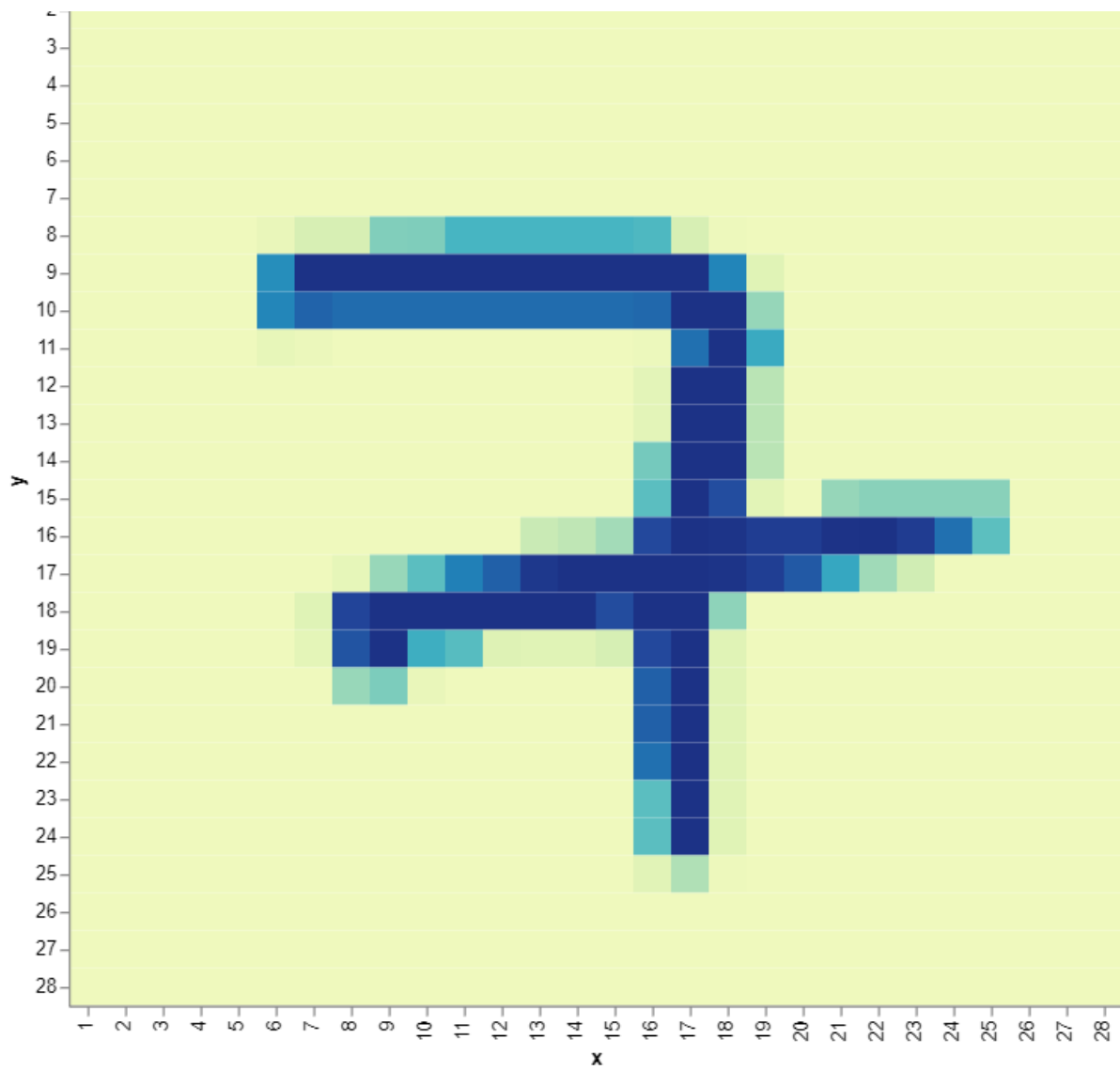
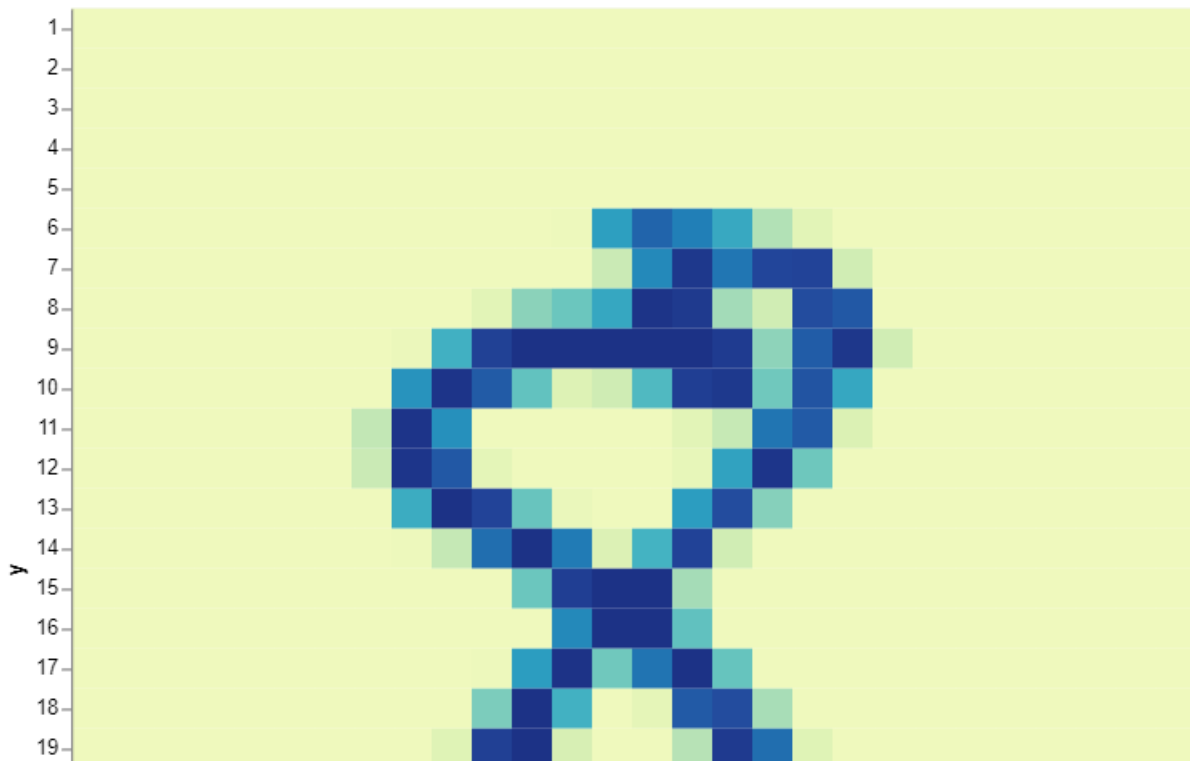


