Embeddings
Recurrent Neural Networks, and Sequences (Part 3)

May 25, 2020

ddebarr@uw.edu

http://cross-entropy.net/ML410/Deep_Learning_5.pdf

Agenda

- Homework Review

- [DLP] Deep Learning for Text and Sequences

[DLP] Deep Learning for Text and Sequences

1. Working with Text Data
2. Understanding Recurrent Networks
3. Advanced Use of Recurrent Neural Networks
4. Sequence Processing with ConvNets

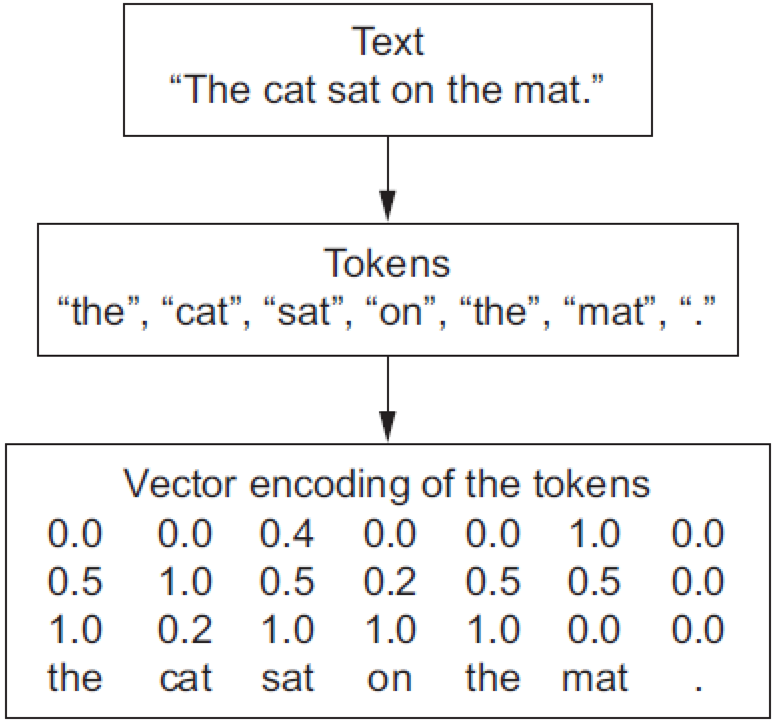
Applications

- Document classification and timeseries classification, such as identifying the topic of an article or the author of a book
- Timeseries comparisons, such as estimating how closely related two documents or two stock tickers are
- Sequence-to-sequence learning, such as decoding an English sentence into French
- Sentiment analysis, such as classifying the sentiment of tweets or movie reviews as positive or negative
- Timeseries forecasting, such as predicting the future weather at a certain location, given recent weather data

Vectorizing the Text

- Transforming text into numeric tensors
- Multiple possibilities exists for tokenization …
  - Segment text into words, and transform each word into a vector
  - Segment text into characters, and transform each character into a vector
  - Extract n-grams of words or characters, and transform each n-gram into a vector [n-grams are overlapping groups of multiple consecutive words or characters]
- Methods for encoding include ..
  - Multi-hot encoding [indicators for presence of tokens]
  - Term Frequency – Inverse Document Frequency encoding (TF-IDF)
  - Token embeddings (typically used for words and called word embeddings)

Text to Tokens to Vectors



N-Grams

- Names
  - 1-grams are called unigrams
  - 2-grams are called bigrams
  - 3-grams are called trigrams
  - 4-grams and called … 4-grams
- “The cat sat on the mat”
  - Unigrams: { “The”, “cat”, “sat”, “on”, “the”, “mat” }
  - Bigrams: { “The cat”, “cat sat”, “sat on”, “on the”, “the mat” }
  - Trigrams: { “The cat sat”, “cat sat on”, “sat on the”, “on the mat” }
- Note: the term “bag” refers to an unordered set rather than a sequence

Word-Level One-Hot Encoding

Builds an index of all tokens in the data

Initial data: one entry per sample (in this example, a sample is a sentence, but it could be an entire document)

Tokenizes the samples via the split method. In real life, you'd also strip punctuation and special characters from the samples.

```
import numpy as np

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

token_index = {}
for sample in samples:
    for word in sample.split():
        if word not in token_index:
            token_index[word] = len(token_index) + 1

max_length = 10

results = np.zeros(shape=(len(samples),
                          max_length,
                          max(token_index.values()) + 1))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = token_index.get(word)
        results[i, j, index] = 1.
```

This is where you store the results.

