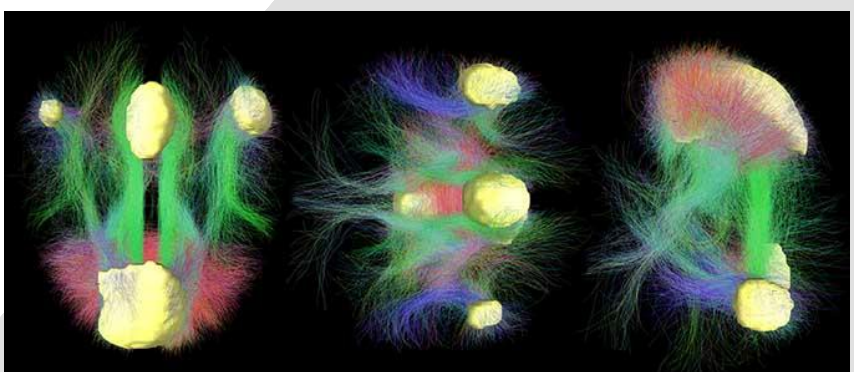
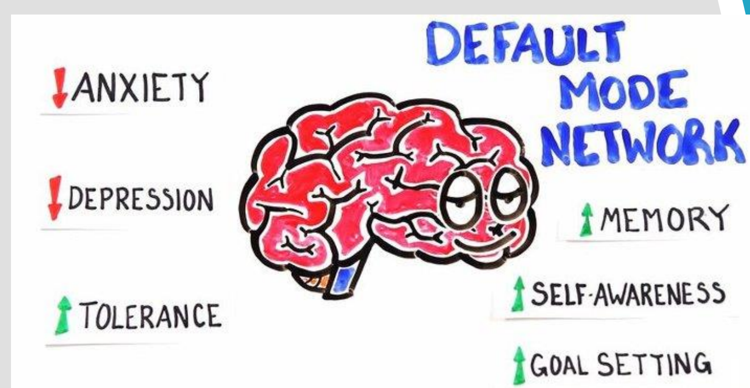
6장 *Deep learning for text and sequences*

"default mode network"





6.3 Advanced use of recurrent neural networks

- three advanced techniques for improving the performance and generalization power of RNNs

- temperature-forecasting problem - timeseries of data points coming from sensors-temperature, air pressure, and humidity-to predict the temperature 24 hours after the last data point

- We'll cover the following techniques:
  - Recurrent dropout —to fight overfitting in recurrent layers
  - Stacking recurrent layers—increases the representational power of the network (at the cost of higher computational loads).
  - Bidirectional recurrent layers—same information to a recurrent network in different ways, increasing accuracy and mitigating forgetting issues.

6.3 Advanced use of recurrent neural networks

6.3.1 A concrete LSTM example in Keras

- Dataset - weather timeseries dataset recorded at the Weather Station at the Max Planck Institute for Biogeochemistry in Jena, Germany
- 14 different quantities (such air temperature, atmospheric pressure, humidity, wind direction, and so on) - recorded every 10 minutes, over several years from 2009–2016
- Input some data from the recent past (a few days' worth of data points) and predicts the air temperature 24 hours in the future

- Download and uncompress the data as follows:

```
cd ~/Downloads
mkdir jena_climate
cd jena_climate
wget https://s3.amazonaws.com/keras-
       datasets/jena_climate_2009_2016.csv.zip
unzip jena_climate_2009_2016.csv.zip
```

6.3 Advanced use of recurrent neural networks

- Dataset - jena_climate_2009_2016.csv
- Stored in –

\deep-learning-with-python-notebooks-master\datasets\jena_climate

Date Time	p (mbar)	T (degC)	Tpot (K)	Tdew (deg	rh (%)	VPmax (m	VPact (mb	VPdef (mb	sh (g/kg)	H2OC (mr	rho (g/m*	wv (m/s)	max. wv (r	wd (deg)
01.01.2009	996.52	-8.02	265.4	-8.9	93.3	3.33	3.11	0.22	1.94	3.12	1307.75	1.03	1.75	152.3
01.01.2009	996.57	-8.41	265.01	-9.28	93.4	3.23	3.02	0.21	1.89	3.03	1309.8	0.72	1.5	136.1
01.01.2009	996.53	-8.51	264.91	-9.31	93.9	3.21	3.01	0.2	1.88	3.02	1310.24	0.19	0.63	171.6
01.01.2009	996.51	-8.31	265.12	-9.07	94.2	3.26	3.07	0.19	1.92	3.08	1309.19	0.34	0.5	198
01.01.2009	996.51	-8.27	265.15	-9.04	94.1	3.27	3.08	0.19	1.92	3.09	1309	0.32	0.63	214.3

