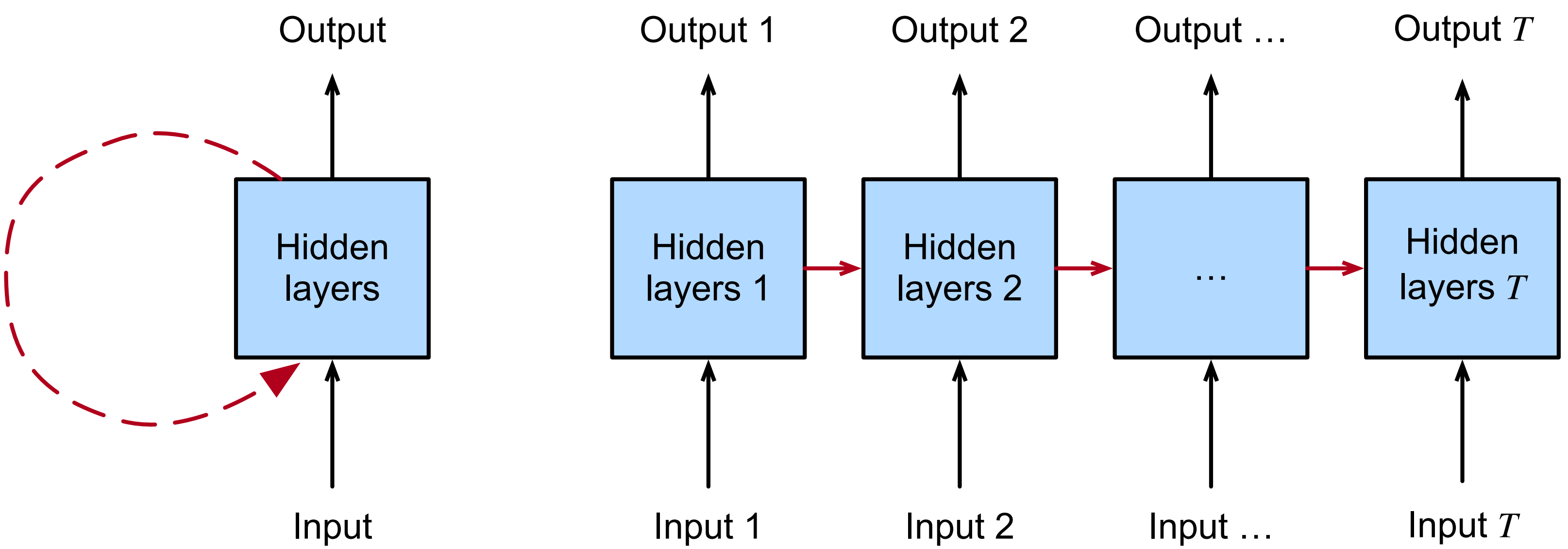
Recurrent and Long Short-term Memory Models

