Implementing data augmentation

In the previous scenario, we learned about how CNNs help in predicting the class of an image when it is translated. While this worked well for translations of up to 5 pixels, anything beyond that is likely to have a very low probability for the right class. In this section, we'll learn how to ensure that we predict the right class, even if the image is translated by a considerable amount.

To address this challenge, we'll train the neural network by translating the input images by 10 pixels randomly (both toward the left and the right) and passing them to the network. This way, the same image will be processed as a different image in different passes since it will have had a different amount of translation in each pass.

Before we leverage augmentations to improve the accuracy of our model when images are translated, let's learn about the various augmentations that can be done on top of an image.

Image augmentations

So far, we have learned about the issues image translation can have on a model's prediction accuracy. However, in the real world, we might encounter various scenarios, such as the following:

- Images are rotated slightly
- Images are zoomed in/out (scaled)
- Some amount of noise is present in the image
- Images have low brightness
- Images have been flipped
- Images have been sheared (one side of the image is more twisted)

A neural network that does not take the preceding scenarios into consideration won't provide accurate results, just like in the previous section, where we had a neural network that had not been explicitly trained on images that had been heavily translated.

Image augmentations come in handy in scenarios where we create more images from a given image. Each of the created images can vary in terms of rotation, translation, scale, noise, and brightness. Furthermore, the extent of the variation in each of these parameters can also vary (for example, translation of a certain image in a given iteration can be +10 pixels, while in a different iteration, it can be -5 pixels).

The `augmenters` class in the `imgaug` package has useful utilities for performing these augmentations. Let's take a look at the various utilities present in the `augmenters` class for generating augmented images from a given image. Some of the most prominent augmentation techniques are as follows:

- Affine transformations
- Change brightness
- Add noise

Note that PyTorch has a handy image augmentation pipeline in the form of `torchvision.transforms`*. However, we still opted to introduce a different library primarily because of the larger variety of options* `imgaug` *contains, as well as due to the ease of explaining augmentations to a new user. You are encouraged to research the torchvision transforms as an exercise and recreate all the functions that are presented to strengthen your understanding.*

Affine transformations

Affine transformations involve translating, rotating, scaling, and shearing an image. They can be performed in code using the `Affine` method that's present in the `augmenters` class. Let's take a look at the parameters present in the `Affine` method by looking at the following screenshot. Here, we have defined all the parameters of the `Affine` method:

```
iaa.Affine(scale=1.0, translate_percent=None, translate_px=None, rotate=0.0, shear=0.0,
order=1, cval=0, mode='constant', fit_output=False, backend='auto', name=None,
deterministic=False, random_state=None)
```

Some of the important parameters in the `Affine` method are as follows:

- `scale` specifies the amount of zoom that is to be done for the image
- `translate_percent` specifies the amount of translation as a percentage of the image's height and width
- `translate_px` specifies the amount of translation as an absolute number of pixels
- `rotate` specifies the amount of rotation that is to be done on the image
- `shear` specifies the amount of rotation that is to be done on part of the image

Before we consider ay other parameters, let's understand where scaling, translation, and rotation come in handy.

The code for this section is available as `Image_augmentation.ipynb` *in the* `Chapter04` *folder of this book's GitHub repository -* https://tinyurl.com/mcvp-packt

Fetch a random image from the training dataset for `fashionMNIST`:

1. Download images from the Fashion-MNIST dataset:

```
from torchvision import datasets
import torch
data_folder = '/content/' # This can be any directory
# you download FMNIST to
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                               train=True)
```

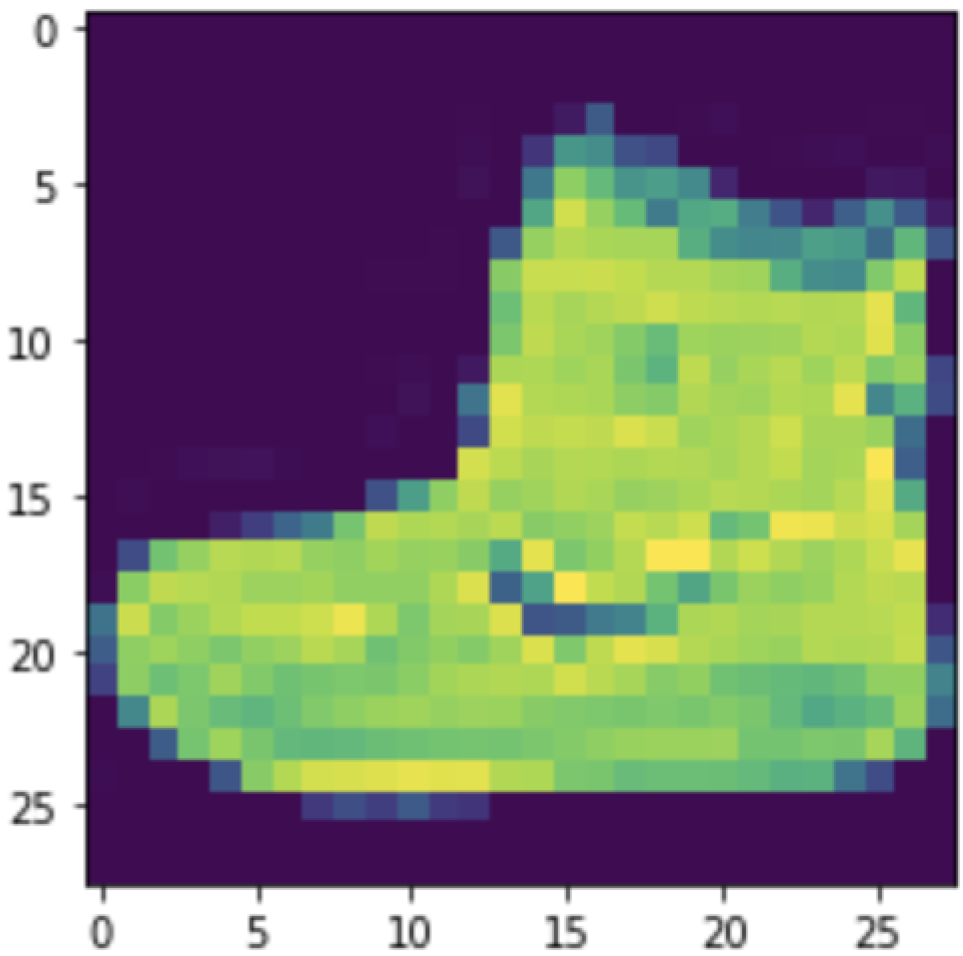
2. Fetch an image from the downloaded dataset:

```
tr_images = fmnist.data
tr_targets = fmnist.targets
```