Assigns a unique index to each unique word. Note that you don't attribute index 0 to anything.

Vectorizes the samples. You'll only consider the first max_length words in each sample.

Character-Level One-Hot Encoding

```
import string

samples = ['The cat sat on the mat.', 'The dog ate my homework.']
characters = string.printable
token_index = dict(zip(range(1, len(characters) + 1), characters))

max_length = 50
results = np.zeros((len(samples), max_length, max(token_index.keys()) + 1))
for i, sample in enumerate(samples):
    for j, character in enumerate(sample):
        index = token_index.get(character)
        results[i, j, index] = 1.
```

All printable ASCII characters

Using Keras for Word-Level Multi-Hot Encoding

```
from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(samples)

sequences = tokenizer.texts_to_sequences(samples)

one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

Creates a tokenizer, configured to only take into account the 1,000 most common words

Builds the word index

Turns strings into lists of integer indices

You could also directly get the one-hot binary representations. Vectorization modes other than one-hot encoding are supported by this tokenizer.

How you can recover the word index that was computed

Word-Level One-Hot Encoding with Hashing Trick

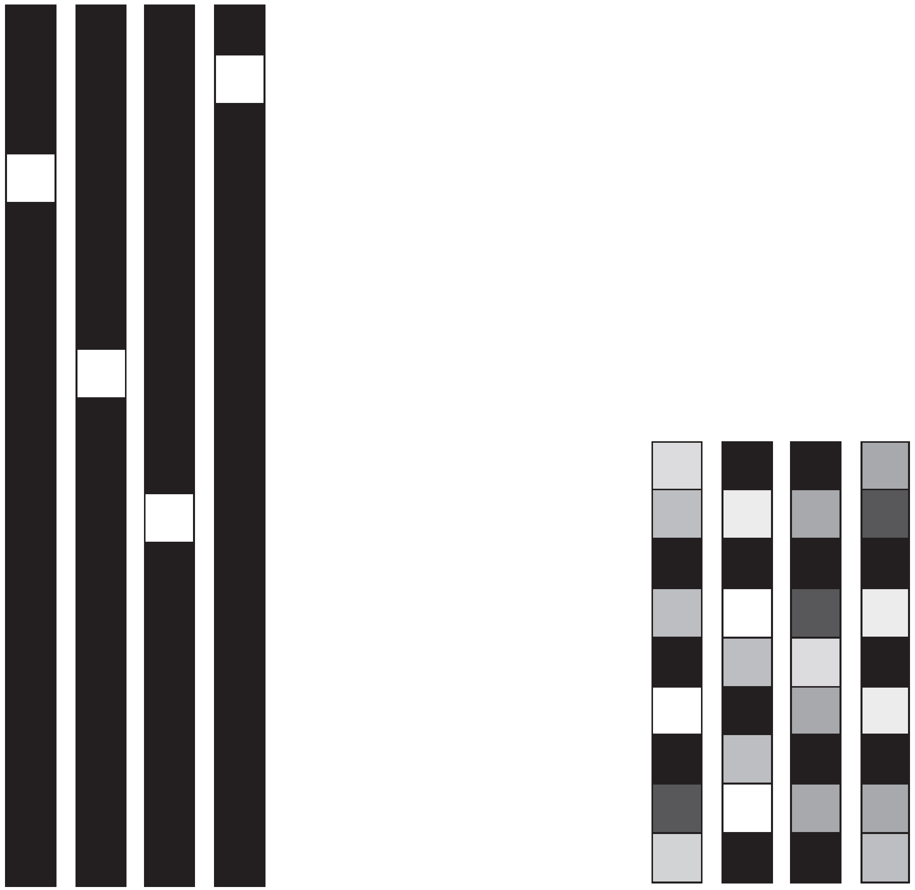
```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']

