CHAPTER

9

Video Action Recognition

If a picture is worth a thousand words, then a video is worth a thousand times more.

Figure 9-1. Sequence of video frames shows correlation between frames

K. Ashley, *Applied Machine Learning for Health and Fitness*,
https://doi.org/10.1007/978-1-4842-5772-2_9

Background

Our brain is a superfast action recognition system that's hard to match. In terms of deep learning, our brain routinely does many things to recognize actions, and it works *fast*! It may be years of evolution, the need to identify incoming danger or provide food: each of us has a miraculously fast video action recognition engine that just works. Our brain is capable of *normalization* and *transformation* since we recognize actions regardless of viewpoint. Human brain is great at *classification*, telling us what's moving and how, and it can also *predict* what's coming next. It turns out, our knowledge of deep learning for action recognition is getting close, but it's not as good just yet. Especially, it becomes clear when generalizing movements: the brain is far ahead of neural nets in its ability to generalize. Although if data science keeps the same pace of evolution as in the last decade, perhaps AI may get closer to the human brain.

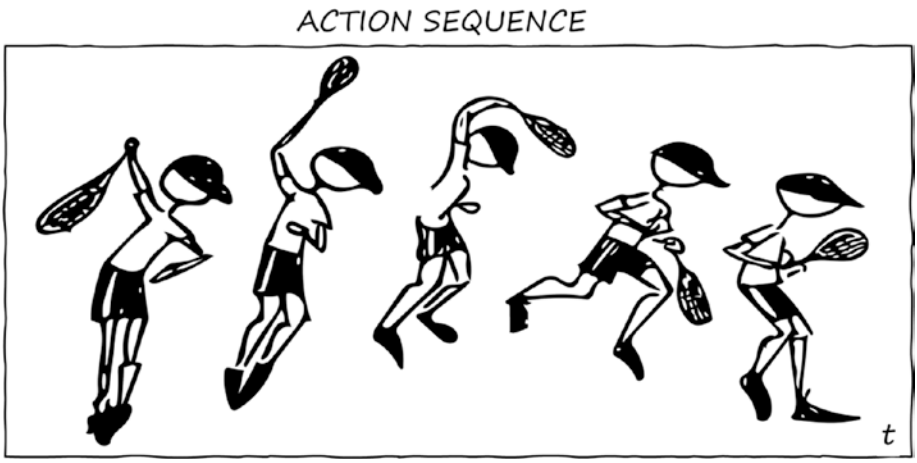

Figure 9-2. Recognizing tennis play from this sequence is easy for human brain

Recognizing actions from videos is key to many industries (Figure 9-1): sports, security, robotics, health care, and many others. For a practical sport data scientist or a coach, action recognition is part of the daily job: a coach's eye and experience is trained to do movement analysis (Figure 9-2).

In biomechanics, we use physical or classical mechanics models to describe movement. This approach worked for many years in sport science, but analytical methods are complicated and as history shows, although movements can be described with classical mechanics, deep learning methods can be more efficient and often demonstrate great precision. To illustrate this point, Kinetics dataset described in this chapter contains 400 activity classes with hundreds of sport activities recognition readily available. You can classify any of those activities with an average-level computer, and you really don't need a PhD in biomechanics.

Kinetics dataset has half a million video clips, covering hundreds of human actions and totaling close to a terabyte of data. That looks like a lot, but for each sport, it covers only a few basic moves. A human ski coach evaluating a skier can narrow it down to dozens of small movements and usually deals with multiple training routines.

Video recognition has been traditionally tough for deep learning because it needs more compute power and storage than most other types of data: that's a lot of power and storage! In the recent years, video recognition methods, models, and datasets made a significant progress to the point that they became practical for a sport scientist. As this chapter shows, these methods are also relatively easy to use with the tools, frameworks, and models available today. In this chapter, we focus on practical use and methods for video action recognition. You don't need an advanced hardware, but a GPU-enabled computer is recommended. If you don't have that handy, use an online service like Microsoft Azure or Google Colaboratory that offers free scaleable compute services for data science.

Video Data

Cinematography is a writing with images in movement and with sounds.

—Robert Bresson, *Notes on the Cinematographer*

Video classification has been an expensive task because of the need to deal with the video, and video is heavy. In this chapter you'll learn data structures for video, used across most of datasets and models for video recognition.