Image of a 8



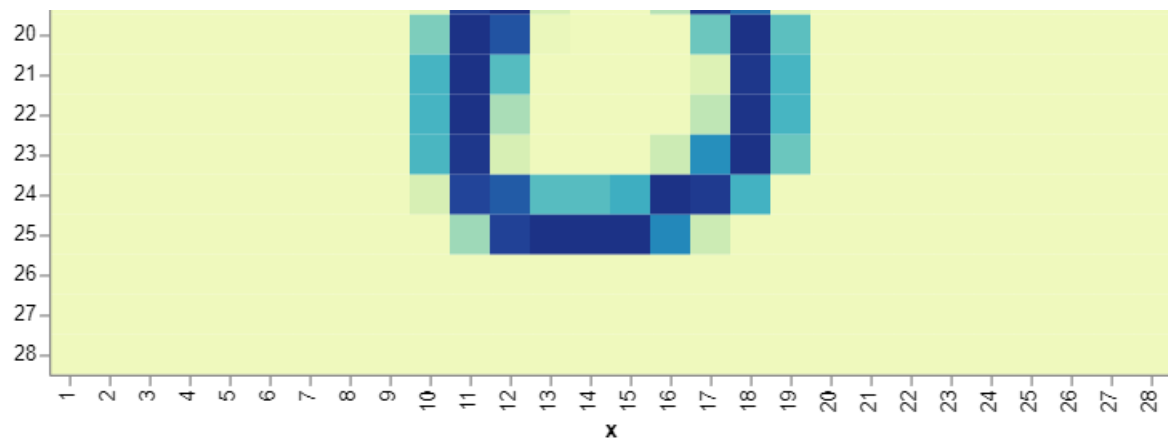


Image of a 7

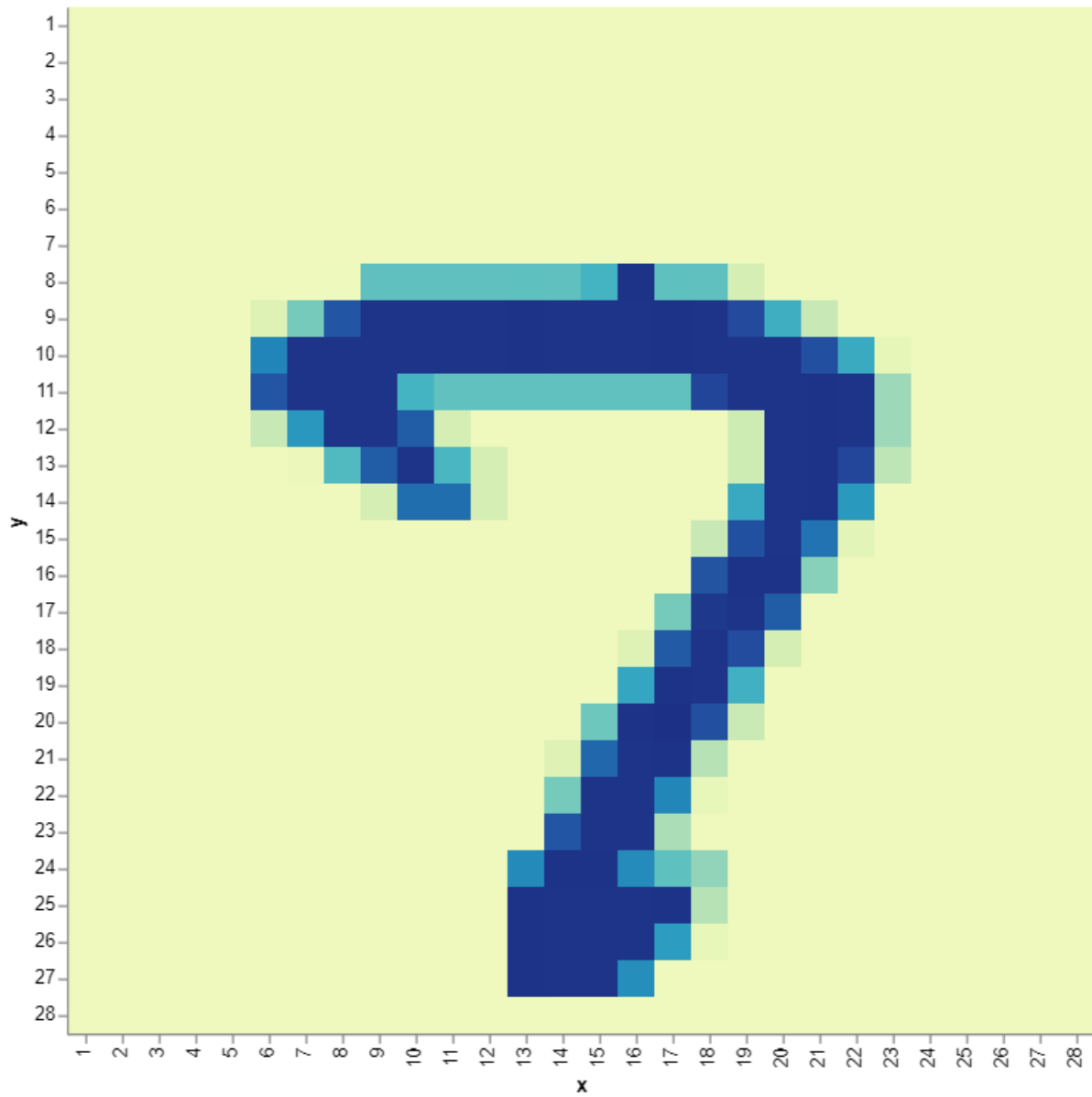
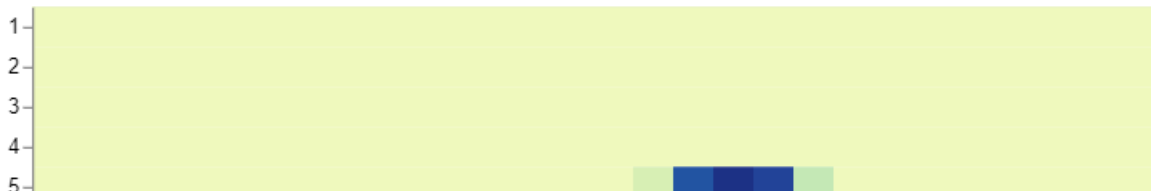


Image of a 6



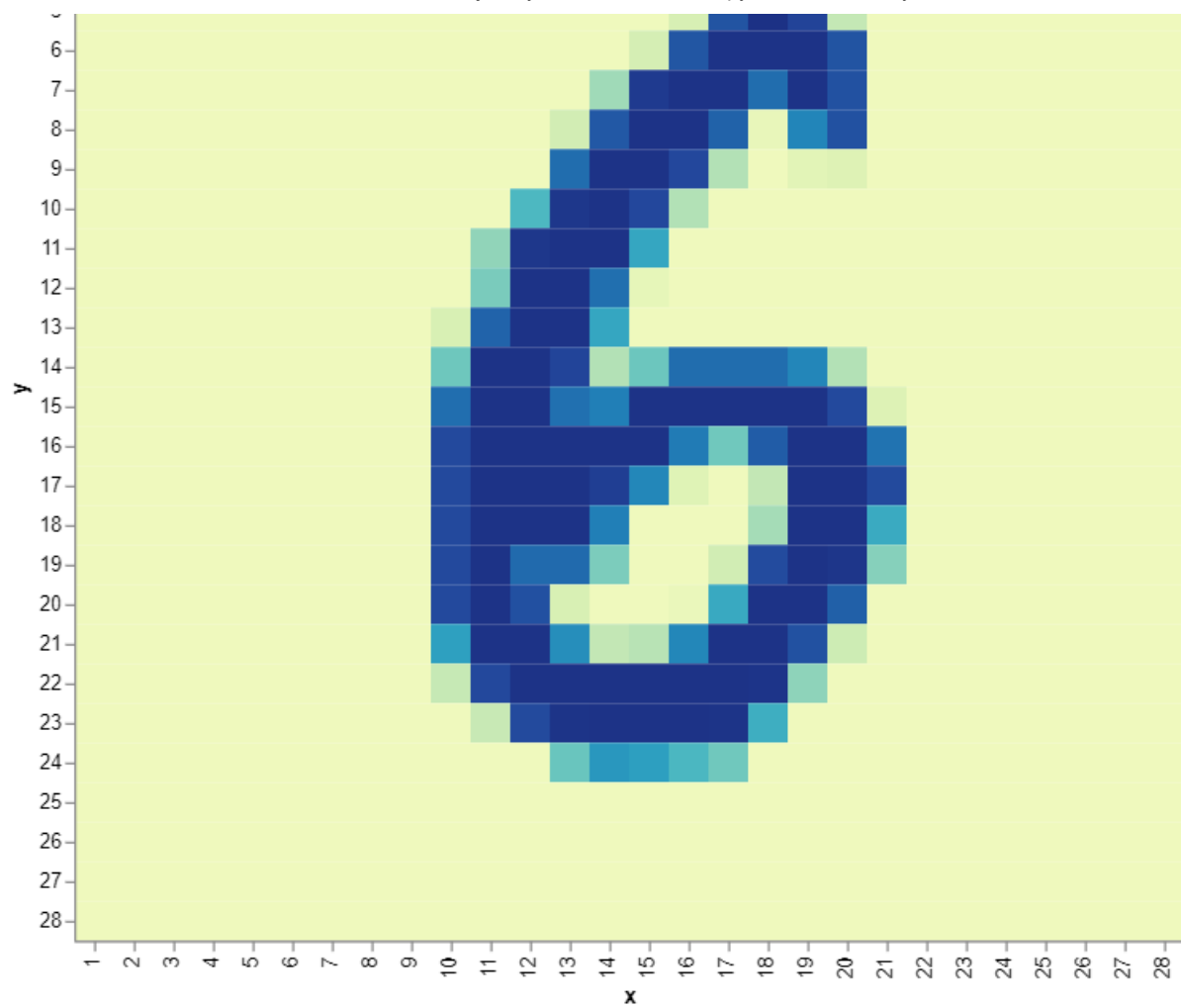
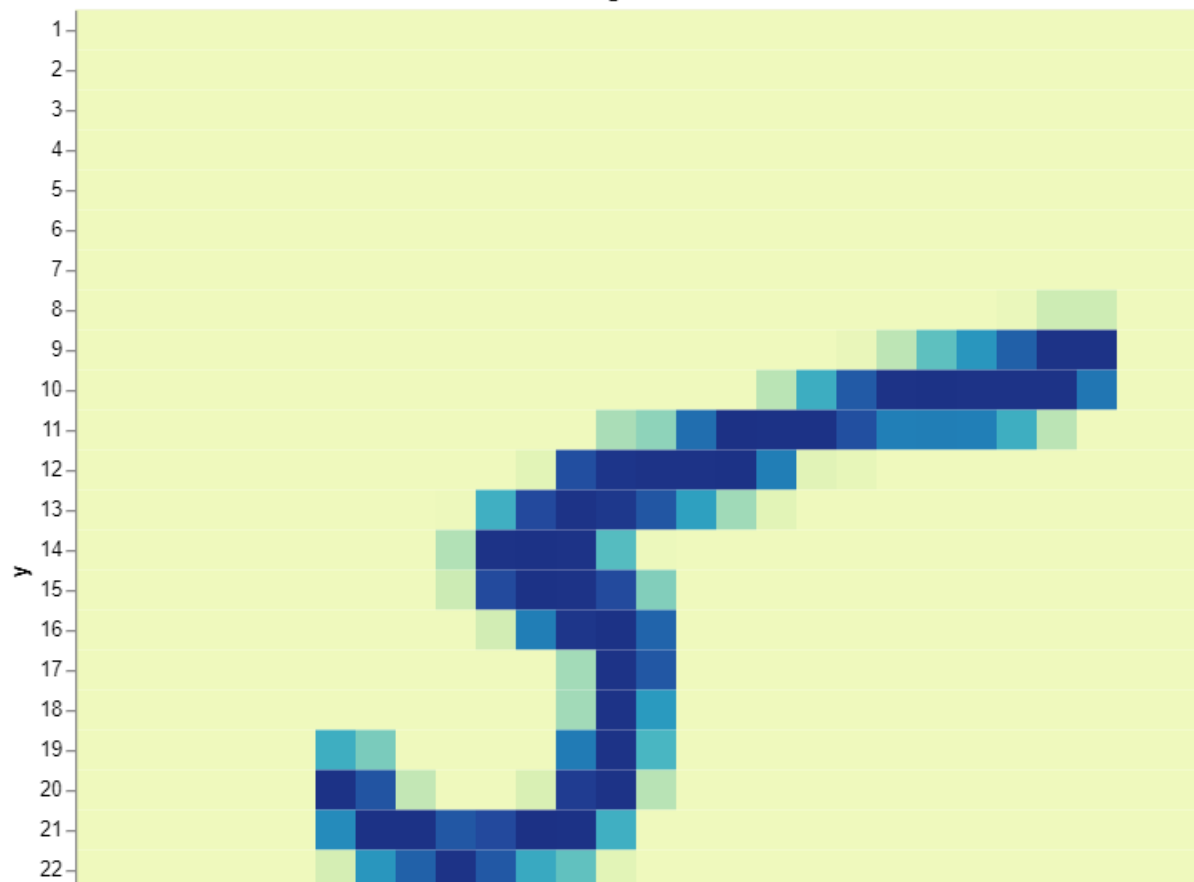