dimensionality = 1000
max_length = 10

results = np.zeros((len(samples), max_length, dimensionality))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = abs(hash(word)) % dimensionality
        results[i, j, index] = 1.
```

Stores the words as vectors of size 1,000. If you have close to 1,000 words (or more), you'll see many hash collisions, which will decrease the accuracy of this encoding method.

Hashes the word into a random integer index between 0 and 1,000

Sparse versus Dense Representation

The primary curse of dimensionality is sparsity



One-hot word vectors:
- Sparse
- High-dimensional
- Hardcoded

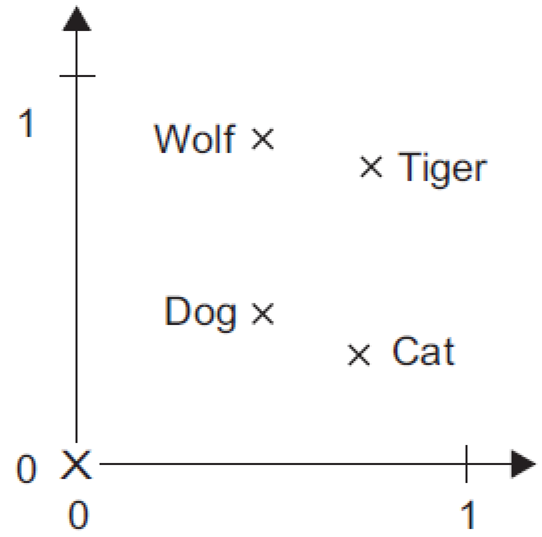
Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

Two Ways to Obtain Word Embeddings

- Learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction). In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of a neural network.
- Load into your model word embeddings that were precomputed using a different machine-learning task than the one you're trying to solve. These are called *pretrained word embeddings*.

Toy Example of a Word Embedding Space

- Vertical dimension could be interpreted as a “wild” index
- Horizontal dimension could be interpreted as a “feline” index



- Another popular example:
  - Queen – Woman == King – Man
  - Consider the possibility that the dimensions include feminine and masculine indexes

Instantiating an Embedding Layer

```
from keras.layers import Embedding

embedding_layer = Embedding(1000, 64)
```

The Embedding layer takes at least two arguments: the number of possible tokens (here, 1,000: 1 + maximum word index) and the dimensionality of the embeddings (here, 64).

Word index → Embedding layer → Corresponding word vector

Loading the Internet Movie DataBase (IMDB) Data for Use with an Embedding Layer

```
from keras.datasets import imdb
from keras import preprocessing

max_features = 10000
maxlen = 20

(x_train, y_train), (x_test, y_test) = imdb.load_data(
    num_words=max_features)

x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
```

Number of words to consider as features

Cuts off the text after this number of words (among the max_features most common words)

Loads the data as lists of integers

Turns the lists of integers into a 2D integer tensor of shape (samples, maxlen)

Only the first 20 words of the review

Using an Embedding Layer and Classifier on the IMDB Data

Specifies the maximum input length to the Embedding layer so you can later flatten the embedded inputs. After the Embedding layer, the activations have shape (samples, maxlen, 8).

Flattens the 3D tensor of embeddings into a 2D tensor of shape (samples, maxlen * 8)

Adds the classifier on top

```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Embedding

model = Sequential()
model.add(Embedding(10000, 8, input_length=maxlen))

model.add(Flatten())

model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
model.summary()

history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_split=0.2)
```

~ 76% Accuracy

Pretrained Word Embeddings

- Word2Vec: skipgram and continuous bag-of-words architectures
  - https://code.google.com/archive/p/word2vec
- Global Vectors (GloVe): matrix factorization
  - https://nlp.stanford.edu/projects/glove
- Embedding() arguments
  - embeddings_initializer=keras.initializers.Constant(embedding_matrix)
  - trainable=False

Processing the Labels of the Raw IMDB Data

http://mng.bz/0tIo