A single image or video frame can be represented as a 3D tensor: (width, height, color) and color depth having three channels, RGB. A sequence of frames can be represented as a 4D tensor: (frame, width, height, color). For video classification, you will typically deal with sequences, or batches of frames, and the video is represented as a 4D or 5D tensor: (sample, frame, width, height, color).

For example, to read a video into structures ready for deep learning, frameworks provide convenience methods, such as PyTorch's `torchvision.io.read_video`. In the following code snippet, the video is loaded as a 4D tensor 255 (frames) x 720 (height) x 1280 (width) x 3 (colors). Notice that this method also loads audio, although we are not going to use it for action recognition:

```
import torchvision.io
video_file = 'media/surfing_cutback.mp4'
video, audio, info = torchvision.io.read_video(video_file, pts_unit="sec")
print(video.shape, audio.shape, info)

Output:
torch.Size([255, 720, 1280, 3]) torch.Size([2, 407552]) {'video_fps':
29.97002997002997, 'audio_fps': 48000}
```

This original video at 720p resolution is large; most models are trained with images and videos that are much smaller in size and were normalized.

Datasets

> *The relationship between space and time is a mysterious one.*
>
> —Carreira, Zimmerman "Quo Vadis? Action Recognition"

Quo Vadis is Latin meaning "where are you going?" Although action recognition is achievable from a still frame, it works best when learning from temporal component as well as spatial information.

From the still frame in Figure 9-3, it's not easy to tell whether the person is swimming or running. Perhaps, this ambiguity in action recognition prompted authors of research article on Kinetics video dataset and the model they developed to choose the title for the research work. Prior to Kinetics, Sports-1M used to be a breakthrough dataset used in many models, and in authors' own words:

> *To obtain sufficient amount of data needed to train our CNN architectures, we collected a new Sports-1M dataset, which consists of 1 million YouTube videos belonging to a taxonomy of 487 classes of sports.*
>
> —Andrej Karpathy et al., "Large-Scale Video Classification with Convolutional Neural Networks"

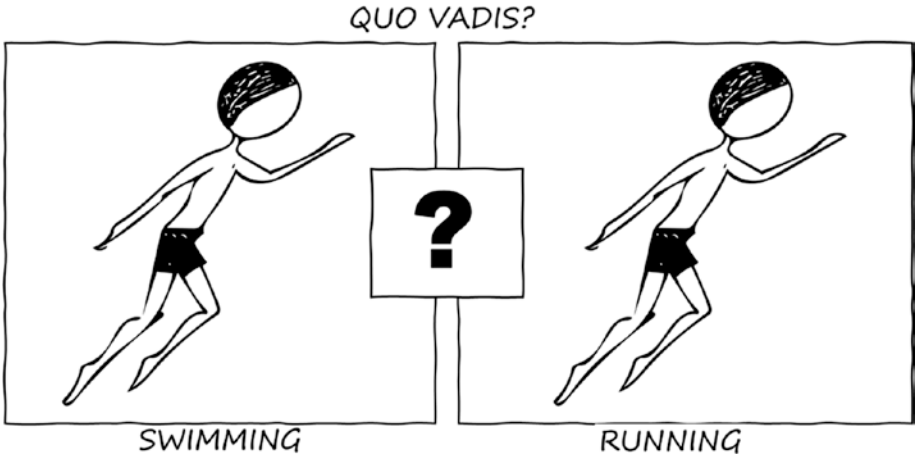

Figure 9-3. Quo Vadis? Why is the article on Kinetics dataset and ConvNets named after 1951 epic?

Historically, deep learning for video recognition focused on activities that were easily available. What source of video data can a data scientist use without the need to store terabytes of videos? YouTube comes very handy, as well as any other online video service! You'll notice that many of the action recognition datasets use online video services because those videos are typically indexed, can be readily retrieved, and often have additional metadata that helps classifying entire videos or even segments. In fact, with massive online video repositories, storing billions of movements and deep learning, we are on the verge of revolution in movement recognition!

Some well-known datasets for human video action sequences include:

- **HMDB 51** is a set of 51 action categories, including facial and body movements and human interaction. This dataset contains some sport activities, but is limited to bike, fencing, baseball, and a few others. This is included in PyTorch: `torchvision.datasets.HMDB51`.
- **UCF 101** is used in many action recognition scenarios, including human-object interaction, body motion, playing musical instruments, and sports. This is included in PyTorch: `torchvision.datasets.UCF101`.
- **Kinetics** is a large dataset of URL links to video clips that covers human action classes, including sports, human interaction, and so on. The dataset is available in different sizes: Kinetics 400, 600, and 700 and is included in PyTorch: `torchvision.datasets.Kinetics400`.