Image of a 5





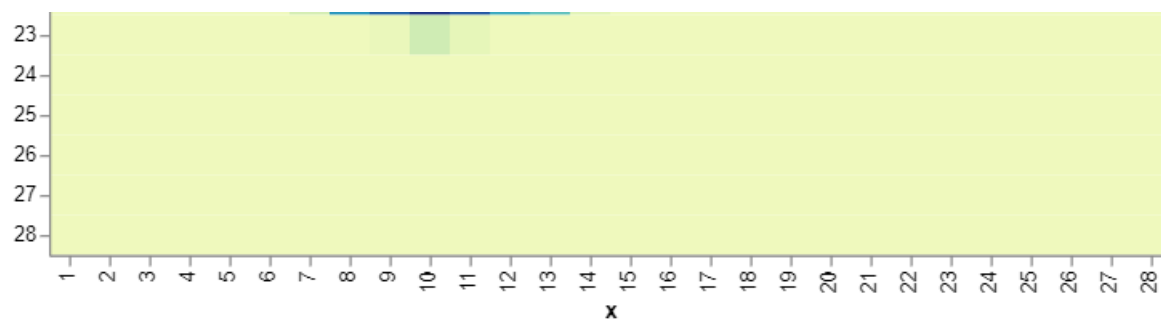


Image of a 6

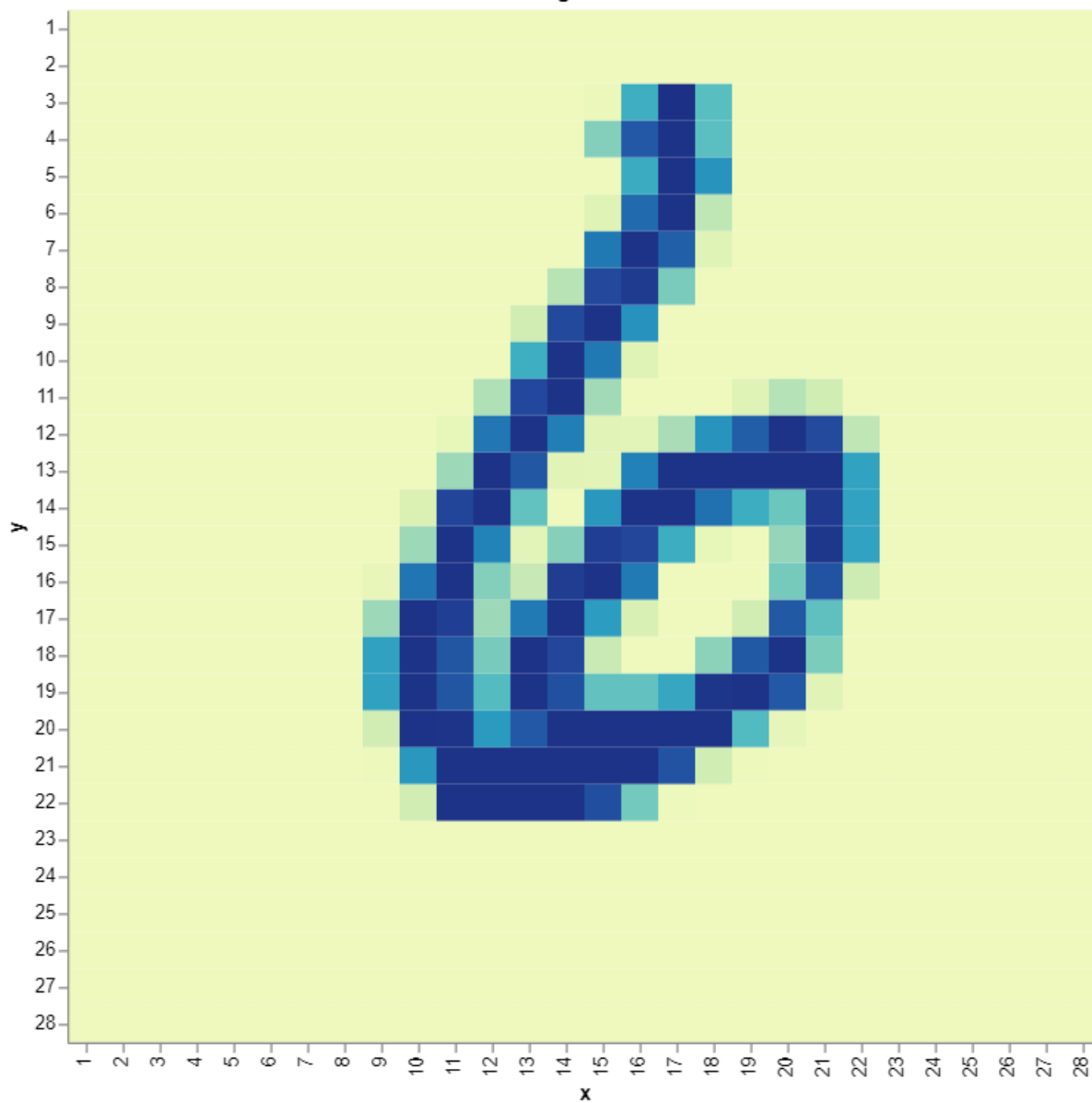
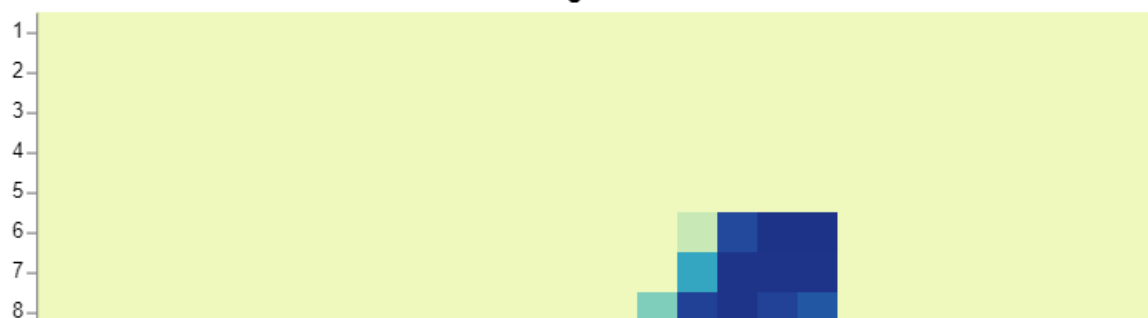