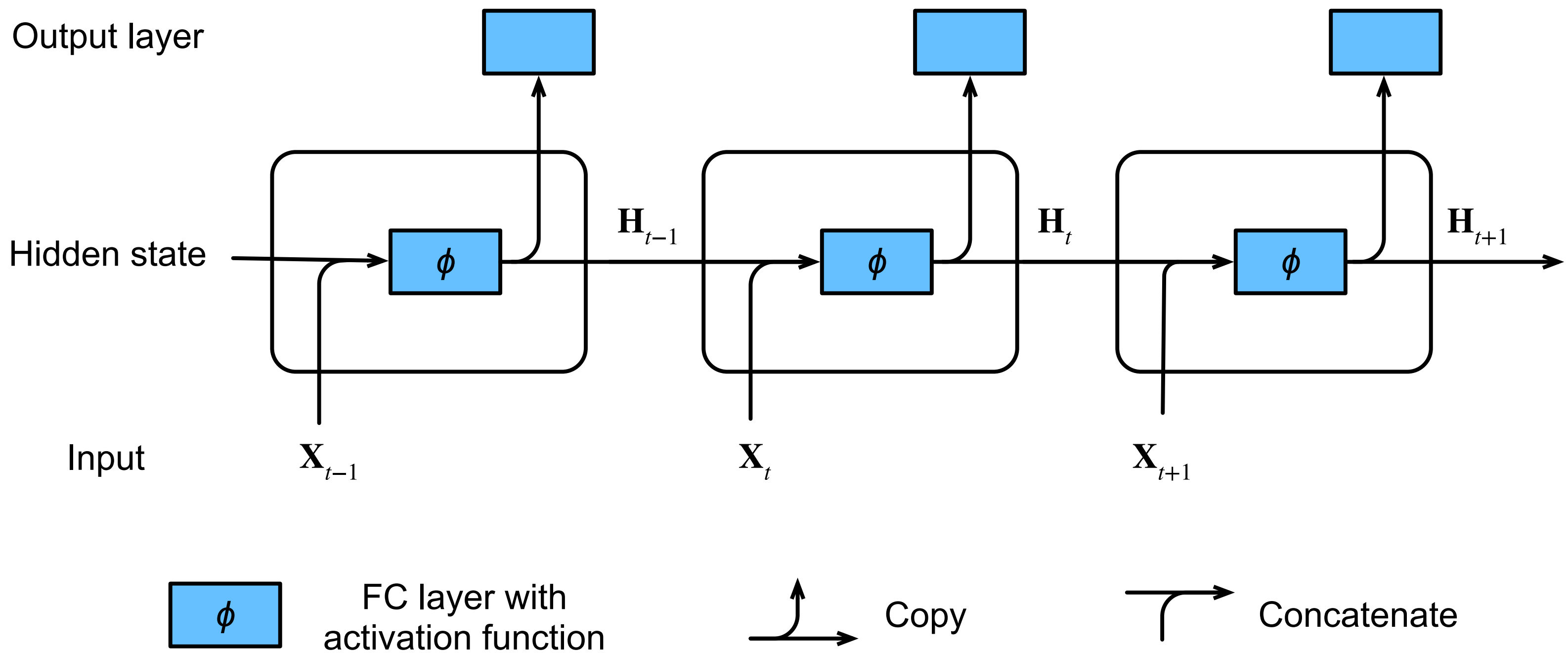
Recurrent Blocks/Networks

- Block structure: A single cell can be repeated, so that it **reoccurs** at each stage of the network.
- Innovation: Model the passage of information from one time step to the next, particularly when looking at a **sequence of linked observations**.