3. Let's plot the first image:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(tr_images[0])
```

The output of the preceding code is as follows:



Perform scaling on top of the image:

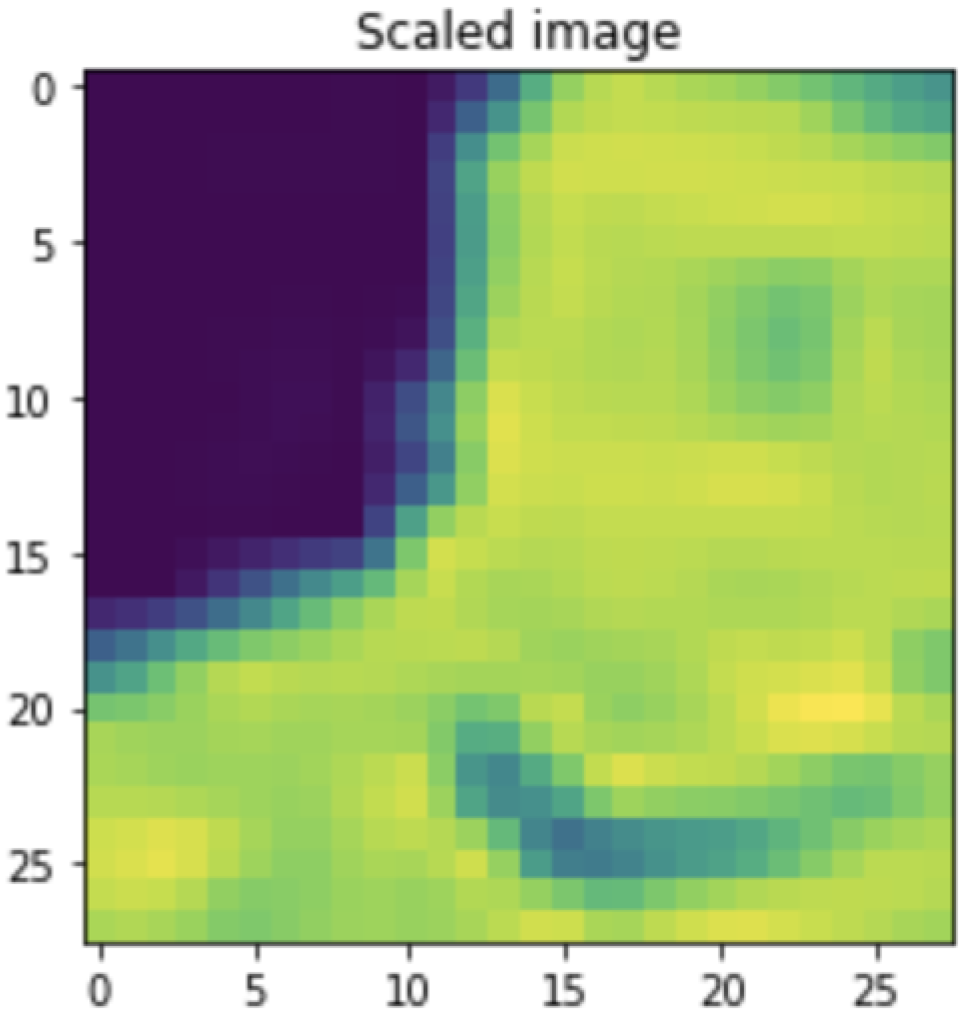
1. Define an object that performs scaling:

```
from imgaug import augmenters as iaa
aug = iaa.Affine(scale=2)
```

2. Specify that we want to augment the image using the `augment_image` method, which is available in the `aug` object, and plot it:

```
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Scaled image')
```

The output of the preceding code is as follows:

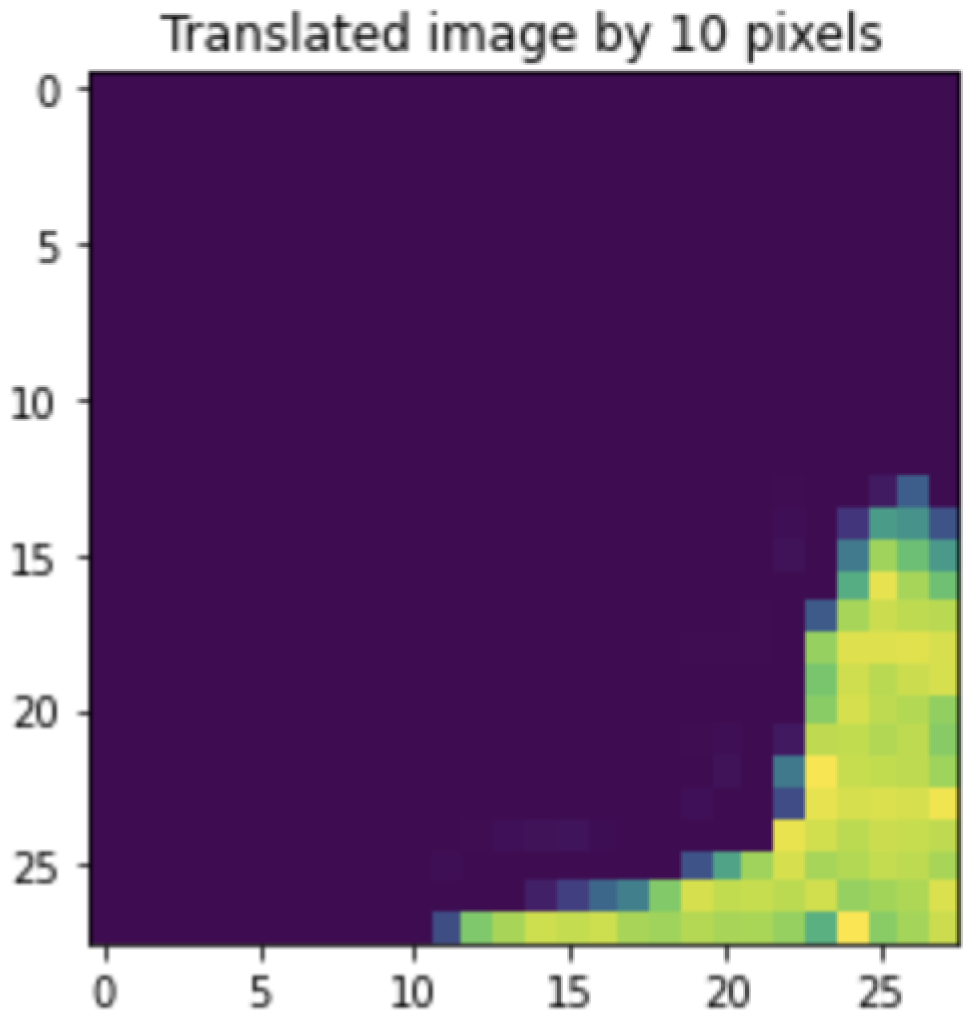


In the preceding output, the image has been zoomed into considerably. This has resulted in some pixels being cut from the original image since the output shape of the image hasn't changed.

Now, let's take a look at a scenario where an image has been translated by a certain number of pixels using the `translate_px` parameter:

```
aug = iaa.Affine(translate_px=10)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Translated image by 10 pixels')
```

The output of the preceding code is as follows:



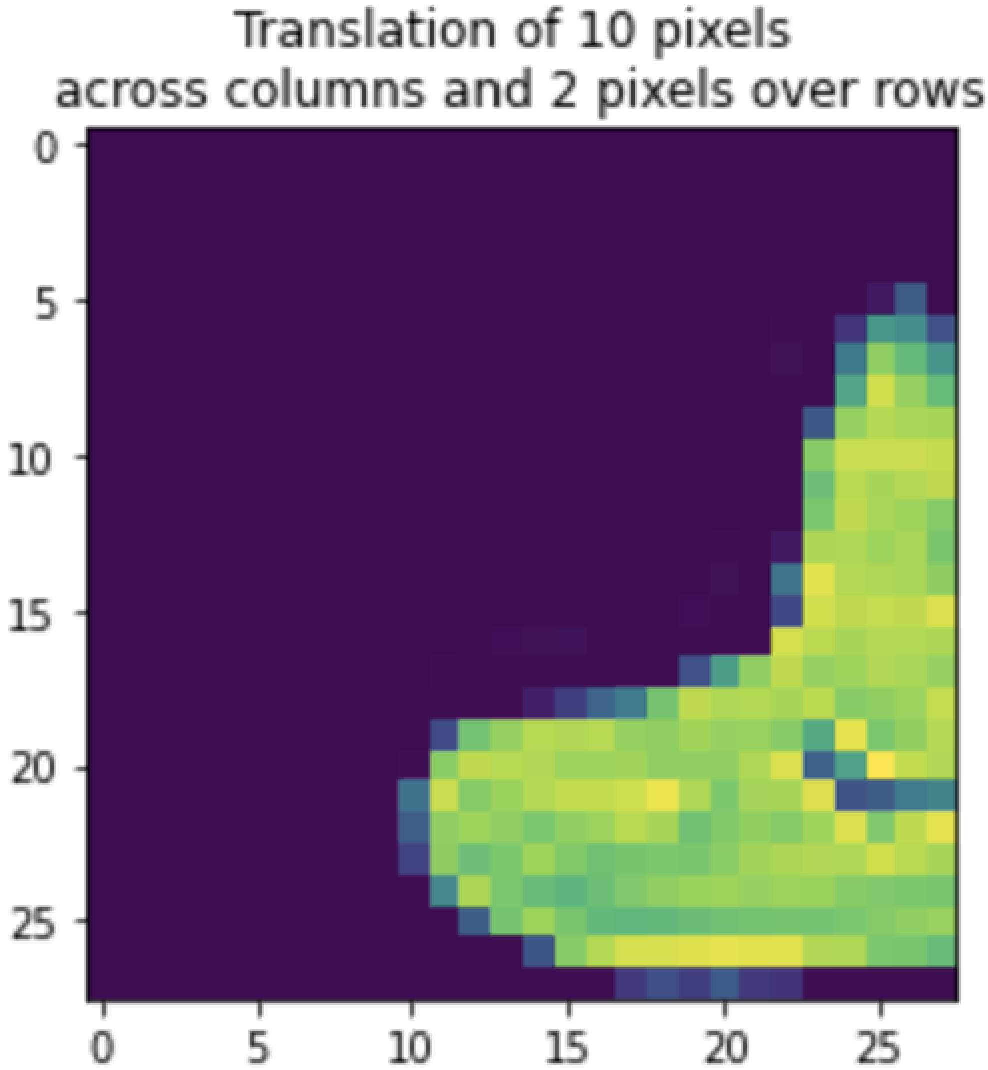
In the preceding output, the translation by 10 pixels has happened across both the x and y axes.

If we want to perform translation more in one axis and less in the other axis, we must specify the amount of translation we want in each axis:

```
aug = iaa.Affine(translate_px={'x':10,'y':2})
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Translation of 10 pixels \nacross columns \
and 2 pixels over rows')
```

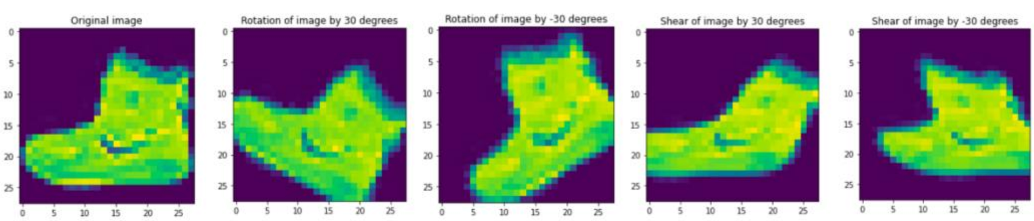
Here, we have provided a dictionary that states the amount of translation in the x and y axes in the `translate_px` parameter.

The output of the preceding code is as follows:



The preceding output shows that more translation happened across columns compared to rows. This has also resulted in a certain portion of the image being cropped.

Now, let's consider the impact rotation and shearing have on image augmentation:

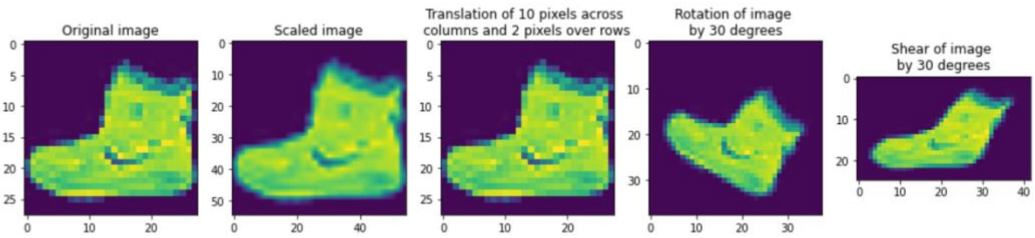


In the majority of the preceding outputs, we can see that certain pixels were cropped out of the image post-transformation. Now, let's take a look at how the rest of the parameters in the `Affine` method help us not lose information due to cropping post-augmentation.

`fit_output` is a parameter that can help with the preceding scenario. By default, it is set to `False`. However, let's see how the preceding outputs vary when we specify `fit_output` as `True` when we scale, translate, rotate, and shear the image:

```
plt.figure(figsize=(20,20))
plt.subplot(161)
plt.imshow(tr_images[0])
plt.title('Original image')
plt.subplot(162)
aug = iaa.Affine(scale=2, fit_output=True)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Scaled image')
plt.subplot(163)
aug = iaa.Affine(translate_px={'x':10,'y':2}, fit_output=True)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Translation of 10 pixels across \ncolumns and \
2 pixels over rows')
plt.subplot(164)
aug = iaa.Affine(rotate=30, fit_output=True)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Rotation of image \nby 30 degrees')
plt.subplot(165)
aug = iaa.Affine(shear=30, fit_output=True)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Shear of image \nby 30 degrees')
```

The output of the preceding code is as follows:



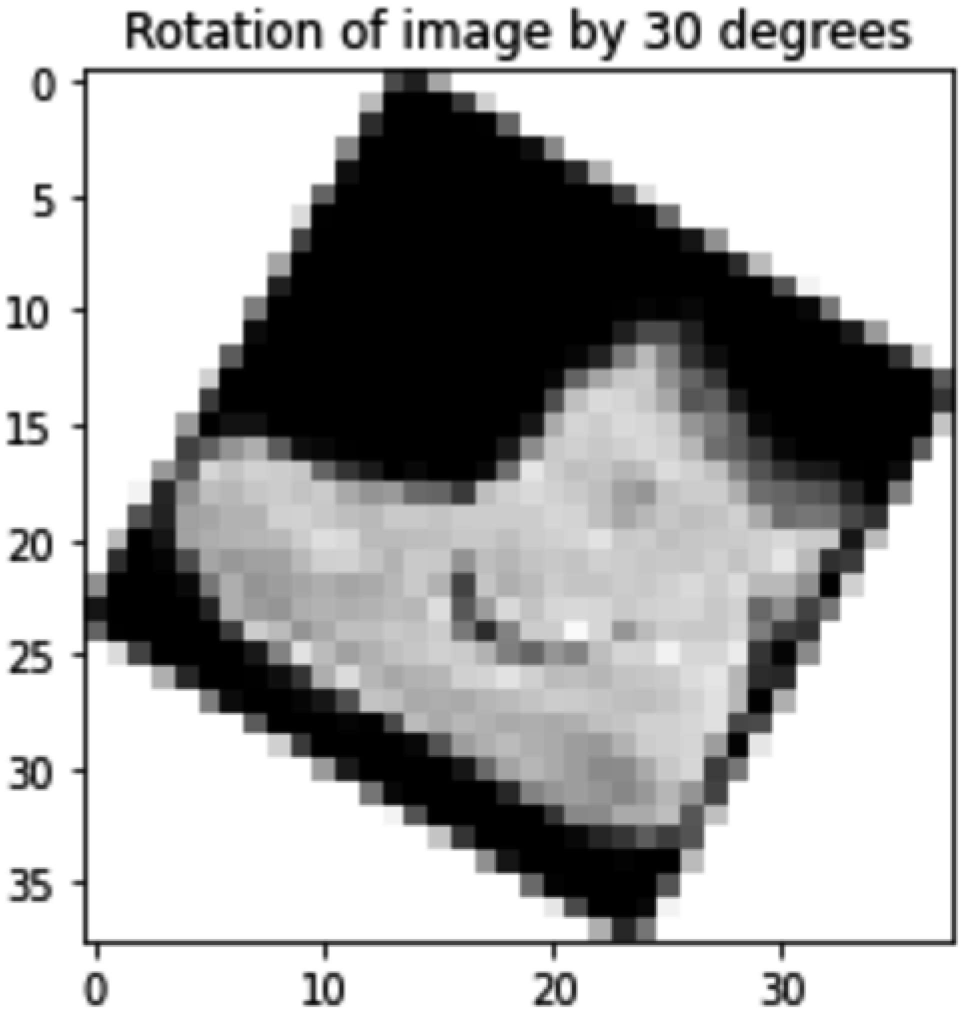
Here, we can see that the original image hasn't been cropped and that the size of the augmented image increased to account for the augmented image not being cropped (in the scaled image's output or when rotating the image by 30 degrees). Furthermore, we can also see that the activation of the `fit_output` parameter has negated the translation that we expected in the translation of a 10-pixel image (this is a known behavior, as explained in the documentation).

Note that when the size of the augmented image increases (for example, when the image is rotated), we need to figure out how the new pixels that are not part of the original image should be filled in.

The `cval` parameter solves this issue. It specifies the pixel value of the new pixels that are created when `fit_output` is `True`. In the preceding code, `cval` is filled with a default value of 0, which results in black pixels. Let's understand how changing the `cval` parameter to a value of 255 impacts the output when an image is rotated:

```
aug = iaa.Affine(rotate=30, fit_output=True, cval=255)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Rotation of image by 30 degrees')
```

The output of the preceding code is as follows:



In the preceding image, the new pixels have been filled with a pixel value of 255, which corresponds to the color white.

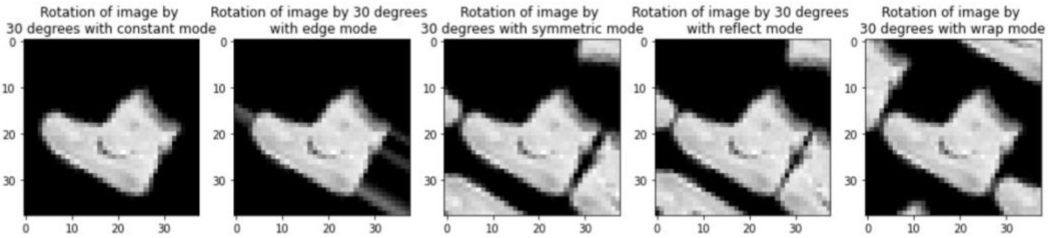
Furthermore, there are different modes we can use to fill the values of newly created pixels. These values, which are for the `mode` parameter, are as follows:

- `constant`: Pads with a constant value.

- `edge`: Pads with the edge values of the array.
- `symmetric`: Pads with the reflection of the vector mirrored along the edge of the array.
- `reflect`: Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.
- `wrap`: Pads with the wrap of the vector along the axis.

The initial values are used to pad the end, while the end values are used to pad the beginning.

The outputs that we receive when `cval` is set to 0 and we vary the `mode` parameter are as follows:

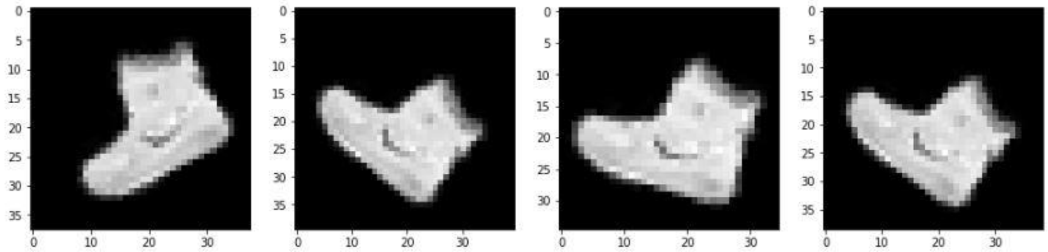


Here, we can see that for our current scenario based on the Fashion-MNIST dataset, it is more desirable to use the `constant` mode for data augmentation.

So far, we have specified that the translation needs to be a certain number of pixels. Similarly, we have specified that the rotation angle should be of a specific degree. However, in practice, it becomes difficult to specify the exact angle that an image needs to be rotated by. Thus, in the following code, we've provided a range that the image will be rotated by. This can be done like so:

```
plt.figure(figsize=(20,20))
plt.subplot(151)
aug = iaa.Affine(rotate=(-45,45), fit_output=True, cval=0, \
                 mode='constant')
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray')
plt.subplot(152)
aug = iaa.Affine(rotate=(-45,45), fit_output=True, cval=0, \
                 mode='constant')
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray')
plt.subplot(153)
aug = iaa.Affine(rotate=(-45,45), fit_output=True, cval=0, \
                 mode='constant')
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray')
plt.subplot(154)
aug = iaa.Affine(rotate=(-45,45), fit_output=True, cval=0, \
                 mode='constant')
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray')
```

The output of the preceding code is as follows:



In the preceding output, the same image was rotated differently in different iterations because we specified a range of possible rotation angles in terms of the upper and lower bounds of the rotation. Similarly, we can randomize augmentations when we are translating or sharing an image.

So far, we have looked at varying the image in different ways. However, the intensity/brightness of the image remains unchanged. Next, we'll learn how to augment the brightness of images.

Changing the brightness

Imagine a scenario where the difference between the background and the foreground is not as distinct as we have seen so far. This means the background does not have a pixel value of 0 and that the foreground does not have a pixel value of 255. Such a scenario can typically happen when the lighting conditions in the image are different.

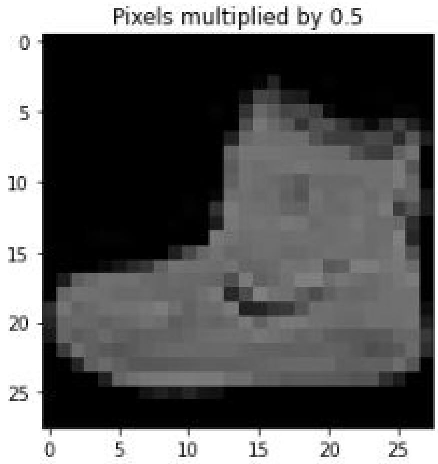
If the background has always had a pixel value of 0 and the foreground has always had a pixel value of 255 when the model has been trained but we are predicting an image that has a background pixel value of 20 and a foreground pixel value of 220, the prediction is likely to be incorrect.

`Multiply` and `Linearcontrast` are two different augmentation techniques that can be leveraged to resolve such scenarios.

The `Multiply` method multiplies each pixel value by the value that we specify. The output of multiplying each pixel value by 0.5 for the image we have been considering so far is as follows:

```
aug = iaa.Multiply(0.5)
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Pixels multiplied by 0.5')
```

The output of the preceding code is as follows:



`Linearcontrast` adjusts each pixel value based on the following formula:

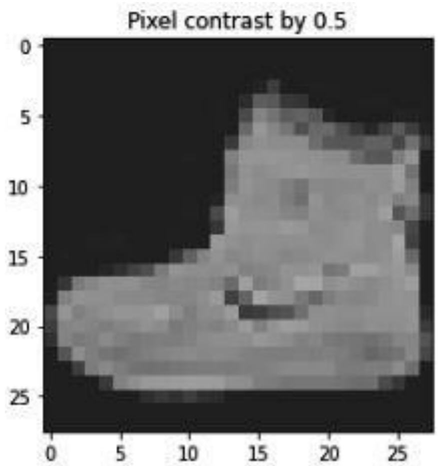
$$127 + \alpha \times (pixelvalue - 127)$$

In the preceding equation, when α is equal to 1, the pixel values remain unchanged. However, when α is less than 1, high pixel values are reduced and low pixel values are increased.

Let's take a look at the impact `Linearcontrast` has on the output of this image:

```
aug = iaa.LinearContrast(0.5)
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Pixel contrast by 0.5')
```

The output of the preceding code is as follows:

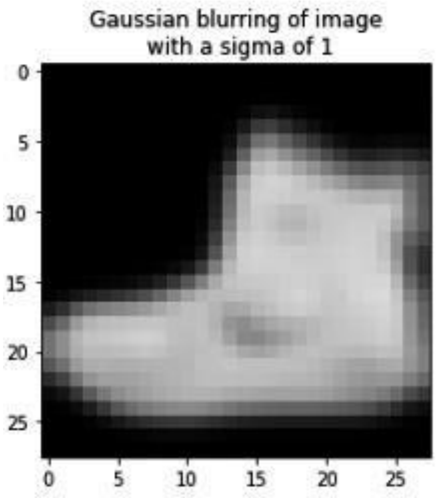


Here, we can see that the background became more bright, while the foreground pixels' intensity reduced.

Next, we'll blur the image to mimic a realistic scenario (where the image can be potentially blurred due to motion) using the `GaussianBlur` method:

```
aug = iaa.GaussianBlur(sigma=1)
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Gaussian blurring of image')
```

The output of the preceding code is as follows:



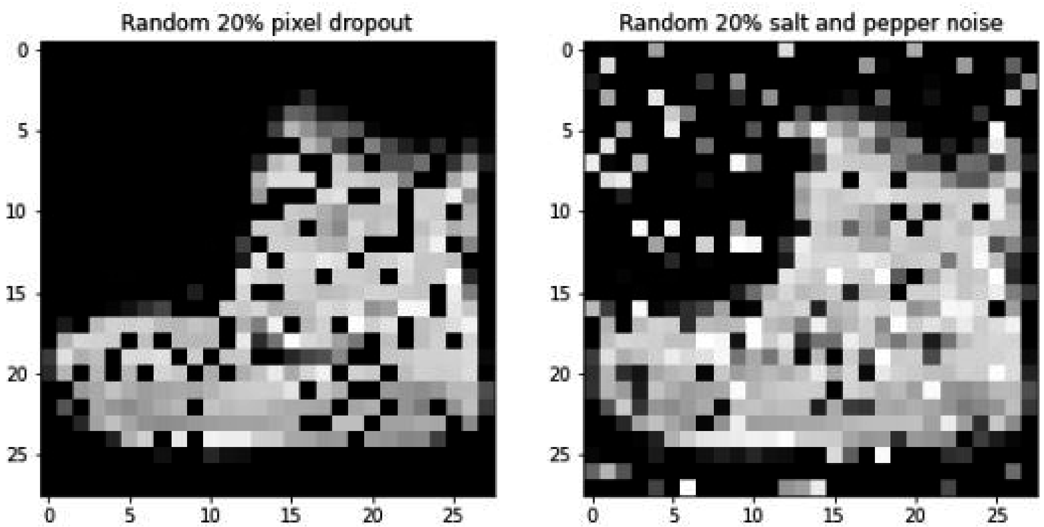
In the preceding image, we can see that the image was blurred considerably and that as the `sigma` value increases (where the default is 0 for no blurring), the image becomes even blurrier.

Adding noise

In a real-world scenario, we may encounter grainy images due to bad photography conditions. `Dropout` and `SaltAndPepper` are two prominent methods that can help in simulating grainy image conditions. Let's take a look at the output of augmenting an image with these two methods:

```
plt.figure(figsize=(10,10))
plt.subplot(121)
aug = iaa.Dropout(p=0.2)
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Random 20% pixel dropout')
plt.subplot(122)
aug = iaa.SaltAndPepper(0.2)
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Random 20% salt and pepper noise')
```

The output of the preceding code is as follows:



Here, we can see that while the `Dropout` method dropped a certain amount of pixels randomly (that is, it converted them so that they had a pixel value of 0), the `SaltAndPepper` method added some white-ish and black-ish pixels randomly to our image.

Performing a sequence of augmentations

So far, we have looked at various augmentations and have also performed. However, in a real-world scenario, we would have to account for as many augmentations as possible. In this section, we will learn about the sequential way of performing augmentations.

Using the `Sequential` method, we can construct the augmentation method using all the relevant augmentations that must be performed. For our example, we'll only consider `rotate` and `Dropout` for augmenting our image. The `Sequential` object looks as follows:

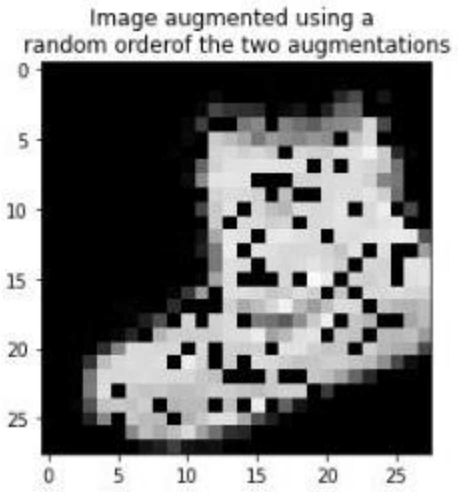
```
seq = iaa.Sequential([
      iaa.Dropout(p=0.2),
      iaa.Affine(rotate=(-30,30))], random_order= True)
```

In the preceding code, we are specifying that we are interested in the two augmentations and have also specified that we're going to be using the `random_order` parameter. The augmentation process is going to be performed randomly between the two.

Now, let's plot the image with these augmentations:

```
plt.imshow(seq.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Image augmented using a \nrandom order \
of the two augmentations')
```

The output of the preceding code is as follows:



From the preceding image, we can see that the two augmentations are performed on top of the original image (you can observe that the image has been rotated and that dropout has been applied).

Performing data augmentation on a batch of images and the need for collate_fn

We have already seen that it is preferable to perform different augmentations in different iterations on the same image.

If we have an augmentation pipeline defined in the `__init__` method, we would only need to perform augmentation once on the input set of images. This means we would not have different augmentations on different iterations.

Similarly, if the augmentation is in the `__getitem__` method – which is ideal since we want to perform a different set of augmentations on each image – the major bottleneck is that the augmentation is performed once for each image. It would be much faster if we were to perform augmentation on a batch of images instead of on one image at a time. Let's understand this in detail by looking at two scenarios where we will be working on 32 images:

- Augmenting 32 images, one at a time
- Augmenting 32 images as a batch in one go

To understand the time it takes to augment 32 images in both scenarios, let's leverage the first 32 images in the training images of the Fashion-MNIST dataset:

The following code is available as `Time_comparison_of_augmentation_scenario.ipynb` in the `Chapter04` folder of this book's GitHub repository - [`https://tinyurl.com/mcvp-packt`](https://tinyurl.com/mcvp-packt)

1. Fetch the first 32 images in the training dataset:

```
from torchvision import datasets
import torch
data_folder = '/content/'
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                                                train=True)
tr_images = fmnist.data
tr_targets = fmnist.targets
```

2. Specify the augmentation to be performed on the images:

```
from imgaug import augmenters as iaa
aug = iaa.Sequential([
              iaa.Affine(translate_px={'x':(-10,10)},
                                          mode='constant'),
            ])
```

Next, we need to understand how to perform augmentation in the `Dataset` class. There are two possible ways of augmenting data:

- Augmenting a batch of images, one at a time
- Augmenting all the images in a batch in one go

Let's understand the time it takes to perform both the preceding scenarios:

- **Scenario 1:** Augmenting 32 images, one at a time:

Calculate the time it takes to augment one image at a time using the `augment_image` method:

```
%%time
for i in range(32):
    aug.augment_image(tr_images[i])
```

It takes ~180 milliseconds to augment for the 32 images.

- **Scenario 2:** Augmenting 32 images as a batch in one go:

Calculate the time it takes to augment the batch of 32 images in one go using the `augment_images` method:

```
%%time
aug.augment_images(tr_images[:32])
```

It takes ~8 milliseconds to perform augmentation on the batch of images.

It is a best practice to augment on top of a batch of images than doing so one image at a time. In addition, the output of the `augment_images` method is a `numpy` array.

However, the traditional `Dataset` class that we have been working on provides the index of one image at a time in the `__getitem__` method. Hence, we need to learn how to use a new function – `collate_fn` – that enables us to perform manipulation on a batch of images.

3. Define the `Dataset` class, which takes the input images, their classes, and the augmentation object as initializers:

```
from torch.utils.data import Dataset, DataLoader
class FMNISTDataset(Dataset):
    def __init__(self, x, y, aug=None):
        self.x, self.y = x, y
        self.aug = aug
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x, y
    def __len__(self): return len(self.x)
```

- Define `collate_fn`, which takes the batch of data as input:

```
    def collate_fn(self, batch):
```

- Separate the batch of images and their classes into two different variables:

```
        ims, classes = list(zip(*batch))
```

- Specify that augmentation must be done if the augmentation object is provided. This is useful is we need to perform augmentation on training data but not on validation data:

```
        if self.aug: ims=self.aug.augment_images(images=ims)
```

In the preceding code, we leveraged the `augment_images` method so that we can work on a batch of images.

- Create tensors of images, along with scaling data, by dividing the image shape by 255:

```
        ims = torch.tensor(ims)[:,None,:,:].to(device)/255.
        classes = torch.tensor(classes).to(device)
        return ims, classes
```

In general, we leverage the `collate_fn` method when we have to perform heavy computations. This is because performing such computations on a batch of images in one go is faster than doing it one image at a time.

4. From now on, to leverage the `collate_fn` method, we'll use a new argument while creating the DataLoader:

- First, we create the `train` object:

```
train = FMNISTDataset(tr_images, tr_targets, aug=aug)
```

- Next, we define the DataLoader, along with the object's `collate_fn` method, as follows:

```
trn_dl = DataLoader(train, batch_size=64, \
                    collate_fn=train.collate_fn,shuffle=True)
```

5. Finally, we train the model, as we have been training it so far. By leveraging the `collate_fn` method, we can train a model faster.

Now that we have a solid understanding of some of the prominent data augmentation techniques we can use, including pixel translation and `collate_fn`, which allows us to augment a batch of images, let's understand how they can be applied to a batch of data to address image translation issues.

Data augmentation for image translation

Now, we are in a position to train the model with augmented data. Let's create some augmented data and train the model:

The following code is available as `Data_augmentation_with_CNN.ipynb` in the `Chapter04` folder of this book's GitHub repository - https://tinyurl.com/mcvp-packt

1. Import the relevant packages and dataset:

```
from torchvision import datasets
import torch
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

device = 'cuda' if torch.cuda.is_available() else 'cpu'
data_folder = '/content/' # This can be any directory
# you want to download FMNIST to
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                                        train=True)
tr_images = fmnist.data
tr_targets = fmnist.targets
val_fmnist=datasets.FashionMNIST(data_folder, download=True, \
```

```
                                        train=False)
val_images = val_fmnist.data
val_targets = val_fmnist.targets
```

2. Create a class that can perform data augmentation on an image that's translated randomly anywhere between -10 to +10 pixels, either to the left or to the right:

- Define the data augmentation pipeline:

```
from imgaug import augmenters as iaa
aug = iaa.Sequential([
              iaa.Affine(translate_px={'x':(-10,10)},
                                        mode='constant'),
            ])
```

- Define the `Dataset` class:

```
class FMNISTDataset(Dataset):
    def __init__(self, x, y, aug=None):
        self.x, self.y = x, y
        self.aug = aug
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x, y
    def __len__(self): return len(self.x)
    def collate_fn(self, batch):
        'logic to modify a batch of images'
        ims, classes = list(zip(*batch))
        # transform a batch of images at once
        if self.aug: ims=self.aug.augment_images(images=ims)
        ims = torch.tensor(ims)[:,None,:,:].to(device)/255.
        classes = torch.tensor(classes).to(device)
        return ims, classes
```

In the preceding code, we've leveraged the `collate_fn` method to specify that we want to perform augmentations on a batch of images.

3. Define the model architecture, as we did in the previous section:

```
from torch.optim import SGD, Adam
def get_model():
    model = nn.Sequential(
                nn.Conv2d(1, 64, kernel_size=3),
                nn.MaxPool2d(2),
                nn.ReLU(),
                nn.Conv2d(64, 128, kernel_size=3),
                nn.MaxPool2d(2),
                nn.ReLU(),
                nn.Flatten(),
                nn.Linear(3200, 256),
                nn.ReLU(),
```