Image of a 1



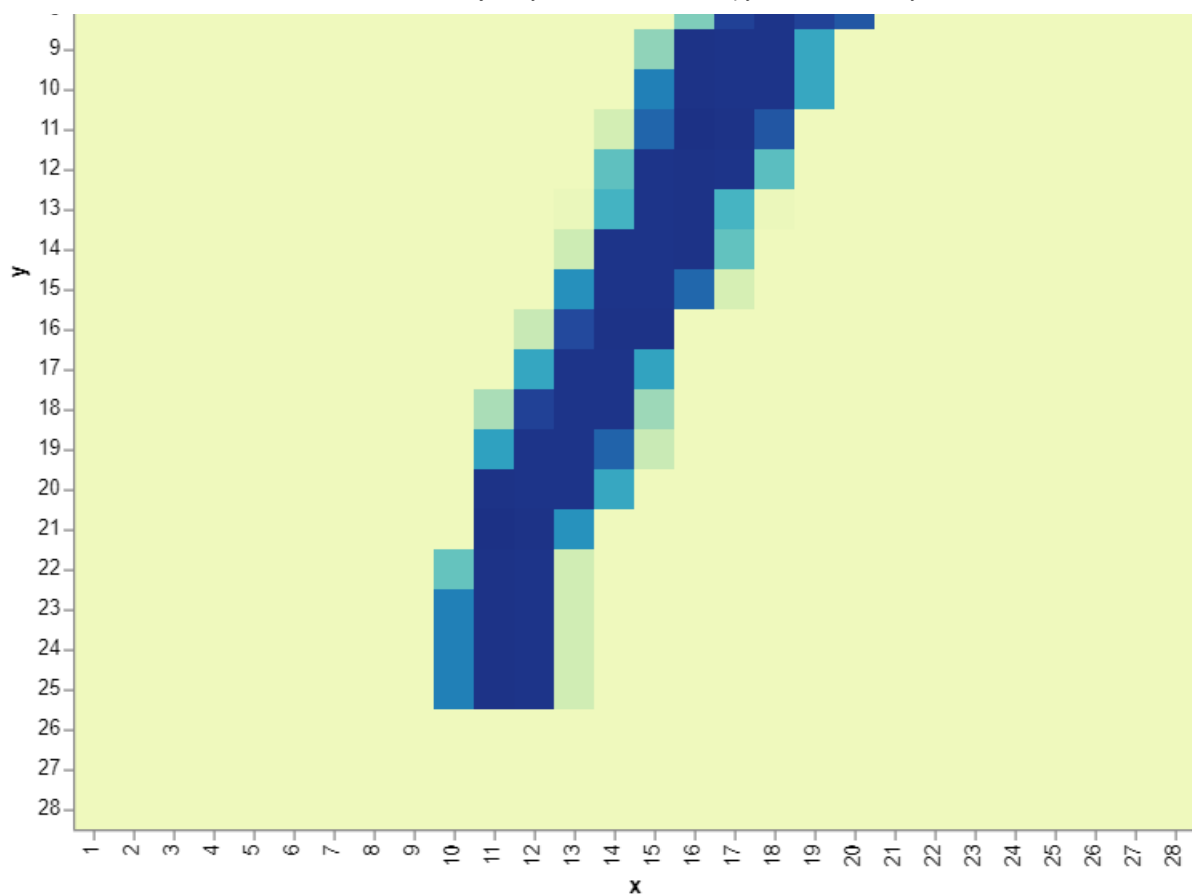
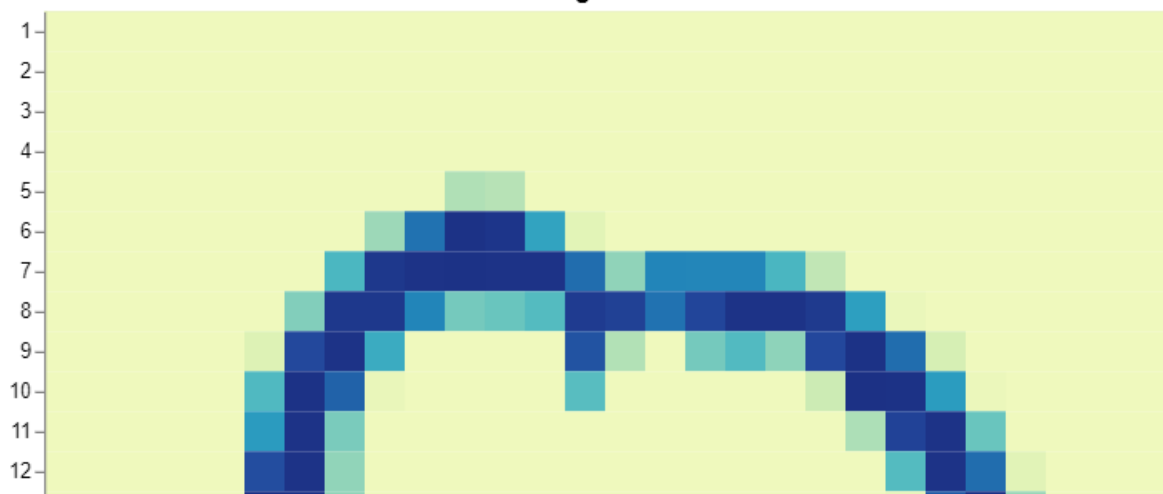


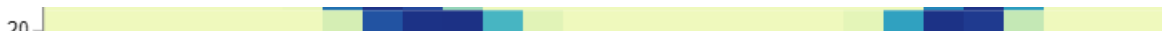
Image of a 0



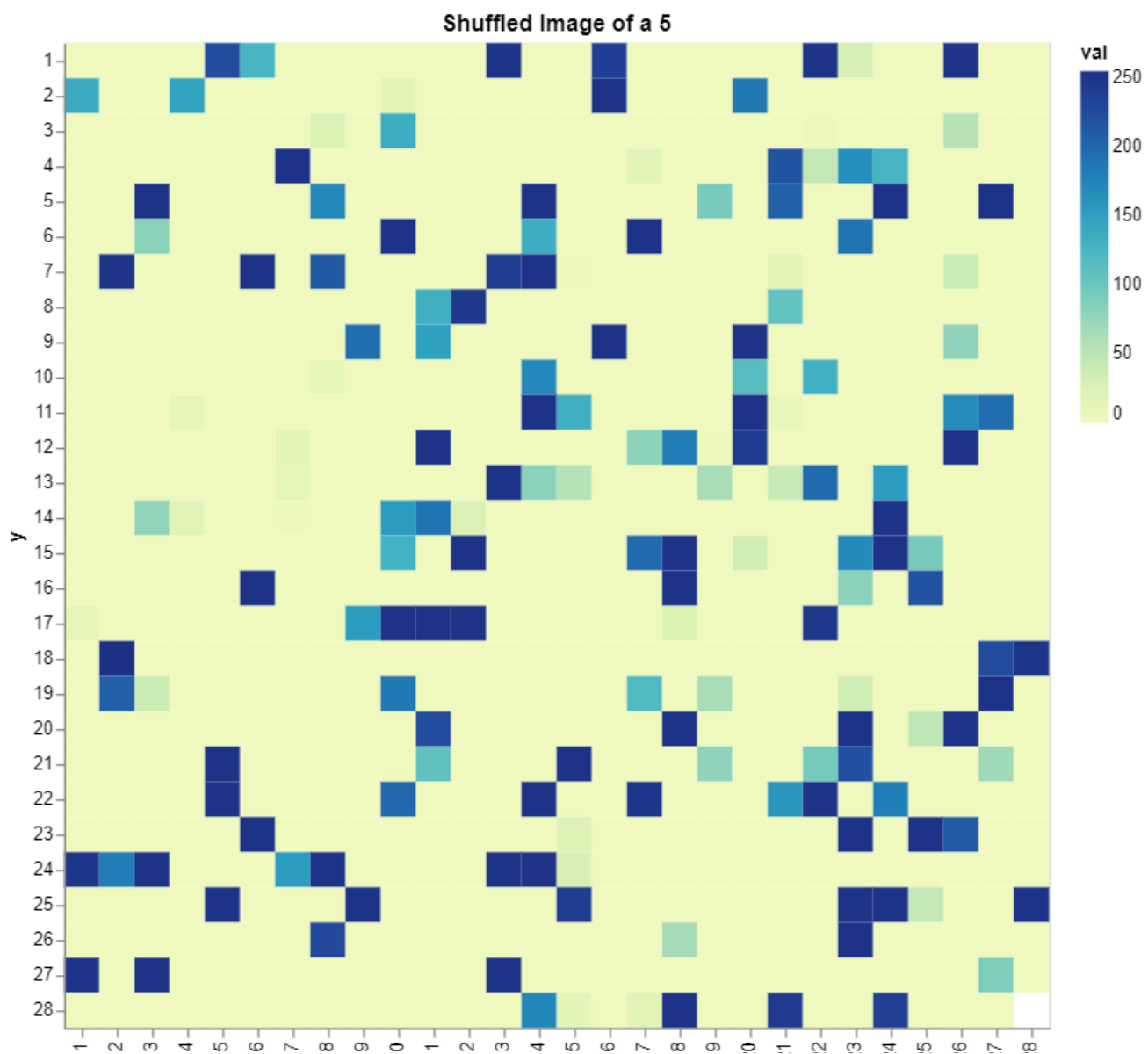
### ▼ What does a NN see?