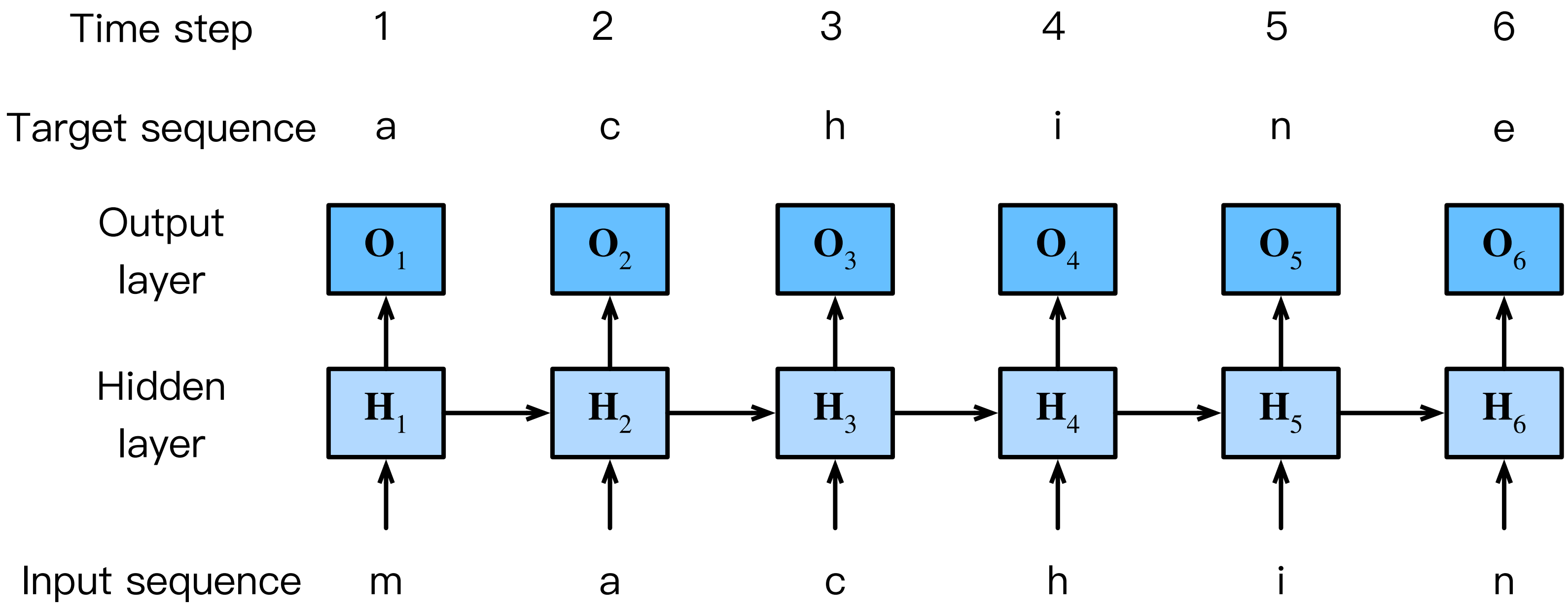
Recurrent networks



Hidden (memory) states



How the model works



How the model works

We pass a **stream** of data into our model, and predict the stream one step forward

- Only really making one prediction
- The stream feeds into the hidden state at each level of the network
- Each time stage is a separate "layer", but has the same weights as the other layers
- Each layer interacts with the "next" part of the sequence

How the model works

Different from our past models, an RNN returns **two** objects:

- The output from a specific layer
- The updated hidden state (handed off to the next layer)

Long Short-term Memory Networks (LSTMs)

Was the state of the art until transformer architectures took over in 2017, although they still inspire aspects of those models.

Still valuable for time-series modeling, though!