```
                nn.Linear(256, 10)
            ).to(device)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=1e-3)
    return model, loss_fn, optimizer
```

4. Define the `train_batch` function in order to train on batches of data:

```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    prediction = model(x)
    batch_loss = loss_fn(prediction, y)
    batch_loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    return batch_loss.item()
```

5. Define the `get_data` function to fetch the training and validation DataLoaders:

```
def get_data():
    train = FMNISTDataset(tr_images, tr_targets, aug=aug)
    'notice the collate_fn argument'
    trn_dl = DataLoader(train, batch_size=64, \
                collate_fn=train.collate_fn, shuffle=True)
    val = FMNISTDataset(val_images, val_targets)
    val_dl = DataLoader(val, batch_size=len(val_images),
                collate_fn=val.collate_fn, shuffle=True)
    return trn_dl, val_dl
```

6. Specify the training and validation DataLoaders and fetch the model object, loss function, and optimizer:

```
trn_dl, val_dl = get_data()
model, loss_fn, optimizer = get_model()
```

7. Train the model over 5 epochs:

```
for epoch in range(5):
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        batch_loss = train_batch(x, y, model, optimizer, \
                                    loss_fn)
```

8. Test the model on a translated image, as we did in the previous section:

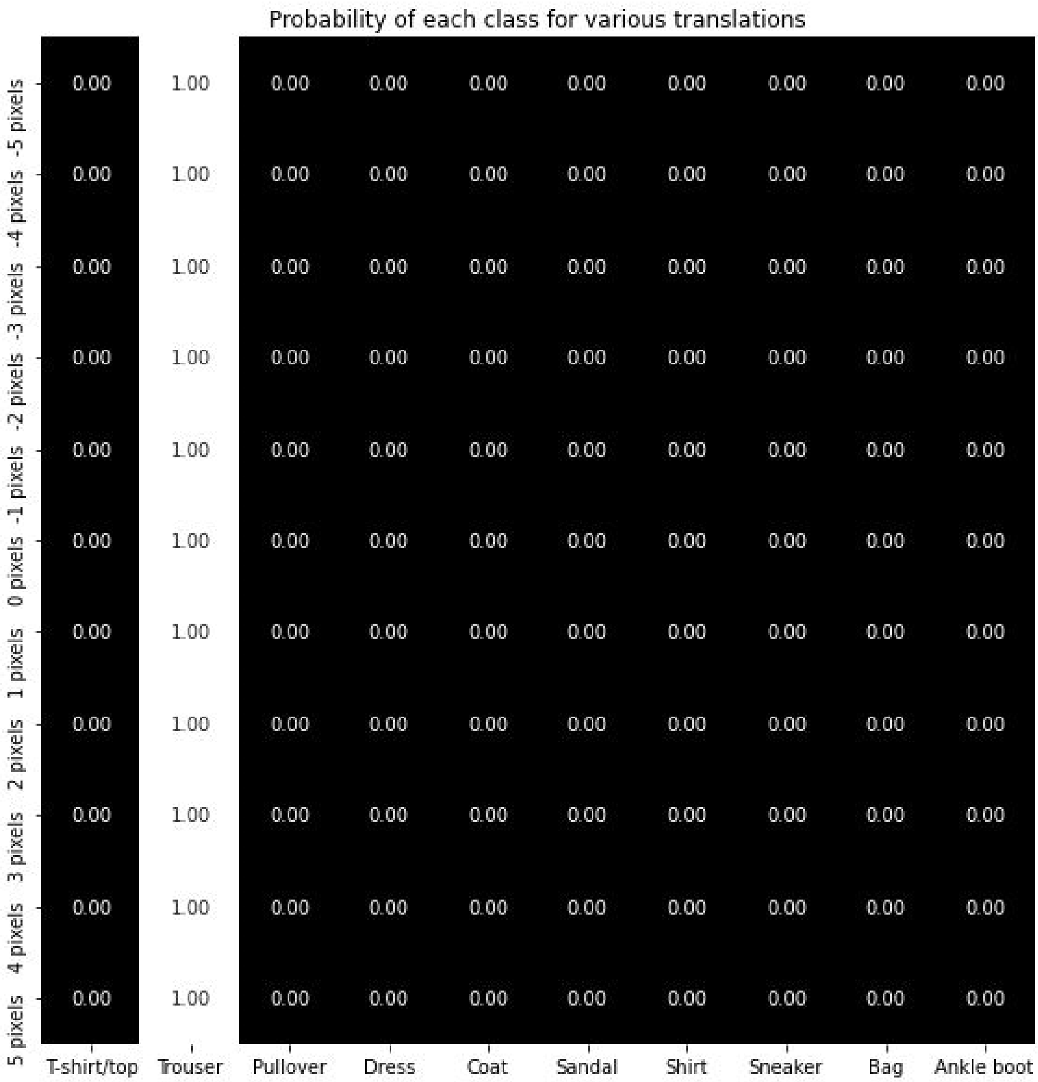
```
preds = []
ix = 24300
for px in range(-5,6):
    img = tr_images[ix]/255.
    img = img.view(28, 28)
    img2 = np.roll(img, px, axis=1)
```

```
        plt.imshow(img2)
        plt.show()
        img3 = torch.Tensor(img2).view(-1,1,28,28).to(device)
        np_output = model(img3).cpu().detach().numpy()
        preds.append(np.exp(np_output)/np.sum(np.exp(np_output)))
```

Now, let's plot the variation in the prediction class across different translations:

```
import seaborn as sns
fig, ax = plt.subplots(1,1, figsize=(12,10))
plt.title('Probability of each class \
for various translations')
sns.heatmap(np.array(preds).reshape(11,10), annot=True, \
            ax=ax, fmt='.2f', xticklabels=fmnist.classes, \
            yticklabels=[str(i)+str(' pixels') \
                         for i in range(-5,6)], cmap='gray')
```

The preceding code results in the following output:



Now, when we predict for various translations of an image, we'll see that the class prediction does not vary, thus ensuring that image translation is taken care of by training our model on augmented, translated images.

So far, we have seen how a CNN model trained with augmented images can predict well on translated images. In the next section, we'll understand what the filters learn, which makes predicting translated images possible.