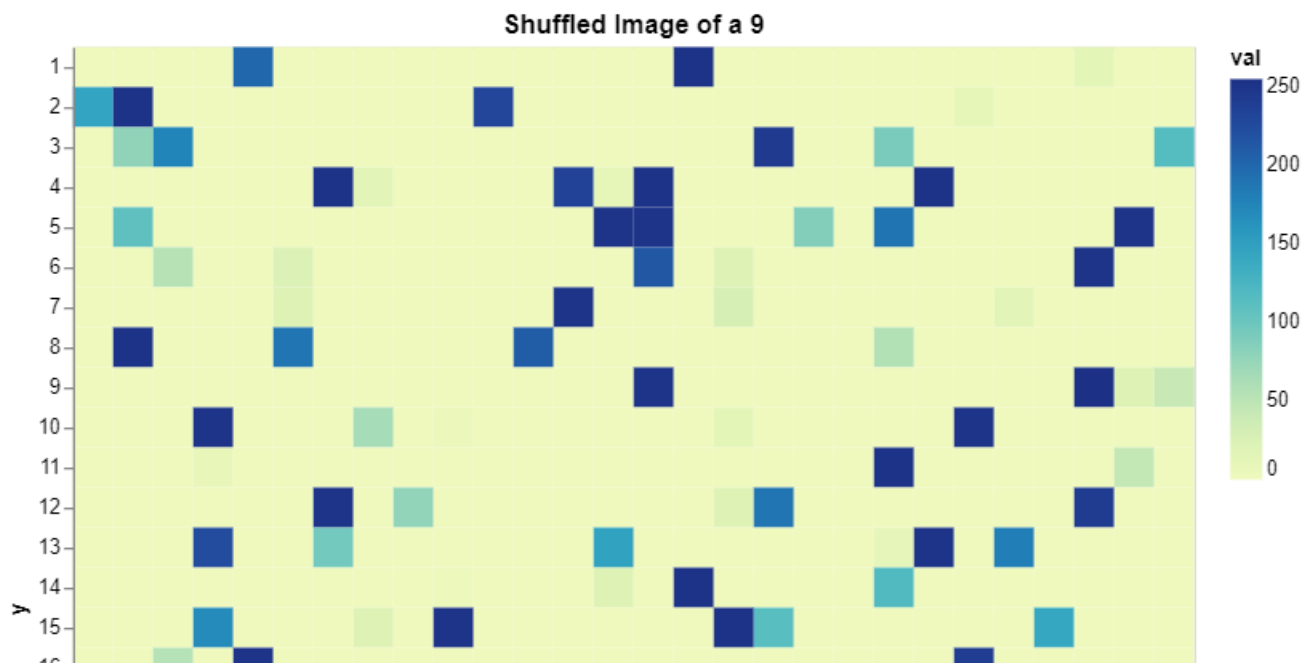
While NN classifiers reach a decent accuracy on this task, it doesn't take into account locations in the model. To see this, let's shuffle each image in MNIST using the **same** shuffling order.



```
im = draw_image(0, shuffle=True)
im
```

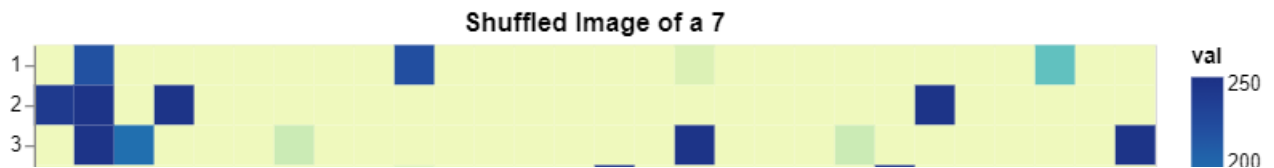


```
im = draw_image(4, shuffle=True)
im
```



```
im = draw_image(15, shuffle=True)
```

```
im
```



Can you recognize what those images are? Let's train an MLP classifier on the shuffled images and report the test accuracy.

```
# Create model
```

```
model = KerasClassifier(build_fn=create_model,
                        epochs=2,
                        batch_size=20,
                        verbose=0)
```

```
shuffled_features = sklearn.utils.shuffle(features, random_state=1234)
```

```
# Fit model
```

```
model.fit(x=df_train[shuffled_features].astype(float),
          y=df_train["class"])
```

```
<keras.callbacks.History at 0x7ff256617b50>
```

```
# Predict on test set
```

```
df_test["predict"] = model.predict(df_test[shuffled_features])
correct = (df_test["predict"] == df_test["class"])
accuracy = correct.sum() / correct.size
output = "accuracy: ", accuracy
output
```

```
('accuracy: ', 0.9172)
```

If you implemented correctly, you should see that the test accuracy on the shuffled images is similar to that on the original images. Think for a moment why shuffling doesn't change the accuracy much.

NNs do not inherently take into account locations in the input as humans do. For instance, the model is not aware that the feature at position (4, 4) is closer to the feature at position (5, 4) compared to position (14, 4).

So we can see that an NN classifier does not take into account the spatial information: it's simply maintaining a input feature, and there is no sense of "closeness" between pixels that are spatially close to each other.

## ▼ Convolutions

Instead of standard NNs we are going to instead use Convolutional Neural Networks (CNNs). CNNs are commonly used to learn features from images and time series. They takes into account the spatial information allowing for locality.

Let's first define a CNN layer for images. For now let's assume that the input of the CNN layer is a image, and the output of the CNN layer is also an image, usually of a smaller size compared to the input (without input padding). The parameters of a CNN layer consist of a little NN that is trying to draw a separator of a region of the image. This is called the `filter`.

To get the output of the CNN layer, we overlay the filter on top of the input such that it covers part of the input without crossing the image boundary. We start from the upperleft corner.

1 ×1	0 ×0	1	0
0 ×1	1 ×1	1	1
0	1	0	0
1	0	0	0

**Input**

2		

**Output**

At each overlay position, we apply the filter with the corresponding portion of the input. The filter is converting the region it looks at into a new feature, the same way that we saw last class.

In this illustration, the input shown in blue is of size 4x4, and the filter shown in pink is of size 2x2. The output size 3x3 is determined by the input size and the filter size.

Now we shift the filter to the right.

1	0 <sub>×1</sub>	1 <sub>×0</sub>	0
0	1 <sub>×1</sub>	1 <sub>×1</sub>	1
0	1	0	0
1	0	0	0

**Input**

2	2	

**Output**

We shift the filter to the right again.

1	0	1 <sub>×1</sub>	0 <sub>×0</sub>
0	1	1 <sub>×1</sub>	1 <sub>×1</sub>
0	1	0	0
1	0	0	0

**Input**

2	2	3

**Output**

We can't shift the filter to the right any more since doing so would cross the boundary. Therefore, we start from the first column of the second row



1	0	1	0
0 <sub>x1</sub>	1 <sub>x0</sub>	1	1
0 <sub>x1</sub>	1 <sub>x1</sub>	0	0
1	0	0	0

Input

2	2	3
1		

Output

Moving right again.



## ▼ Pattern Matching



Last week we saw how neural networks were able to learn to match patterns in the training data. For instance they were able to spot certain features in order to distinguish between red and blue points.



CNNs can be used to do the same thing. However, instead of looking at all the features at once they look at small groups of feature.



CNNs can match patterns with filter weights. For instance, if we set the filter to be  $\begin{bmatrix} -0.5 & 0.5 \\ -0.5 & 0.5 \end{bmatrix}$ , it can detect vertical edges.

**Input**

**Output**

In the below example, the first column of the output takes values 1, corresponding to a line in the original image.

0 $\times -0.5$	1 $\times 0.5$	1	1
0 $\times -0.5$	1 $\times 0.5$	1	1
0	1	1	1
0	1	1	1

**Input**

1	0	0
1	0	0
1	0	0

**Output**

What if we shift the edge in the original input to the right by 1 pixel? We can see that the 1's in the output also shift to the right by 1.

0	0 $\times -.5$	1 $\times .5$	1
0	0 $\times -.5$	1 $\times .5$	1
0	0	1	1
0	0	1	1

**Input**

0	1	0
0	1	0
0	1	0

**Output**

Similarly, if we shift the edge to the right by 2 pixels, the 1's in the output shift to the right by 2.

0	0	0 $\times -.5$	1 $\times .5$
0	0	0 $\times -.5$	1 $\times .5$
0	0	0	1
0	0	0	1

**Input**

0	0	1
0	0	1
0	0	1

**Output**

## ▼ Group Exercise A

### ▼ Question 0

Icebreakers

Who are other members of your group today?

Kera McGovern, Grace Leverett and Razan Tatai

📄📄📄📄 FILLME

- What's their favorite flower or plant?

Rose and trees

 FILLME

- Do they prefer to work from home or in the office?

Work from home

 FILLME

## ▼ Question 1

Brainstorm ways that you might write a program tell apart the digits 5 and 6.

# FILLME

By writing 2 by 2 functions

Do these methods need to look at the whole image? Which parts do they need to look at?

# FILLME





Yes, they need to look at the parts of which the image will appear on.

Would these approaches still work if the digits were bigger or smaller? What if they moved around on the page?

If the digits were bigger or smaller, these approaches would generally still work. However, if the image is too small, it might not work because it would be more difficult to detect the angle towards the top of the 5 as well. If the digits moved around on the page, we would have to look at the whole image in order to look at the pixels correctly for the methods to apply to determine which digit it is. If our method used relative location to detect where the number is on the page, then these approaches would work, because they would check where the pixels are located in relation to other pixels that makes up the digit.