6.3 Advanced use of recurrent neural networks

Listing 6.28 Inspecting the data of the Jena weather dataset

```
import os

data_dir = '/users/fchollet/Downloads/jena_climate'
fname = os.path.join(data_dir,
                     'jena_climate_2009_2016.csv')

f = open(fname)
data = f.read()
f.close()

lines = data.split('\n')
header = lines[0].split(',')
lines = lines[1:] # except header line

print(header)
print(len(lines)) # 420,551
```

6.3 Advanced use of recurrent neural networks

- This outputs a count of 420,551 lines of data (each line is a timestep: a record of a date and 14 weather-related values), as well as the following header:
- Header

["Date Time",
"p (mbar)",
"T (degC)",
"Tpot (K)",
"Tdew (degC)",
"rh (%)",
"VPmax (mbar)",
"VPact (mbar)",
"VPdef (mbar)",
"sh (g/kg)",
"H2OC (mmol/mol)",
"rho (g/m**3)",
"wv (m/s)",
"max. wv (m/s)",
"wd (deg)"]

- 14 weather-related values

```
01.01.2009 00:10:00, 996.52, -8.02, 265.40, -8.90, 93.30, 3.33, 3.11, 0.22, 1.94, 3.12, 1307.75, 1.03, 1.75, 152.30
```

6.3 Advanced use of recurrent neural networks

- Now, convert all 420,551 lines of data into a Numpy array.

Listing 6.29 Parsing the data

```
import numpy as np

float_data = np.zeros((len(lines), len(header) - 1)) # 420551,14
for i, line in enumerate(lines): # except Date Time
   values = [float(x) for x in line.split(',')[1:]]
   float_data[i, :] = values

print(float_data[0])

[ 9.96520e+02 -8.02000e+00  2.65400e+02 -8.90000e+00  9.33000e+01
  3.33000e+00  3.11000e+00  2.20000e-01  1.94000e+00  3.12000e+00
  1.30775e+03  1.03000e+00  1.75000e+00  1.52300e+02]
```

6.3 Advanced use of recurrent neural networks

- plot of temperature (in degrees Celsius) over time (see figure 6.18)
- yearly periodicity of temperature

Listing 6.30 Plotting the temperature timeseries

```
from matplotlib import pyplot as plt
temp = float_data[:, 1] # temperature (in degrees Celsius)
plt.plot(range(len(temp)), temp)
```

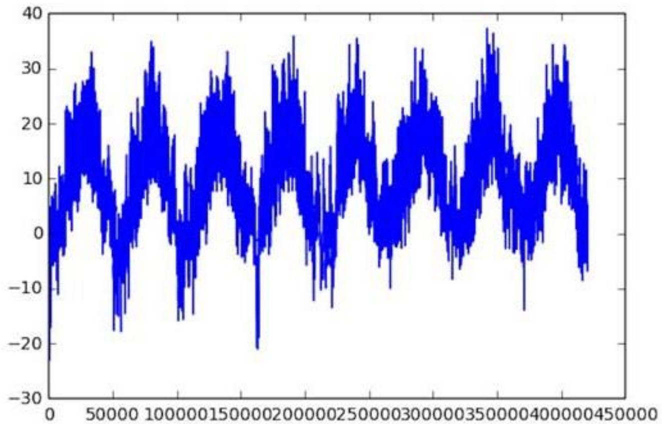


Figure 6.18 Temperature over the full temporal range of the dataset (°C)

6.3 Advanced use of recurrent neural networks

- the first 10 days of temperature data (see figure 6.19) -1440 data
- 144 data points per day - recorded every 10 minutes

Listing 6.31 Plotting the first 10 days of the temperature timeseries

```
plt.plot(range(1440), temp[:1440])
```

- daily periodicity for the last 4 days from a fairly cold winter month
- Is this timeseries predictable at a daily scale? Let’s find out.