Models

While video presents many challenges, such as computational cost and capturing both spatial and temporal action over long periods of time, it also presents unique opportunities in terms of designing data models. Over the last few years, researchers experimented with various approaches to video action recognition modeling. Methods that prove most effective so far are using pretrained networks, fusing various *streams* of data from video, for example, motion stream from optical flow and spatial pretrained context (Figure 9-4).

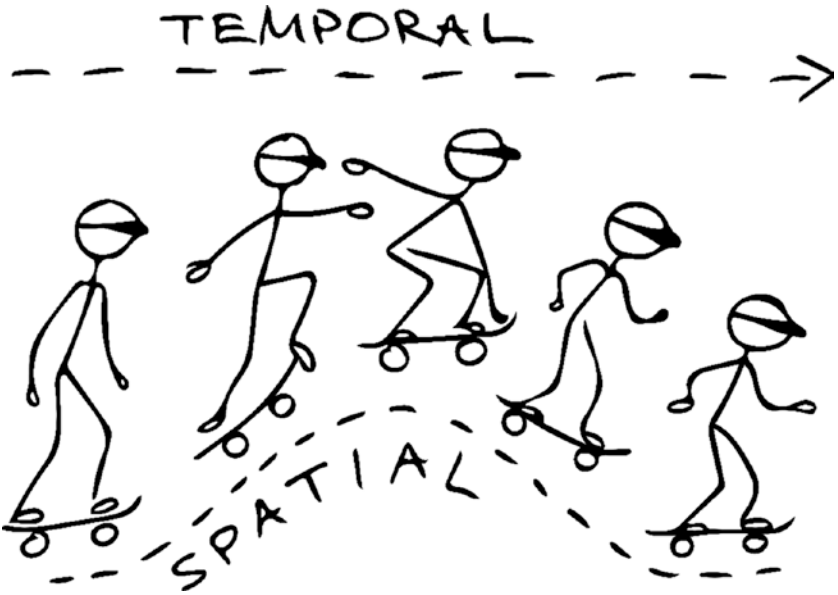

Figure 9-4. Modern models use fusion of context streams: for example, temporal and spatial for action recognition

Some earlier methods tried experimenting and benchmarking model performance with various context streams, for example, using different context resolutions with a technique authors creatively called a "fovea" stream in one research combined with the main feature learning stream.

> *Fovea – a small depression in the retina of the eye where visual acuity is the highest.*
>
> —Oxford Dictionary

This area of deep learning is still under active research and we may still see state-of-the-art models that outperform existing methods.

Video Classification QuickStart

PROJECT 9-1: QUICK START ACTION RECOGNITION

This project provides a quick start for video classification: the goal is to have a practical sport data scientist quickly started on the human activity recognition. Before we start on this project, let's take a look at the list of human activities we can classify with minimal effort. I'll be using PyTorch here, because it provides video classification datasets and pretrained models out of the box. PyTorch computer vision module, torchvision, contains many models and datasets we can use in sports data science, including classification, semantic segmentation, object detection, person keypoint detection, and video classification. Video classification models and datasets included with PyTorch are what we'll be using for this task to get started quickly.

In PyTorch video classification models are trained with Kinetics 400 dataset. Although not all of these human activities are sports related, in the source code I put together a helper in `utils.kinetics`; conveniently, it provides a list of sport-related activities:

```
from utils.kinetics import kinetics
categories = kinetics.categories()
classes = kinetics.classes()
sports = kinetics.sport_categories()

count = 0
for key in categories.keys():
    if key in sports:
        print(key)
        for label in categories[key]:
            count+=1
            print("\t{}".format(label))
print(f'Sport activities labels: {count}')

Output:
Sport activities labels: 134
```

■ **Note for activity granularity** The trend with activity recognition is getting even more granular. For example, in golf, Kinetics 400 classifies chipping, driving, and putting and, for swimming, backstroke, breast stroke, butterfly, and so on.