## ▼ Question 2





Now it is your turn to finish the result of the CNN computations. Write down the full result of applying this CNN layer to the given input.

```
#     FILLME
# 2 2 3, 1 2 1, 1 1 0
```

If we increase the height of the input by 1, how would the size of the output change? If we increase the width of the filter by 1, how would the size of the output change?\*\*





```
#     FILLME
# If the input's height is increased by 1, the output would be 4x3.
```

In the above illustrations, it seems that a CNN layer processes the input image in a sequential order. Are computations at different positions dependent upon each other? Can we use a different order such as starting from the bottom right corner?

```
#     FILLME
Computations are at different positions dependent upon each other and we can use a different
```

### ▼ Question 3

Design a filter to detect edges in the horizontal direction.\*\*

```
#     FILLME
def threshold(img, lowThresholdRatio=0.05, highThresholdRatio=0.09):

    highThreshold = img.max() * highThresholdRatio;
    lowThreshold = highThreshold * lowThresholdRatio;

    M, N = img.shape
    res = np.zeros((M,N), dtype=np.int32)

    weak = np.int32(25)
    strong = np.int32(255)





    strong_i, strong_j = np.where(img >= highThreshold)
    zeros_i, zeros_j = np.where(img < lowThreshold)

    weak_i, weak_j = np.where((img <= highThreshold) & (img >= lowThreshold))

    res[strong_i, strong_j] = strong
    res[weak_i, weak_j] = weak

    return (res, weak, strong)
```

Design a filter to detect edges along a diagonal direction.\*\*

```
#     FILLME
def features_filters(img):
    Kx = np.array([[ -1,  0,  1], [-2,  0,  2], [-1,  0,  1]], np.float32)
    Ky = np.array([[ 1,  2,  1], [ 0,  0,  0], [-1, -2, -1]], np.float32)

    Ix = ndimage.filters.convolve(img, Kx)
    Iy = ndimage.filters.convolve(img, Ky)

    G = np.hypot(Ix, Iy)
    G = G / G.max() * 255
    theta = np.arctan2(Iy, Ix)

    return (G, theta)
```

Is it possible to design a filter to detect other edges such as curves?

```
#     FILLME
# Yes
```

## ▼ Unit B

### ▼ Multiple Filters

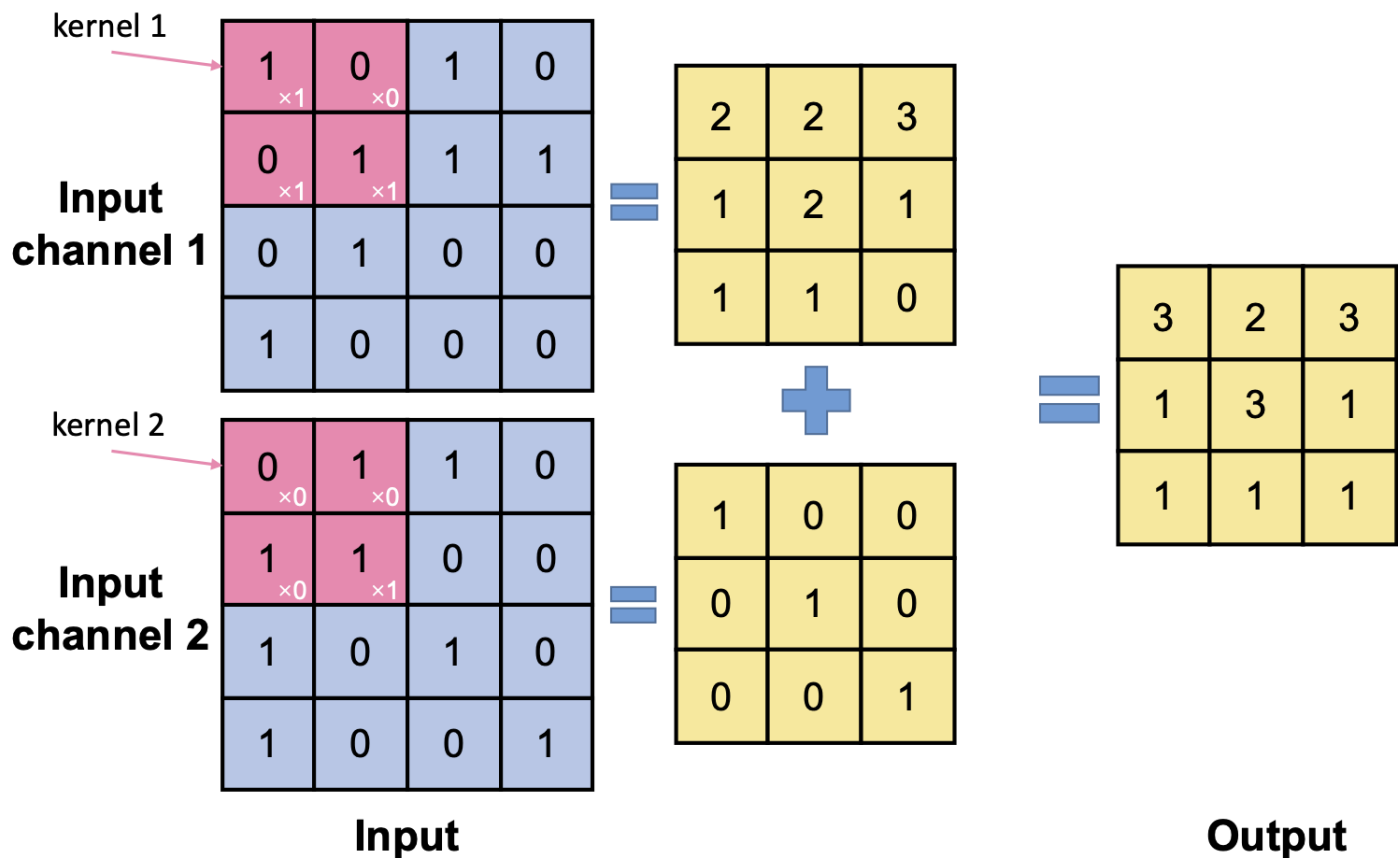
In practice, we want to go beyond only being able to detect a single type of edge. Therefore, instead of only using a single filter, we use multiple "output channels", each with a different filter, such that each output channel can detect a different kind of pattern.

The output shape is thereby augmented by an output channel dimension, forming a 3-D shape of size (output height, output width, output channels).

Similarly, the input can also have multiple channels: for example, the input may be color image split into red, green, and blue channels. Or it can be the output of a previous CNN layer with multiple output channels.

Instead of using a single filter, we use one filter per input channel, then apply the convolution operation to each input channel independently. This process is illustrated below, where for

simplicity we only use a single output channel.



One of the tricky parts of CNNs is keeping track of all of the elements grouped together.

1. input: (num images, input height, input width, input channels).
2. output: (num images, output height, output width, output channels).
3. filter: (filter height, filter width, input channels, output channels).

## ▼ CNN Visualization

In the above discussions, we can see that a CNN layer can detect edges in the input image. By stacking multiple layers of CNNs together, it can learn to build up more and more sophisticated pattern matchers, detecting not only edges, but also mid-level and high-level patterns.

To get a sense of how a CNN works in practice and the types of patterns it can match, let us look at the CNN Explainer. For now let's just play with the demo on the top of the website.

[CNN Explainer](#)



There are several things to look at in the tool.

1. What's the height and width of the input image?
2. What's the number of the output channels of conv\_1\_1?
3. What kind of patterns does the model use for each image?

## ▼ CNN in Keras

Let's take a look at how to create a CNN layer in Keras.

```
Conv2D(  
    filters,  
    kernel_size,  
    strides=(1, 1),  
    padding="valid",  
)
```

The argument `filters` specifies the number of output channels. The argument `kernel_size` is a tuple (height, width) specifying the size of the filter.

Now let's discuss what `strides` does. In the above convolution layer examples, when we shift the filter to the right, we move by 1 pixel; similarly, when we move the filter down, we move down 1 pixel.

We can generalize the step size of movements using `strides`. For example, we can use a stride of 2 along the width dimension, so we move by two pixels each time we move right (note that we still move down by 1 pixel since the stride along the height dimension is 1), resulting an output of size 3 x 2.

1 ×1	0 ×0	1	0
0 ×1	1 ×1	1	1
0	1	0	0
1	0	0	0

Input

2	

Output

1	0	1	0
0 <sub>x1</sub>	1 <sub>x0</sub>	1	1
0 <sub>x1</sub>	1 <sub>x1</sub>	0	0
1	0	0	0

Input

2	3
1	

Output

1	0	1	0
0	1	1 <sub>×1</sub>	1 <sub>×0</sub>
0	1	0 <sub>×1</sub>	0 <sub>×1</sub>
1	0	0	0

**Input**

2	3
1	1

**Output**

1	0	1	0
0	1	1	1
0	1	0 <sub>x1</sub>	0 <sub>x0</sub>
1	0	0 <sub>x1</sub>	0 <sub>x1</sub>

# Input

2	3
1	1
1	0

# Output

 **Student question:** What would the output be if we use a stride of 2 both along the width dimension and the height dimension?

#     FILLME  
pass

## ▼ Convolution Layers in Keras

Now we are ready to implement a CNN layer in Keras! First, we need to import `Conv2D` from `keras.layers`.

```
from keras.layers import Conv2D
```

Now let's use Keras to verify the convolution results we calculated before.