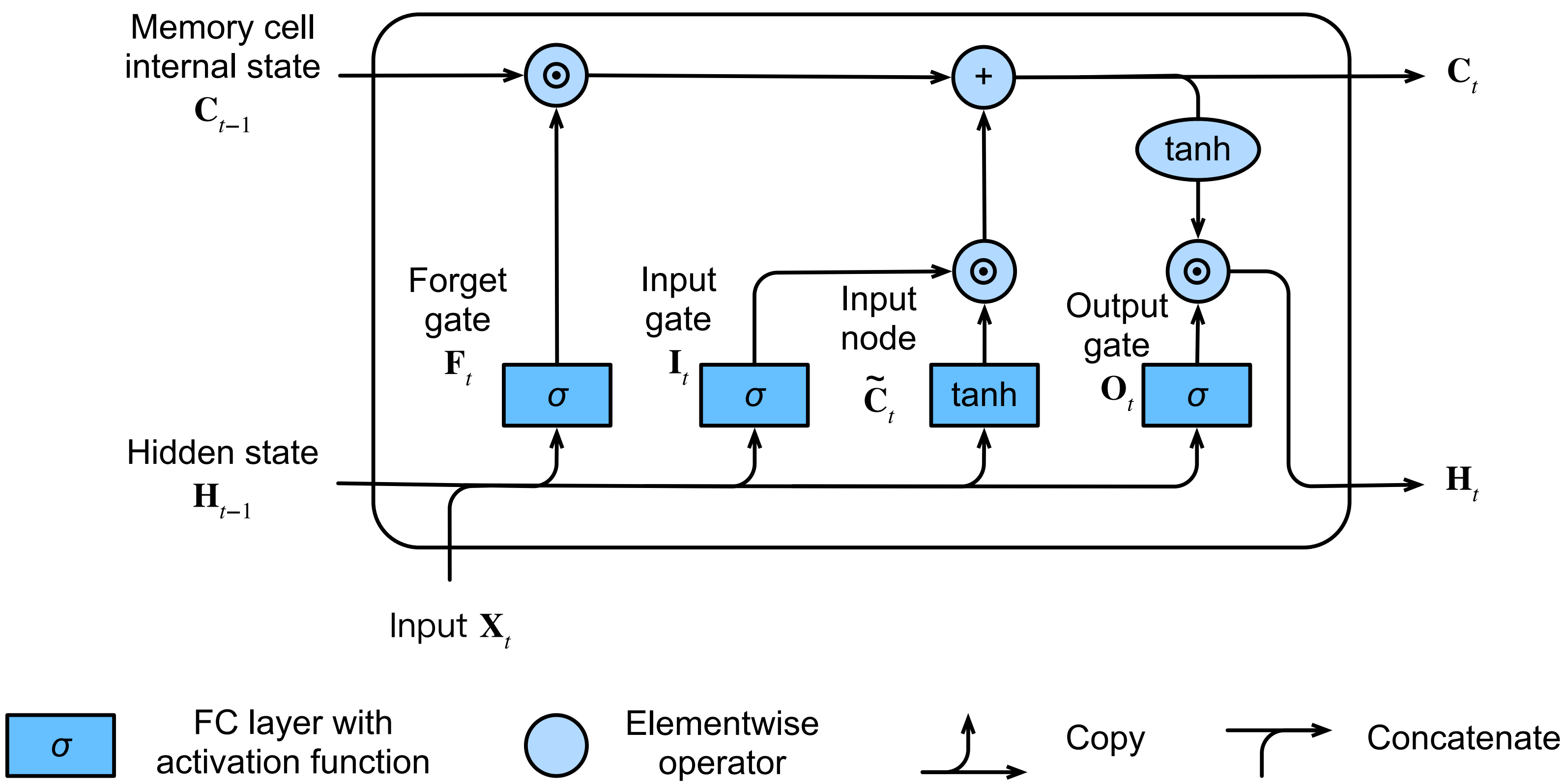
LSTM Advances

Rather than having a simple hidden state, LSTM models implement a more complex hidden structure:

- Input gates
- Forget gates
- Output gates

These allow different hidden states to have different impacts dependent on the current input, rather than uniform effects like in RNN models

Memory cell structures in LSTMs



LSTM key differences

- **Memory** that is propagated from one period to the next
  - That memory is forgotten based on the activation of the **forget gate**
- Inputs are filtered through the **input gate** before being added to memory
- Outputs are a combination of the **hidden state**, memory, and our current inputs

Hidden State vs Memory Cell

- The hidden state **always** affects the output
- Memory cells only affect output as permitted by the **output gate**, so that they have *selective* impacts on predictions

Model outputs

At each stage, the model will return **three** objects:

- The output (prediction)
- The hidden state
- The memory state

Both the hidden and memory states are passed on to the next round of the model

Make an LSTM!

Let's build one of these so we can better see how it works.