So, from several video classification models available in torchvision pretrained on Kinetics 400 dataset, we should be able to get 130+ sports-related actions classified. To quick start our development, we will jump start action recognition with pretrained models available in PyTorch torchvision. Let's start by importing the required modules:

```
import torch
import torchvision
import torchvision.models as models
```

Since we are dealing with models that involve video tensors, having a CUDA-enabled installation of PyTorch really helps. Processing on a CPU will work, but may take a very long time. To make sure we are using a CUDA device, run this command:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

Output:
cuda
```

Then, get an appropriate model trained on Kinetics 400. Currently, PyTorch supports three video classification models out of the box: ResNet 3D, ResNet Mixed Convolution, and ResNet (2+1)D. I instantiated ResNet 3D (r3d_18) and commented out two other models. The important thing of course is `pretrained=True` flag that saves us downloading all the videos for Kinetics 400 dataset to train the model!

```
model = models.video.r3d_18(pretrained=True)
#model = models.video.mc3_18(pretrained=True)
#model = models.video.r2plus1d_18(pretrained=True)
model.eval()
```

Thanks to PyTorch magic, the pretrained model gets downloaded automatically, a huge time saver! Training on Kinetics 400 dataset requires a massive number of videos downloaded. So, having a pretrained video classification model in PyTorch is a great starter for a practical sport data scientist.

■ **Practical tip** Having a pretrained model for Kinetics in PyTorch torchvision offers a big advantage from the practical standpoint. Downloading datasets and videos to train video classification models takes a lot of space and compute power!

Next, we need to define our normalization for the video. For Kinetics, height and width are normalized to 112 pixels, the mean to [0.43216, 0.394666, 0.37645], and standard deviation to [0.22803, 0.22145, 0.216989], parameters recommended in the pretrained model. In the code snippet that follows, this normalization is implemented

in several methods that we will later apply in sequence to convert the video to PyTorch tensor, resize and crop it, and normalize to the mean and standard deviation, so that it matches the model:

```
# Normalization for Kinetics datasets

mean = [0.43216, 0.394666, 0.37645]
std = [0.22803, 0.22145, 0.216989]

# To normalized float tensor
def normalize(video):
    return video.permute(3, 0, 1, 2).to(torch.float32) / 255

# Resize the video
def resize(video, size):
    return torch.nn.functional.interpolate(video, size=size, scale_
    factor=None, mode="bilinear", align_corners=False)

# Crop the video
def crop(video, output_size):
    h, w = video.shape[-2:]
    th, tw = output_size
    i = int(round((h - th) / 2.))
    j = int(round((w - tw) / 2.))
    return video[..., i:(i + th), j:(j + tw)]

# Normalize using mean and standard deviation
def normalize_base(video, mean, std):
    shape = (-1,) + (1,) * (video.dim() - 1)
    mean = torch.as_tensor(mean).reshape(shape)
    std = torch.as_tensor(std).reshape(shape)
    return (video - mean) / std
```

We will use torchvision.io method to read a video file and show shape and some other useful information about the video we use as a source:

```
import torchvision.io
video_file = 'media/surfing_cutback.mp4'
video, audio, info = torchvision.io.read_video(video_file)
shape = video.shape
print(f'frames {shape[0]}, size {shape[1]} {shape[2]}\n{info}')

Output:
frames 255, size 720 1280 {'video_fps': 29.97002997002997, 'audio_fps':
48000}
```

As you can see, the original video has 255 frames and 720p resolution, which is relatively large for our model. As discussed earlier, we need to normalize the video before submitting it to the model; this is the time to call the normalizing methods we defined earlier:

```
video = normalize(video)
video = resize(video,(128, 171))
video = crop(video,(112, 112))
video = normalize_base(video, mean=mean, std=std)
shape = video.shape
print(f'frames {shape[0]}, size {shape[1]} {shape[2]}')

Output:
frames 3, size 255 112
```

Much better! After normalization, the video is much smaller, and you can experiment with the number of frames (authors of the original model used in PyTorch mention various clip lengths of 8, 16, 32, 40, and 48 frame clips; in our experiment we only use 3 frames for inference). If you have a GPU-enabled device with CUDA, you can accelerate the process by moving both model and video tensor to the CUDA-enabled device:

```
# Make use of accelerated CUDA if available
if torch.cuda.is_available():
    model.cuda()
    video = video.cuda()
```

Now comes the magic of applying our pretrained model and making it process an actual surfer video. Note that in the first line, the model expects a tensor with one extra dimension, so in the call to the model, we apply `unsqueeze` method to the video to satisfy the model's requirements. The result is an array of scored classes (activities). In the second line, we use `argmax()` function to select the best matching value and assign it to prediction. We then print the best score, which is the number of the class in the list of activities of Kinetics dataset. This may take some time if you have a CPU only, depending on your environment, so be patient:

```
# Score the video (takes some time!)
score = model(video.unsqueeze(0))
# Get prediction with max score
prediction = score.argmax()
print(prediction)

Output:
tensor(337)
```

The resulting index is not very meaningful, so let's convert it to the actual class name it represents, by using our utility script `kinetics.classes()`. That returns a list of classes in the Kinetics dataset which makes it very easy to map the result of the prediction to the name of the activity `classes[prediction.item()]`:

```
from utils.kinetics import kinetics
classes = kinetics.classes()
print(classes[prediction.item()])

Output:
surfing water
```

And it turns out to be "surfing water", the class Kinetics model was trained with to detect a surfing action. Our predicted result is correct! In this example we used a custom video file from a consumer grade 720p resolution video camera. We used a PyTorch pretrained model trained on Kinetics dataset for video classification of 400 activities, of which more than a hundred are sports related. We normalized the video and classified an activity correctly on the video, by returning the best score (score.argmax) and mapping the numerical result to the index in the list of activities in Kinetics dataset.

Loading Videos for Classification

PyTorch includes a number of modules simplifying video classification. In the previous project, you already explored an introduction to video classification, based on a pretrained models, included in torchvision. We also used `torchvision.io.read_video` method to load videos in a convenient structure of tensors that include video frames, audio, and relevant video information. In the following project, we'll take it further and will do some practical video loading and model training, as well as transfer learning for video classification.

PROJECT 9-2: LOADING VIDEOS FOR CLASSIFIER TRAINING

In this project I'll show you how to use video dataset modules, such as Kinetics400 and DataLoader to visualize videos and prepare them for training. Kinetics folder structure follows a common convention that includes train/test/validation folders and videos split into classes of actions we need to recognize. Since datasets, such as Kinetics, are based on indexed online videos, there're many scripts out there that simplify loading videos for training and structuring them in folders. For now, we'll define the base directory of our dataset:

```
base_dir = Path('data/kinetics400/')
data_dir = base_dir/'dataset'
```

Conveniently, as part of `torchvision.datasets`, PyTorch includes `Kinetics400` dataset that serves as a cookie cutter for our project. Internally, video datasets use `VideoClips` object to store video clips data:

```
data = torchvision.datasets.Kinetics400(
            data_dir/'train',
            frames_per_clip=32,
            step_between_clips=1,
            frame_rate=None,
            extensions=('mp4',),
            num_workers=0
        )
```

■ **Note** Although you can and should take advantage of the multiprocessing nature of datasets, especially in the production environment, on some systems you may get an error; num_workers = 0 makes sure you use dataset single threaded.

According to this constructor earlier, each video clip loaded with our dataset should be a 4D tensor with the shape (frames, height, width, channels); in our case 32 frames, RGB video, note that Kinetics doesn't require all clips to be of the same height/width:

```
print((data[0][0]).shape)
Output:
torch.Size([32, 226, 400, 3])
```

Visualizing Dataset

Sometimes, it may be handy to visualize the entire dataset catalog as a table, summarizing the number of frames. The helper function to_dataframe loads the entire video catalog into Pandas DataFrame and displays the content:

```
from utils.video_classification.helpers import to_dataframe
to_dataframe(data)
```

Let's say we want to display the size of a video in the dataset:

```
VIDEO_NUMBER = 130
video_table = to_dataframe(data)
video_info = video_table['filepath'][VIDEO_NUMBER]
```

With notebook IPython.display video helper, we can also show the video embedded in the notebook (Figure 9-5), but keep in mind that setting `embed=True` while displaying the video may significantly increase the size of your notebook:

```
from IPython.display import Video
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