1	0	1	0
0	1	1	1
0	1	0 <sub>x1</sub>	0 <sub>x0</sub>
1	0	0 <sub>x1</sub>	0 <sub>x1</sub>

# Input

2	2	3
1	2	1
1	1	0

# Output

Note that we need to do lots of reshapes to add the sample dimension, or the input/output channel dimension. To recap, the relevant shapes are:

1. input: (num samples, input height, input width, input channels).
2. output: (num samples, output height, output width, output channels).
3. filter: (filter height, filter width, input channels, output channels).

```
input = [[1, 0, 1, 0],
         [0, 1, 1, 1],
         [0, 1, 0, 0],
         [1, 0, 0, 0]]
filter = [[1, 0.],
          [1, 1.]]
```

```
def cnn(input, filter):
    # Code to shape the correct
    filter = tf.convert_to_tensor(filter, dtype=tf.float32)
    filter = tf.reshape(filter, (2, 2, 1, 1))
    input = tf.convert_to_tensor(input, dtype=tf.float32)
    input = tf.reshape(input, (1, 4, 4, 1))
```

# Call Keras

```
.. ---- .. --
cnn_layer = Conv2D(filters=1, kernel_size=(2, 2))
cnn_layer(input)
cnn_layer.set_weights((filter, tf.convert_to_tensor([0.])))
output = cnn_layer(input)

# Output
return tf.reshape(output, (3, 3))

print(cnn(input, filter))

tf.Tensor(
[[2. 2. 3.]
 [1. 2. 1.]
 [1. 1. 0.]], shape=(3, 3), dtype=float32)
```

Nice, our calculations were correct!

## ▼ Pooling

In a convolution layer, we took the convolution between the filter and a portion of the input to calculate the output. However sometimes these areas are very small. What if we want a feature over a larger area?

If we simply take the max value of that portion of input instead of using the convolution, we get a max pooling layer, as illustrated below.

1	0	1	0
0	1	1	1
0	1	0	0
1	0	0	0

max

**Input**

1		

**Output**

1	0	1	0
0	1	1	1
0	1	0	0
1	0	0	0

max

**Input**

1	1	

**Output**



1	0	1	0
0	1	1	1
0	1	0	0
1	0	0	0

**Input**

1	1	1

**Output**

1	0	1	0
0	1	1	1
0	1	0	0
1	0	0	0

**Input**

1	1	1
1		

**Output**

1	0	1	0
0	1	1	1
0	1	0	0
1	0	0	0

**Input**

1	1	1
1	1	

**Output**

1	0	1	0
0	1	1	1
0	1	0	0
1	0	0	0

**Input**

1	1	1
1	1	1

**Output**

1	0	1	0
0	1	1	1
0	1	0	0
1	0	0	0

max

**Input**

1	1	1
1	1	1
1		

**Output**

1	0	1	0
0	1	1	1
0	1	0	0
1	0	0	0

max

**Input**

1	1	1
1	1	1
1	1	

**Output**

1	0	1	0
0	1	1	1
0	1	0	0
1	0	0	0

**Input**

1	1	1
1	1	1
1	1	0

**Output**

Let's take a look at how to create a max pooling layer in Keras.

```
MaxPool2D(
    pool_size=(2, 2),
    strides=None,
    **kwargs
)
```

Notice how similar it is to convolution layers? Can you infer what `strides` does here?

Now let's use Keras to verify the max pooling results above.

```
from keras.layers import MaxPool2D
```

```
input = [
    [1, 0, 1, 0],
    [0, 1, 1, 1],
    [0, 1, 0, 0],
    [1, 0, 0, 0]
]
```

```

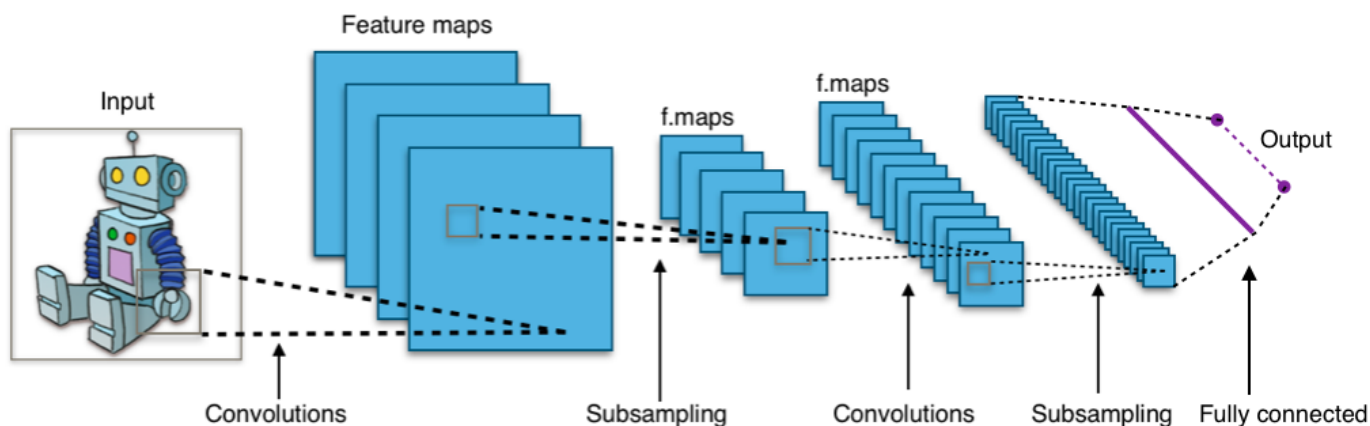
input_shape = (1, 4, 4, 1)
input = tf.convert_to_tensor(input, dtype=tf.float32)
input = tf.reshape(input, input_shape)
pooling_layer = MaxPool2D(pool_size=(2, 2),
                           strides=(1, 1))
output = pooling_layer(input)
print (tf.reshape(output, (3, 3)))

tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 0.]], shape=(3, 3), dtype=float32)

```

In the above example we used `strides=(1,1)`.

## ▼ Putting Everything Together



Now we can put everything together to build a full CNN classifier for MNIST classification. Below shows an example model, where we need to use a `Reshape` layer to reshape the input into a single-channel 2-D image, as well as a `Flatten` layer to flatten the feature map back to a vector.

```
from keras.layers import Flatten, Reshape
```

```

def create_cnn_model():
    # create model
    input_shape = (28, 28, 1)
    model = Sequential()
    model.add(Reshape(input_shape))
    model.add(Conv2D(32, kernel_size=(3, 3), activation="relu"))
    model.add(MaxPool2D(pool_size=(2, 2)))
    model.add(Conv2D(64, kernel_size=(3, 3), activation="relu"))
    model.add(MaxPool2D(pool_size=(2, 2)))
    model.add(Flatten())

```

```

model.add(Dense(10, activation="softmax")) # output a vector of size 10
# Compile model
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="adam",
              metrics=["accuracy"])

return model

#
# create model
model = KerasClassifier(build_fn=create_cnn_model,
                        epochs=2,
                        batch_size=20,
                        verbose=0)

# fit model
model.fit(x=df_train[features].astype(float),
        y=df_train["class"])
df_test["predict"] = model.predict(df_test[features])
correct = (df_test["predict"] == df_test["class"])
accuracy = correct.sum() / correct.size
print ("accuracy: ", accuracy)

accuracy: 0.9791

```

We are able to get much better accuracy than using basic NNs!

## ▼ Group Exercise B

### ▼ Question 1

Apply the CNN model to the shuffled MNIST dataset. What accuracy do you get? Is that what you expected?

```

# FILLME
shuffled_features1 = sklearn.utils.shuffle(features, random_state=1234)
# fit model
model.fit(x=df_train[shuffled_features1].astype(float),
        y=df_train["class"])
df_test["predict"] = model.predict(df_test[shuffled_features1])
correct = (df_test["predict"] == df_test["class"])
accuracy = correct.sum() / correct.size
print ("accuracy: ", accuracy)

accuracy: 0.9329





```

## ▼ Question 2




For this question we will use the CNN Explainer website.

### [CNN Explainer](#)

- Use the tool under the section "Understanding Hyperparameters" to figure out the output shape of each layer in the above CNN model.

```
#     FILLME
# Layer reshape input - (28, 28)
# Layer Conv2D 1 - (26, 26)
# Layer MaxPool2D 1 - (13, 13)
# Layer Conv2D 2 - (11, 11)
# Layer MaxPool2D 2- (5, 5)
```

- Use `print (model.model.summary())` to print the output shape of each layer. Did you get the same results as above?

```
#     FILLME
print (model.model.summary())
```

Model: "sequential\_7"

Layer (type)	Output Shape	Param #
reshape_4 (Reshape)	(20, 28, 28, 1)	0
conv2d_9 (Conv2D)	(20, 26, 26, 32)	320
max_pooling2d_9 (MaxPooling2D)	(20, 13, 13, 32)	0
conv2d_10 (Conv2D)	(20, 11, 11, 64)	18496
max_pooling2d_10 (MaxPooling2D)	(20, 5, 5, 64)	0
flatten_4 (Flatten)	(20, 1600)	0
dense_11 (Dense)	(20, 10)	16010
Total params: 34,826		
Trainable params: 34,826		
Non-trainable params: 0		