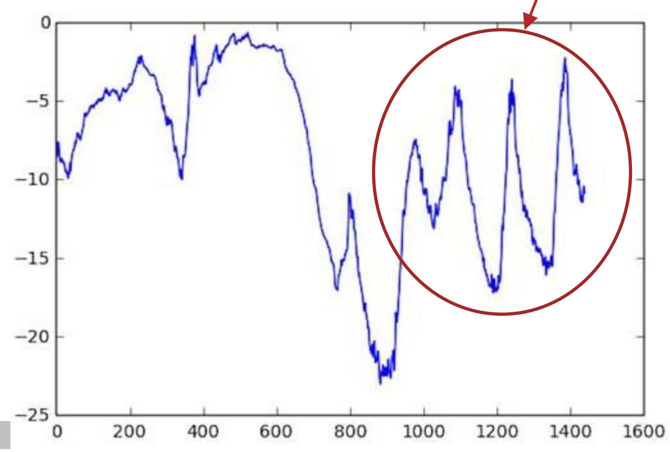


Figure 6.19 Temperature over the first 10 days of the dataset (°C)

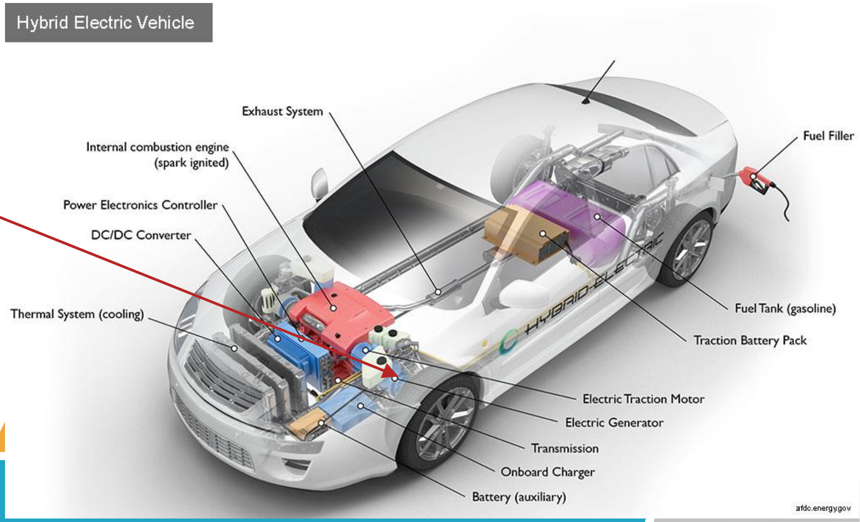
6.3 Advanced use of recurrent neural networks

6.3.2 Preparing the data

- `lookback` timesteps (10 minutes), `steps` timesteps, and `delay` timesteps
  - `lookback = 1440`—Observations will go back 10 days.
  - `steps = 6`—sampled at one data point per hour.
  - `delay = 144`—Targets will be 24 hours in the future.
- To get started, you need to do two things:
  - Preprocess of normalization in small values on a similar scale - temperature between -20 and +30, atmospheric pressurearound 1,000 in mbar
  - Write a Python generator

너무너무 중요하다고 언급하심

이해를 쉽게하기 위한 자동차 그림
-> 제너레이터는 발전기다. 발전한 전기를 배터리에 저장
발전기가 없으면 차가 작동하지 않음



6.3 Advanced use of recurrent neural networks

- Normalizing the data - subtracting the mean of each timeseries and dividing by the standard deviation.

- Use the first 200,000 timesteps as training data.

20만개, 10만개, 12만개를 각각
train, val, test dataset으로 사용

Listing 6.32 Normalizing the data (Z-Distribution)

```
mean =float_data[:200000].mean(axis=0)
float_data -= mean
std = float_data[:200000].std(axis=0)
float_data /= std
```

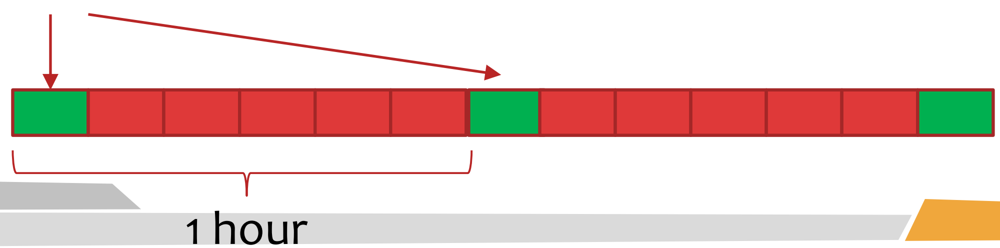
(세 번째 언급한다고 하신 부분)
axis=0이라는 것
표가 있으면 axis 0은 세로로 평균을 내는 것
1은 가로로 평균내고, 3차원이면 z축으로

6.3 Advanced use of recurrent neural networks

- Listing 6.33 - data generator
- yields a tuple (`samples`, `targets`) - `samples` : one batch of input data, `targets` : the corresponding array of target temperatures

제너레이터는 yield[일드]를 한다. 양보라는 뜻보다는 수확한다라는 뜻
samples와 targets를 return하는데, return한다고 메소드가 죽는게 아니고 쉰다(?). 살아있다.