Video(video_info, width=400, embed=False)
```

[29]:

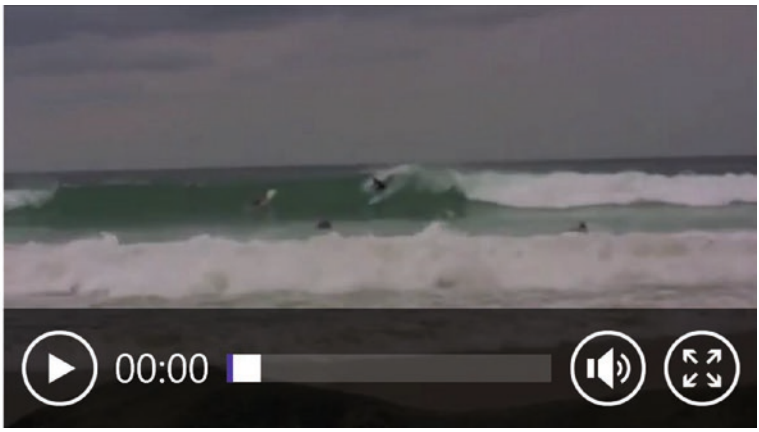

Figure 9-5. Displaying embedded video in notebooks

So instead of embedding the video, it may be sufficient to just visualize the first and last frames (Figure 9-6):

```
def show_clip_start_end(f):
    last = len(f)
    plt.imshow(f[0])
    plt.title(f'frame: 1')
    plt.axis('off')
    plt.show()
    plt.imshow(f[last-1])
    plt.title(f'frame: {last}')
    plt.axis('off')
    plt.show()

show_clip_start_end(data[0][0])
```

Figure 9-6. Loading and visualizing video frames

Video Normalization

As with most of the data, before training our model, video needs to be normalized for video classification models included in torchvision. This involves getting image data in the range [0,1] and normalizing with standard deviation and the mean provided with the model:

```
t = torchvision.transforms.Compose([
        T.ToFloatTensorInZeroOne(),
        T.Resize((128, 171)),
        T.RandomHorizontalFlip(),
        T.Normalize(mean=[0.43216, 0.394666, 0.37645],
                            std=[0.22803, 0.22145, 0.216989]),
        T.RandomCrop((112, 112))
    ])
```

This transformation uses Compose method from PyTorch, which combines several transformations including converting the video to a float tensor, resize, cropping, and normalizing with a mean and standard deviation. Note that in this transformation we also apply an augmentation function `T.RandomHorizontalFlip()` which provides a random horizontal flip to a frame. Once we've defined the transform, we can pass it to the Kinetics400 dataset:

```
train_data = torchvision.datasets.Kinetics400(
            data_dir/'train',
            frames_per_clip=32,
            step_between_clips=1,
            frame_rate=None,
            transform=t,
            extensions=('mp4',),
            num_workers=0
        )
```

DataLoader class in PyTorch provides many useful features and makes it easy to use from Python, including iterable datasets, automatic batching, memory pinning, sampling, and data loading order customization. Learning rate, as a hyperparameter for training neural networks, is important: if you make learning rate too small, the model will likely converge too slowly.

Mysterious constant The so-called Karpathy constant defines the best learning rate for Adam as 3e-4. The author of the famous tweet in data science, Andrej himself in the response to his own tweet, says that this was a joke. Nevertheless, the constant made it to Urban Dictionary and many data science blogs.

As an illustration (Figure 9-7), notice that by making learning rate too large for gradient descent, the model will never reach its minimum.

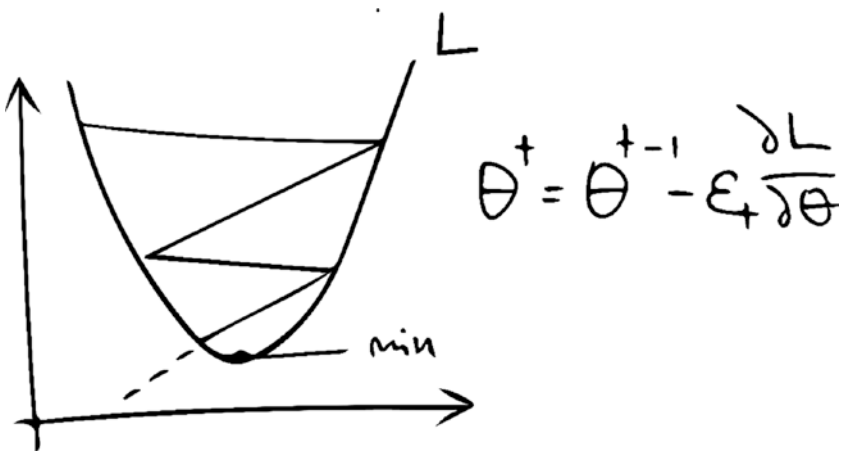

Figure 9-7. Large learning rate may miss the gradient descent minimum and the model will fail to converge. L is the loss function and εt is the learning rate and θ is weights (parameters)

To deal with this problem, many frameworks, including fastai and PyTorch, now include learning rate finder module.

In case of video action recognition, with the size of the data and large differences in training times for video data, it is recommended to use a proper learning rate, which is often in the middle of the descending loss curve.

Training the Model

Training the model for video action recognition in PyTorch follows the same principles as for image classifier, but since video classification functionality is relatively new in PyTorch, it's worth including a small example in this chapter.

PROJECT 9-3: VIDEO RECOGNITION MODEL TRAINING

To start, let's create two datasets, for training and validation, based on built-in Kinetics object. The idea here is to take advantage of built-in objects that PyTorch offers for simplicity. I use the same normalizing video transformation T, already used in previous examples. On some systems you can get a significant speed improvement if you set num_workers > 0, but on my system I had to be conservative, so I keep it at zero (basically, it means don't take advantage of parallelization):