Make an LSTM

First, we need to install some helpers (our old ones won't cut it anymore...)

```
!pip install d2l
```

Make an LSTM

```
# Let's use Dracula as our source text:
# https://www.gutenberg.org/cache/epub/345/pg345.txt

# For reading/cleaning data
import requests
import re
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

# For visualizing
import plotly.express as px

# For model building
import torch
import torch.nn as nn
import torch.nn.functional as F

# Import helpers
# import nnhelpers as nnh
from d2l import torch as d2l
```

Make an LSTM

```
# Customized version of the d2l Time Machine class object,
#   but using a better source text

class Dracula(d2l.DataModule):

    def _download(self):
        dracula = "https://www.gutenberg.org/cache/epub/345/pg345.txt"
        fname = d2l.download(dracula, self.root)
        with open(fname) as f:
            return f.read()

    def _preprocess(self, text):
        """Defined in :numref:`sec_text-sequence`"""
        return re.sub('[^A-Za-z]+', ' ', text).lower()

    def _tokenize(self, text):
        """Defined in :numref:`sec_text-sequence`"""
        return list(text)
```

Make an LSTM

```
class Dracula(d2l.DataModule):
    ...

    def build(self, raw_text, vocab=None):
        """Defined in :numref:`sec_text-sequence`"""
        tokens = self._tokenize(self._preprocess(raw_text))
        if vocab is None: vocab = d2l.Vocab(tokens)
        corpus = [vocab[token] for token in tokens]
        return corpus, vocab

    def __init__(self, batch_size, num_steps, num_train=10000, num_val=5000):
        """Defined in :numref:`sec_language-model`"""
        super(Dracula, self).__init__()
        self.save_hyperparameters()
        corpus, self.vocab = self.build(self._download())
        array = d2l.tensor([corpus[i:i+num_steps+1]
                            for i in range(len(corpus)-num_steps)])
        self.X, self.Y = array[:,:-1], array[:,1:]

    def get_dataloader(self, train):
        """Defined in :numref:`subsec_partitioning-seqs`"""
        idx = slice(0, self.num_train) if train else slice(
            self.num_train, self.num_train + self.num_val)
        return self.get_tensorloader([self.X, self.Y], train, idx)
```

Make an LSTM

```
class LSTM(d2l.Module):
    def __init__(self, num_inputs, num_hiddens,
                 sigma=0.01, lr=3, numeric=False):
        super().__init__()
        self.save_hyperparameters()
        self.lr = lr
        self.numeric = numeric
        init_weight = lambda *shape: nn.Parameter(
            torch.randn(*shape) * sigma)
        triple = lambda: (init_weight(num_inputs, num_hiddens),
                          init_weight(num_hiddens, num_hiddens),
                          nn.Parameter(torch.zeros(num_hiddens)))
        self.W_xi, self.W_hi, self.b_i = triple()  # Input gate
        self.W_xf, self.W_hf, self.b_f = triple()  # Forget gate
        self.W_xo, self.W_ho, self.b_o = triple()  # Output gate
        self.W_xc, self.W_hc, self.b_c = triple()  # Input node
```

Make an LSTM

```
class LSTM(d2l.Module):
    ...

    def forward(self, inputs, H_C=None):
        if H_C is None:
            # Initial state with shape: (batch_size, num_hiddens)
            H = torch.zeros((inputs.shape[1], self.num_hiddens),
                              device=inputs.device)
            C = torch.zeros((inputs.shape[1], self.num_hiddens),
                              device=inputs.device)
        else:
            H, C = H_C
        outputs = []
        for X in inputs:
            I = torch.sigmoid(torch.matmul(X, self.W_xi) +
                                torch.matmul(H, self.W_hi) + self.b_i)
            F = torch.sigmoid(torch.matmul(X, self.W_xf) +
                                torch.matmul(H, self.W_hf) + self.b_f)
            O = torch.sigmoid(torch.matmul(X, self.W_xo) +
                                torch.matmul(H, self.W_ho) + self.b_o)
            C_tilde = torch.tanh(torch.matmul(X, self.W_xc) +
                                   torch.matmul(H, self.W_hc) + self.b_c)
            C = F * C + I * C_tilde
            H = O * torch.tanh(C)
            outputs.append(H)
        return outputs, (H, C)
```