- It takes the following arguments:
  - `data`—The original normalized array of floating-point data
  - `lookback`—How many timesteps back the input data should go.
  - `delay`—How many timesteps in the future the target should be.
  - `min_index` and `max_index`—data for validation and testing
  - `shuffle`—Whether to shuffle the samples or draw them in chronological order, specially for train data
  - `batch_size`—The number of samples per batch.
  - `step`—6 in order to draw one data point every hour.



RNN보다 이 제너레이터가 더 어려움
골치아파서 시험보기 참 좋다 라고 언급하심

6.3 Advanced use of recurrent neural networks

Listing 6.33 Generator yielding timeseries samples and their targets (yields 128 batch samples & targets)

```
def generator(data, lookback, delay, min_index, max_index,
      shuffle=False, batch_size=128, step=6):
#lookback=1440,delay=144,min_index=0,200001,300001,max_index=200000,300000,None
   if max_index is None:
      max_index = len(data) - delay - 1
   i = min_index + lookback  # lookback = 1440 (10 days)
   while 1: # true
      if shuffle: # for training - no seasonal period
         rows = np.random.randint(  # [82519  4579 ... 174730] (128,)
            min_index + lookback, max_index, size=batch_size) #low,high,batch_size개
      else:
         if i + batch_size >= max_index:
            i = min_index + lookback
         rows = np.arange(i, min(i + batch_size, max_index)) # batch_size개
                # [1440 1441 ...  1567] (128,)
         i += len(rows) # i += 128 when next call
      samples = np.zeros((len(rows), lookback//step, data.shape[-1])) # (128,240,14)
      targets = np.zeros((len(rows),)) # [-1.31  ...  -2.03] (128,)
      for j, row in enumerate(rows):
         indices = range(rows[j] - lookback, rows[j], step)
                  # [0, 6, 12, ... , 1434] (240,)
         samples[j] = data[indices] # [[[0.4296 ...]...]...] (128,240,14)
         targets[j] = data[rows[j] + delay][1] # T(degC), (128,)
      yield samples, targets # wait
```

6.3 Advanced use of recurrent neural networks

- `generator` function
  - training generator - the first 200,000 time-steps
  - validation generator - the following 100,000
  - test generator - the remainder (300,001 ~ end)

Listing 6.34 Preparing the training, validation, and test generators

```
lookback = 1440
step = 6
delay = 144
batch_size = 128
train_gen = generator(float_data,lookback=lookback, delay=delay, min_index=0,
          max_index=200000,shuffle=True, step=step, batch_size=batch_size)
val_gen=generator(float_data,lookback=lookback,delay=delay,min_index=200001,
          max_index=300000, step=step, batch_size=batch_size)
test_gen=generator(float_data,lookback=lookback,delay=delay,min_index=300001,
          max_index=None, step=step, batch_size=batch_size)
val_steps = (300000 - 200001 - lookback) // batch_size
# 769 steps to draw from val_gen in order to see the entire validation set

test_steps = (len(float_data) - 300001 - lookback) // batch_size
# 930 steps to draw from test_gen in order to see the entire test set
```

6.3.3 A common-sense, non-machine-learning baseline

- common-sense base-lines - unbalanced classification tasks, where some classes are much more common than others.
- 90% instances of class A and 10% instances of class B - 90% accurate overall
- the temperature timeseries - tomorrow are likely to be close to the temperatures today as well as periodical with a daily period.
- the mean absolute error (MAE) metric:

```
np.mean(np.abs(preds - targets))
```

6.3 Advanced use of recurrent neural networks

- Here’s the evaluation loop.