None

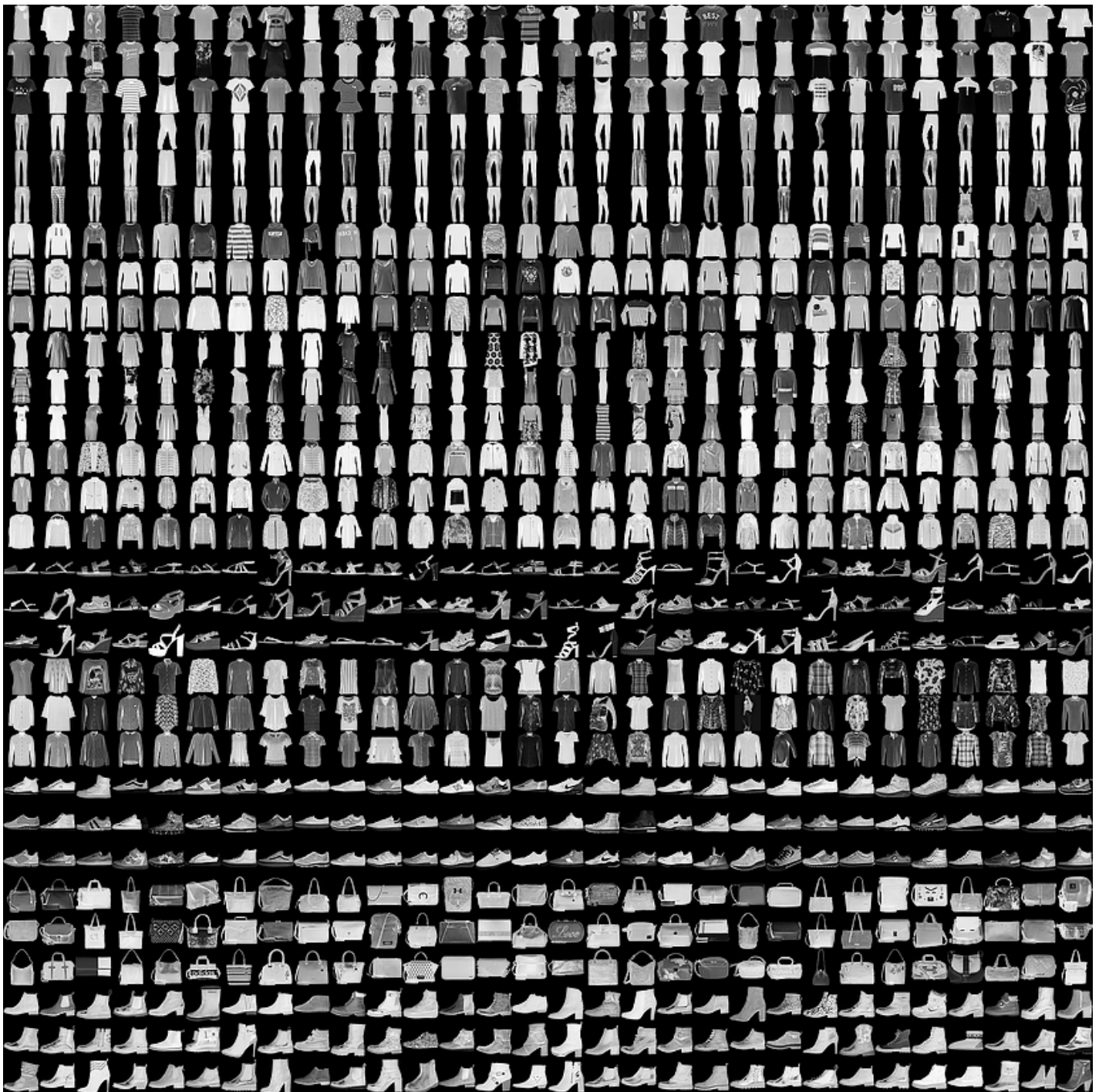
### ▼ Question 3

Let's apply our model to a different dataset, [Fashion MNIST](#), where the goal is to classify an image into one of the below 10 classes:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Some examples from the dataset are shown below, where each class takes three rows.





We have processed the dataset into the same format as MNIST:

```
df_train = pd.read_csv('https://srush.github.io/BT-AI/notebooks/fashion_mnist_train.csv.gz',  
df_test = pd.read_csv('https://srush.github.io/BT-AI/notebooks/fashion_mnist_test.csv.gz', co  
df_train
```

	class	1x1	1x2	1x3	1x4	1x5	1x6	1x7	1x8	1x9	1x10	1x11	1x12	1x13	1x14
<b>0</b>	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>1</b>	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>2</b>	6	0	0	0	0	0	0	0	5	0	0	0	105	92	101
<b>3</b>	0	0	0	0	1	2	0	0	0	0	0	114	183	112	55
<b>4</b>	3	0	0	0	0	0	0	0	0	0	0	0	0	46	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
<b>59995</b>	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>59996</b>	1	0	0	0	0	0	0	0	0	0	0	83	155	136	116
<b>59997</b>	8	0	0	0	0	0	0	0	0	0	0	1	0	0	87
<b>59998</b>	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0

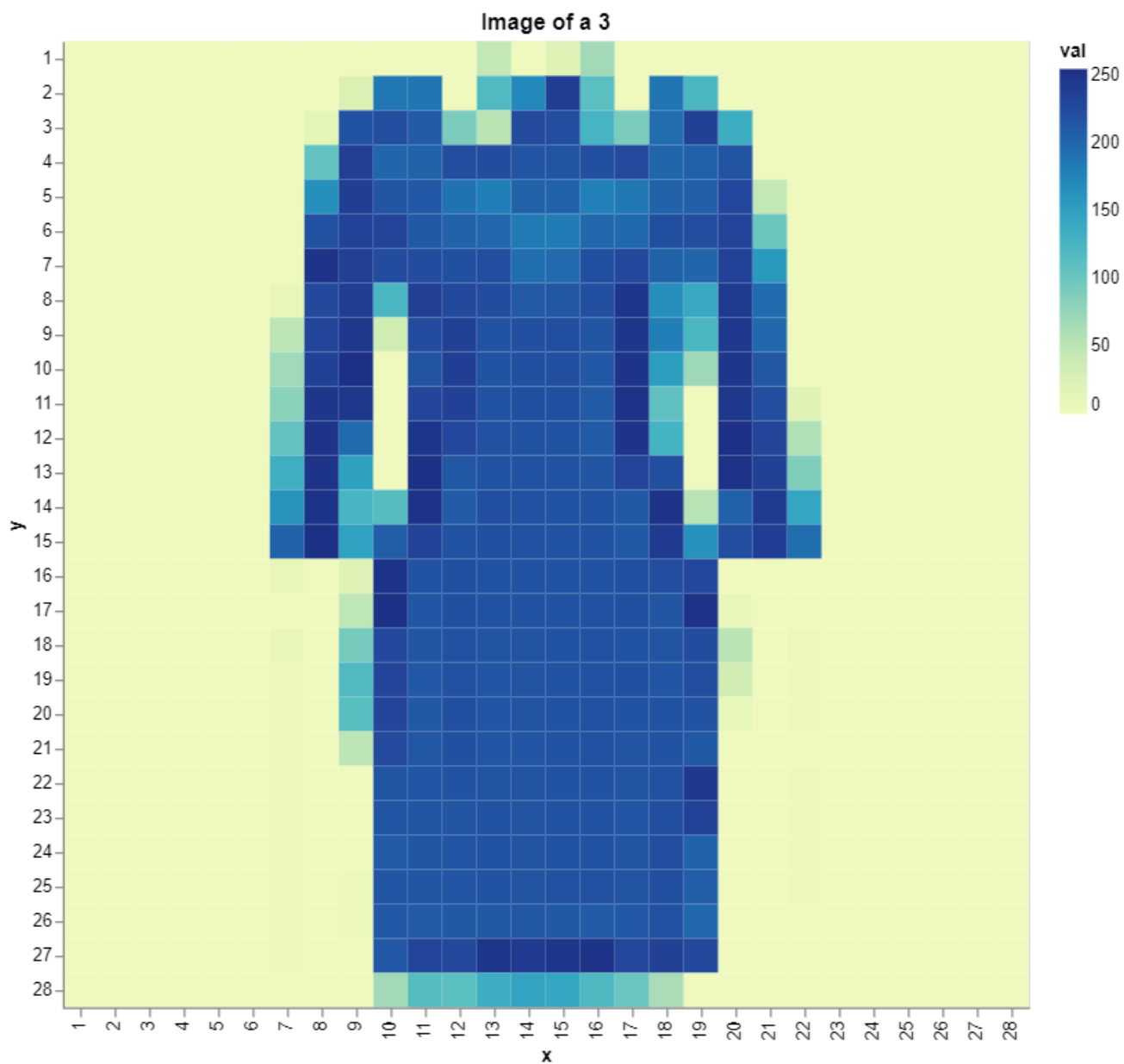
Let's visualize some examples first

===== rows ===== columns

draw\_image(1)



```
draw_image(4)
```



```
draw_image(10)
```



Apply the CNN model to this dataset and print out the accuracy.

```
28-
# FILLME
# fit model
model.fit(x=df_train[features].astype(float),
          y=df_train["class"])
df_test["predict"] = model.predict(df_test[features])
correct = (df_test["predict"] == df_test["class"])
accuracy = correct.sum() / correct.size
print ("accuracy: ", accuracy)
```

accuracy: 0.8811

---

✓ 0s completed at 7:48 PM