```
train_data = torchvision.datasets.Kinetics400(
            data_dir/'train',
            frames_per_clip=32,
            step_between_clips=1,
```

```
            frame_rate=None,
            transform=t,
            extensions=('mp4',),
            num_workers=0
        )

valid_data = torchvision.datasets.Kinetics400(
            data_dir/'valid',
            frames_per_clip=32,
            step_between_clips=1,
            frame_rate=None,
            transform=t,
            extensions=('mp4',),
            num_workers=0
        )
```

PyTorch allows using familiar DataLoaders with video data, and for video data PyTorch includes `VideoClips` class used for enumerating clips in the video and also sampling clips in the video while loading. `FirstClipSampler` in the following example used `video_clips` property from the dataset to sample a specified number of clips in the video:

```
train_sampler = FirstClipSampler(train_data.video_clips, 2)
train_dl = DataLoader(train_data,
                      batch_size=4,
                      sampler=train_sampler,
                      collate_fn=collate_fn,
                      pin_memory=True)
valid_sampler = FirstClipSampler(valid_data.video_clips, 2)
valid_dl = DataLoader(valid_data,
                      batch_size=4,
                      sampler=valid_sampler,
                      collate_fn=collate_fn, pin_memory=True)
```

Loading and renormalizing video data can take a really long time, so you may want to save the normalized dataset in cache directory:

```
import os
cache_dir = data_dir/'.cache'
if not os.path.exists(cache_dir):
    os.makedirs(cache_dir)
cache_dir.ls()

torch.save(train_data, f'{cache_dir}/train')
torch.save(valid_data, f'{cache_dir}/valid')
```

Next, you initialize the model with hyperparameters, including the learning rate obtained earlier. Note that since we'll be training the model, we instantiate it without weights (`pretrained=False` or omitted):

```
model = models.video.r2plus1d_18()
```

Next, we can train the model; in the following example, I chose 10 epochs:

```
for epoch in range(10):
    train_one_epoch(model,
                    criterion,
                    optim,
                    lr_scheduler,
                    train_dl, device,
                    epoch, print_freq=100)
    evaluate(model,
             criterion,
             valid_dl,
             device)
```

You can also save the model weights once it's trained:

```
SAVED_MODEL_PATH = './videoresnet_action.pth'
torch.save(model.state_dict(), SAVED_MODEL_PATH)
```

In the previous sections, we used a pretrained model; in this example, we trained a model and saved the weights. This resulting model can now be loaded and used to predict actions from a video. As a data scientist working with video recognition, you may need this to improve accuracy, or add new activities. Kinect is a large dataset, but it doesn't cover everything, so you may need to retrain your models based on activities you need to classify.

Summary

In this chapter we covered practical methods and tools for video action recognition and classification. We discussed data structures for loading, normalizing, and storing videos; datasets for sport action classification, such as Kinetics; and deep learning models. Using readily available pretrained models, we can classify hundreds of sport actions and train the models to recognize new activities. For a sport data scientist, this chapter provides practical examples for deep learning, movement analysis, and action recognition on any video.

Although video action recognition is becoming more usable today, and made progress in thousands of classifications, we are still far from the goals of generalized action recognition. That means, as a sport data scientist, you are still left with a lot of work to apply video recognition in the field. Is this the right time to make video action recognition a part of your toolbox? With practical examples and notebooks accompanying this chapter, I think that this is the right time for coaches and sport scientists to start using these methods in everyday sport data science. Video action recognition requires a lot of compute power, so you may want to consider referring to chapters of this book that describe using the cloud and automating model training.