Listing 6.35 Computing the common-sense baseline MAE

```
def evaluate_naive_method():
  batch_maes = []
  for step in range(val_steps): #930 steps
     samples, targets=next(val_gen) # 128
     preds = samples[:, -1, 1] # 128
     mae = np.mean(np.abs(preds-targets))
     batch_maes.append(mae)
  print(np.mean(batch_maes))

evaluate_naive_method()
```

- MAE of 0.29 - normalized to be centered on 0 and standard deviation of 1
- MAE of 0.29 $\times$ `temperature_std` = 2.57˚C.

Listing 6.36 Converting the MAE back to a Celsius error

```
celsius_mae = 0.29 * std[1] # T(degC),
# std = float_data[:200000].std(axis=0)
```

- Now the game is to use your knowledge of deep learning to do better.

6.3.4 A basic machine-learning approach

RNN하기전에 비교를 위해서 신경망으로 돌려봄

- try simple, cheap machine-learning models (such as small, densely connected networks) before looking into complicated and computationally expensive models such as RNNs.
- fully connected model - starts by flattening the data and then runs it through two `Dense` layers
- No activation function on the last `Dense` layer - typical for a regression problem.
- You use MAE as the loss.

6.3 Advanced use of recurrent neural networks

Listing 6.37 Training and evaluating a densely connected model

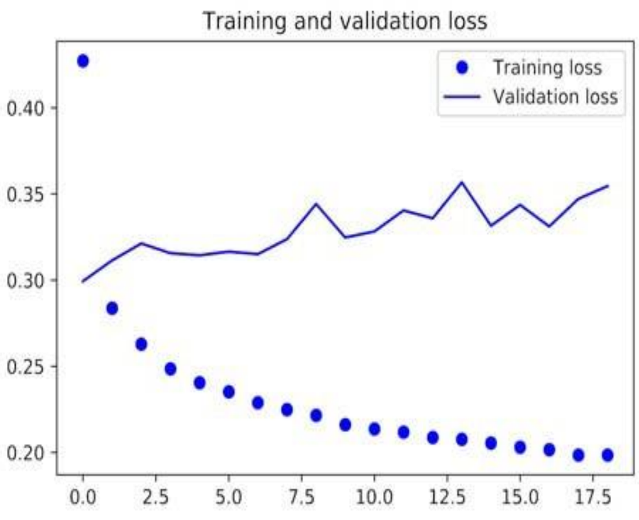
```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Flatten(input_shape=(lookback // step,
    float_data.shape[-1]))) # 240*14
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
    steps_per_epoch=500,#200000/128=1562, shuffled
    epochs=20,
    validation_data=val_gen
    validation_steps=val_steps) # 769 steps
```

6.3 Advanced use of recurrent neural networks

- Let’s display the loss curves for validation and training (see figure 6.20).