Make an LSTM

```
data = Dracula(batch_size=1024, num_steps=100)
lstm = LSTM(num_inputs=len(data.vocab), num_hiddens=128)
model = d2l.RNNLMScratch(lstm, vocab_size=len(data.vocab), lr=4)
trainer = d2l.Trainer(max_epochs=50, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```

Make an LSTM

```
model.predict('as the boat arrived at the quay ',
    50, data.vocab, d2l.try_gpu())
```

```
'as the boat arrived at the quay the
country stound the country stound the country '
```

Continuous Time Series LSTM

```
# Time series models inspired by
#   https://machinelearningmastery.com/
#     lstm-for-time-series-prediction-in-pytorch/

import pandas as pd
import plotly.express as px
import numpy as np

# Read in data, grab relevant column
temp = pd.read_csv("https://github.com/dustywhite7/Econ8310/
    raw/master/DataSets/omahaNOAA.csv")
temp = temp['HOURLYDRYBULBTEMPF'].fillna(0
    ).replace(0, method='pad').values[-(365*24):]

px.line(temp)
```

Continuous Time Series LSTM

```
# train-test split for time series
train_size = int(len(temp) * 0.67)
test_size = len(temp) - train_size
train, test = temp[:train_size], temp[train_size:]
```

Continuous Time Series LSTM

```
def create_dataset(dataset, lookback):

    X, y = [], []
    for i in range(len(dataset)-lookback):
        feature = dataset[i:i+lookback]
        target = dataset[i+1:i+lookback+1]
        X.append(feature)
        y.append(target)
    return torch.tensor(X, dtype=torch.float32),
        torch.tensor(y, dtype=torch.float32)
```

Continuous Time Series LSTM

```
lookback = 1
X_train, y_train = create_dataset(train, lookback=lookback)
X_test, y_test = create_dataset(test, lookback=lookback)
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

Continuous Time Series LSTM

```
class TempLSTM(nn.Module):
    def __init__(self):
        super().__init__()
        self.lstm = nn.LSTM(input_size=lookback,
             hidden_size=50, num_layers=1, batch_first=True)
        self.linear = nn.LazyLinear(1)
    def forward(self, x):
        x, _ = self.lstm(x)
        x = self.linear(x)
        return x
```

Training our model

```
model = TempLSTM()
optimizer = torch.optim.Adam(model.parameters())
loss_fn = nn.MSELoss()
loader = DataLoader(TensorDataset(X_train, y_train),
     shuffle=True, batch_size=32)

n_epochs = 200
for epoch in range(n_epochs):
    model.train()
    for X_batch, y_batch in loader:
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    # Validation
    if epoch % 10 != 0:
        continue
    model.eval()
    with torch.no_grad():
        y_pred = model(X_train)
        train_rmse = np.sqrt(loss_fn(y_pred, y_train))
        y_pred = model(X_test)
        test_rmse = np.sqrt(loss_fn(y_pred, y_test))
    print("Epoch %d: train RMSE %.4f, test RMSE %.4f"
        % (epoch, train_rmse, test_rmse))
```

Forecasting

```
with torch.no_grad():
    # shift train predictions for plotting
    train_plot = np.ones_like(temp) * np.nan
    y_pred = model(X_train)
    y_pred = y_pred[:, -1]
    train_plot[lookback:train_size] = model(X_train)[:, -1]
    # shift test predictions for plotting
    test_plot = np.ones_like(temp) * np.nan
    test_plot[train_size+lookback:len(temp)] = model(X_test)[:, -1]
```

Plotting our forecast

```
# Build plotting data
plot_data = pd.DataFrame([temp, test_plot]).T
plot_data.columns = ['truth', 'forecast']

px.line(plot_data, y = ['truth', 'forecast'])
```

LSTMs in summary

- Designed to handle time-series/sequential data
- Allow for both consistent and provisional inputs to the model

Lab Time!