Listing 6.38 Plotting results

```
import matplotlib.pyplot as plt
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```



결과가 별로 안좋음
(학습 안한게 더 좋음)

Figure 6.20 Training and validation loss on the Jena temperature-forecasting task with a simple, densely connected network

6.3 Advanced use of recurrent neural networks

- Some of the validation losses are close to the no-learning baseline.
- why doesn’t the model you’re training find it and improve on it?
- significant limitation of machine learning
- hardcoded to look for a specific kind of simple model - parameter learning can sometimes fail to find a simple solution to a simple problem.

6.3 Advanced use of recurrent neural networks

6.3.5 A first recurrent baseline

- first flattened the time series - removed the notion of time from the input data.
- sequence - causality and order matter.
- recurrent-sequence processing model — exploits the temporal ordering of data points

GRU는 LSTM과 거의 같은데 Output이 하나임 , 근데 성능이 좀 떨어지지만 또 빠르긴 함;

- Gated recurrent unit (GRU) - same principle as LSTM, streamlined and cheaper to run (although they may not have as much representational power as LSTM)
- This trade-off between computational expensiveness and representational power is seen everywhere in machine learning.

6.3 Advanced use of recurrent neural networks

Listing 6.39 Training and evaluating a GRU-based model

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()  싫으면 LSTM, RNN 쓰면 됨 / 여기선 GRU 사용
model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

- Figure 6.21 shows the results - beat the common-sense baseline
- superiority of recurrent networks compared to sequence-flattening dense networks on this type of task.

6.3 Advanced use of recurrent neural networks

- The new validation MAE of ~0.265 - translates to a mean absolute error of 2.35˚C after denormalization.
- That's a solid gain on the initial error of 2.57˚C, but you probably still have a bit of a margin for improvement.

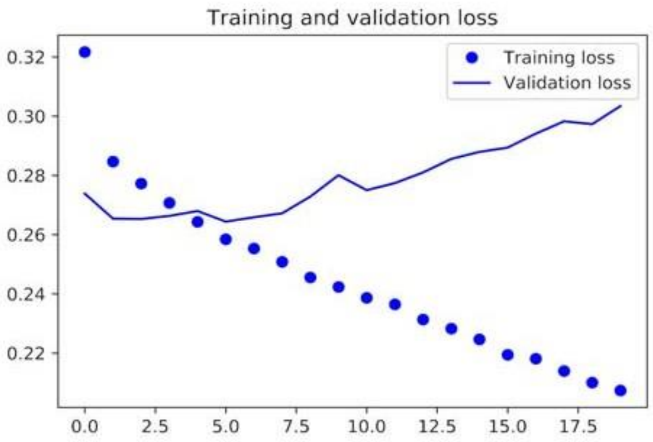


Figure 6.21 Training and validation loss on the Jena temperature-forecasting task with a GRU

6.3 Advanced use of recurrent neural networks

6.3.7 Stacking recurrent layers

- Because you're no longer overfitting but seem to have hit a performance bottleneck, you should consider increasing the capacity of the network. Recall the description of the universal machine-learning workflow: it's generally a good idea to increase the capacity of your network until overfitting becomes the primary obstacle (assuming
- you're already taking basic steps to mitigate overfitting, such as using dropout). As long as you aren't overfitting too badly, you're likely under capacity.
- Increasing network capacity is typically done by increasing the number of units in the layers or adding more layers. Recurrent layer stacking is a classic way to build more-powerful recurrent networks: for instance, what currently powers the Google
- Translate algorithm is a stack of seven large `LSTM` layers—that's huge.
- To stack recurrent layers on top of each other in Keras, all intermediate layers should return their full sequence of outputs (a 3D tensor) rather than their output at the last timestep. This is done by specifying `return_sequences=True`.

6.3 Advanced use of recurrent neural networks

6.3.6 Using recurrent dropout to fight overfitting

- Dropout - randomly zeros out input units of a layer in order to break happenstance correlations in the training data
- In 2015, Yarin Gal, as part of his PhD thesis on Bayesian deep learning: the same dropout mask (the same pattern of dropped units) should be applied at every time-step, instead of a dropout mask that varies randomly from timestep to timestep.
- What's more, in order to regularize the representations formed by the recurrent gates of layers such as `GRU` and `LSTM`, a temporally constant dropout mask should be applied to the inner recurrent activations of the layer (a *recurrent* dropout mask). Using the same dropout mask at every timestep allows the network to properly propagate its learning error through time; a temporally random dropout mask would disrupt this error signal and be harmful to the learning process.
- Yarin Gal did his research using Keras and helped build this mechanism directly into Keras recurrent layers. Every recurrent layer in Keras has two dropout-related arguments: `dropout`, a float specifying the dropout rate for input units of the layer

6.3 Advanced use of recurrent neural networks

Listing 6.40 Training and evaluating a dropout-regularized GRU-based model

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
          dropout=0.2,
          recurrent_dropout=0.2,
          input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                  steps_per_epoch=500, epochs=40,
                  validation_data=val_gen,
                  validation_steps=val_steps)
```

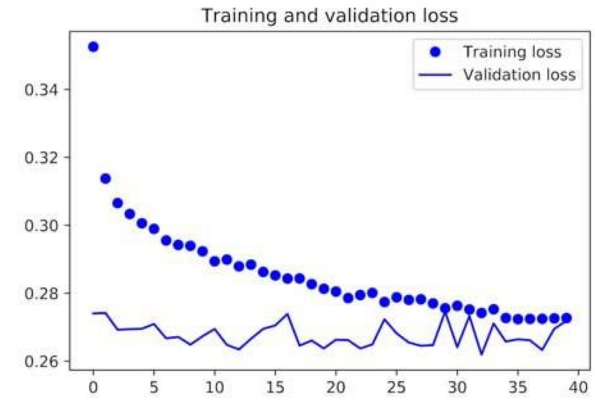


Figure 6.22 Training and validation loss on the Jena temperature-forecasting task with a dropout-regularized GRU

- Figure 6.22 shows the results. Success! You're no longer overfitting during the first 30 epochs. But although you have more stable evaluation scores, your best scores aren't much lower than they were previously.