```
import os

imdb_dir = '/Users/fchollet/Downloads/aclImdb'
train_dir = os.path.join(imdb_dir, 'train')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

Tokenizing the Text of the Raw IMDB Data

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100                    <- Cuts off reviews after 100 words
training_samples = 200          <- Trains on 200 samples
validation_samples = 10000      <- Validates on 10,000 samples
max_words = 10000               <- Considers only the top 10,000 words in the dataset

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
```

Creating the Train and Val Data Sets for IMDB

```
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]
```

Splits the data into a training set and a validation set, but first shuffles the data, because you're starting with data in which samples are ordered (all negative first, then all positive)

Parsing the GloVe Word-Embeddings File

http://nlp.stanford.edu/data/glove.6B.zip

```
glove_dir = '/Users/fchollet/Downloads/glove.6B'

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
```

Preparing the GloVe Word Embeddings Matrix

```
embedding_dim = 100

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    if i < max_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
```

Words not found in the embedding index will be all zeros.

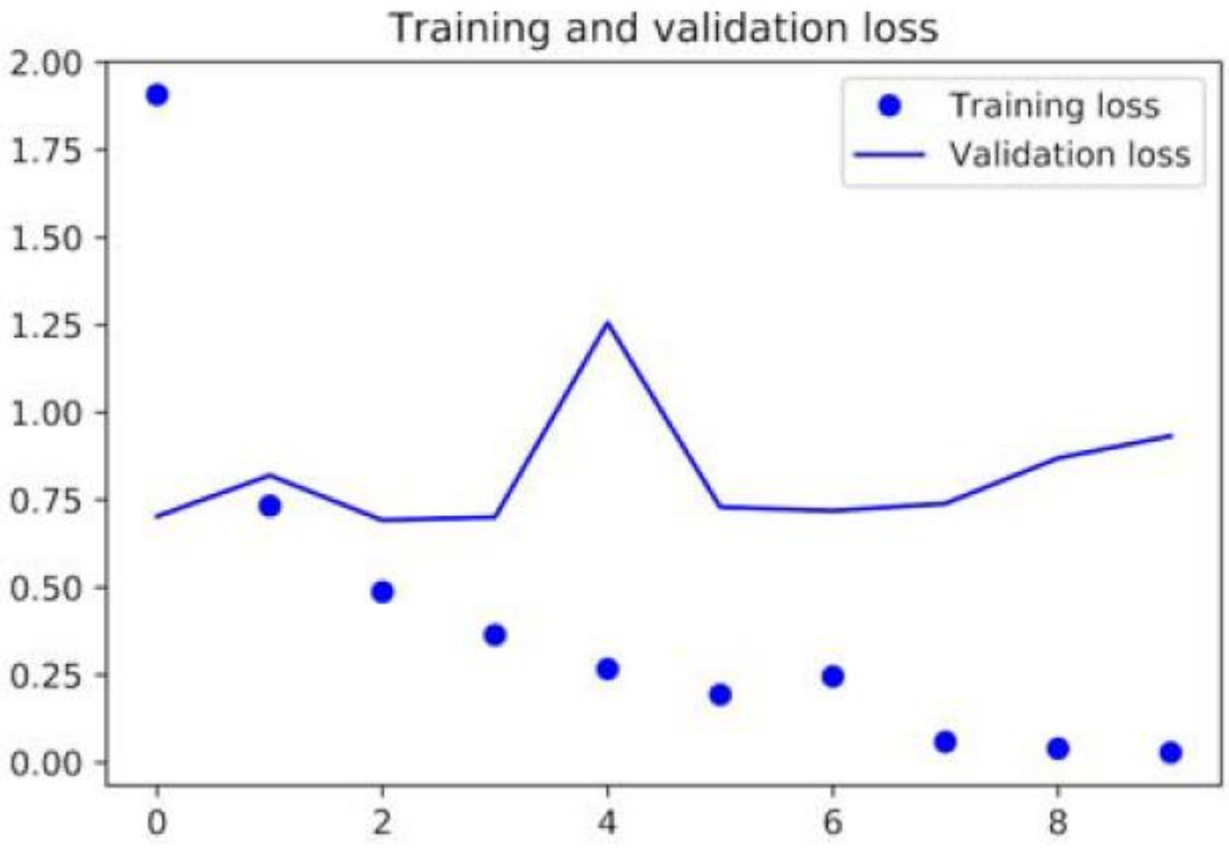
Model Definition

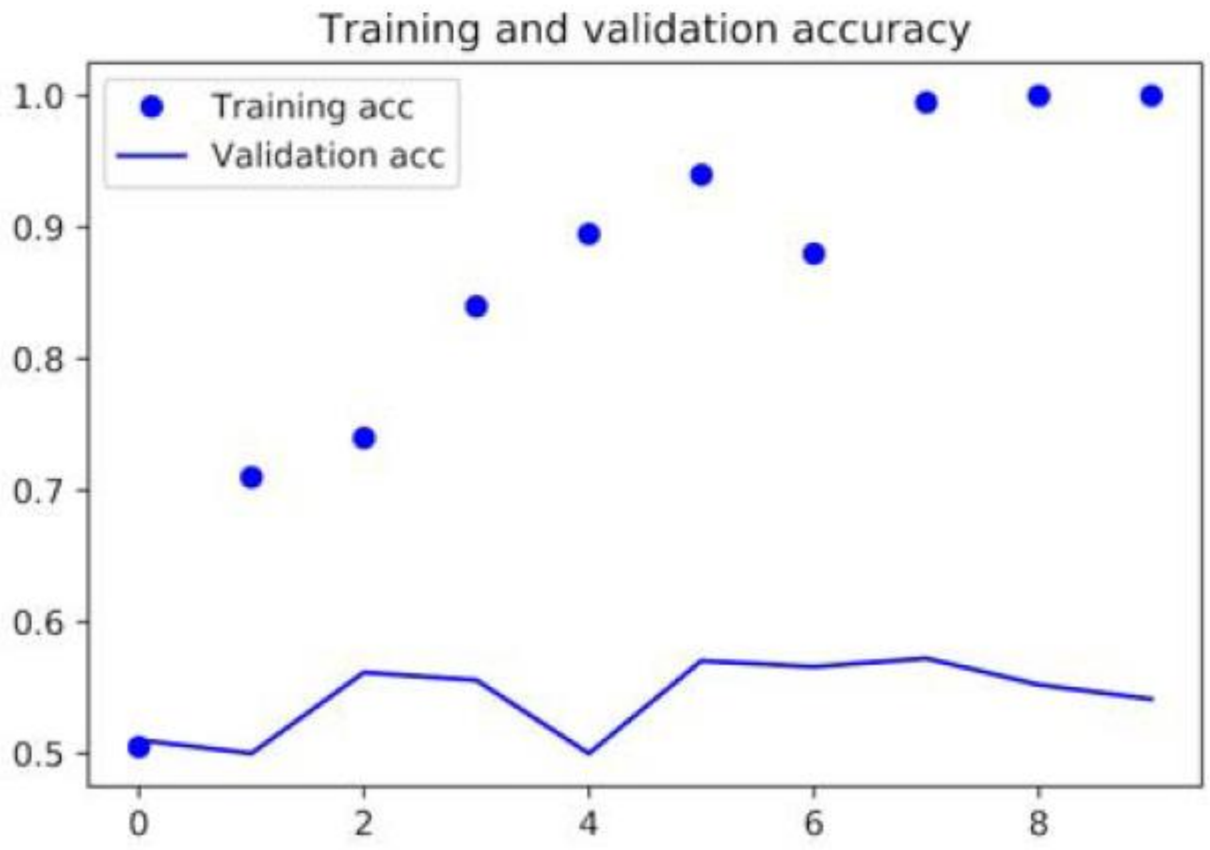
```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()


model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

Training and Evaluation

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(x_val, y_val))
model.save_weights('pre_trained_glove_model.h5')
```

Loss and Accuracy



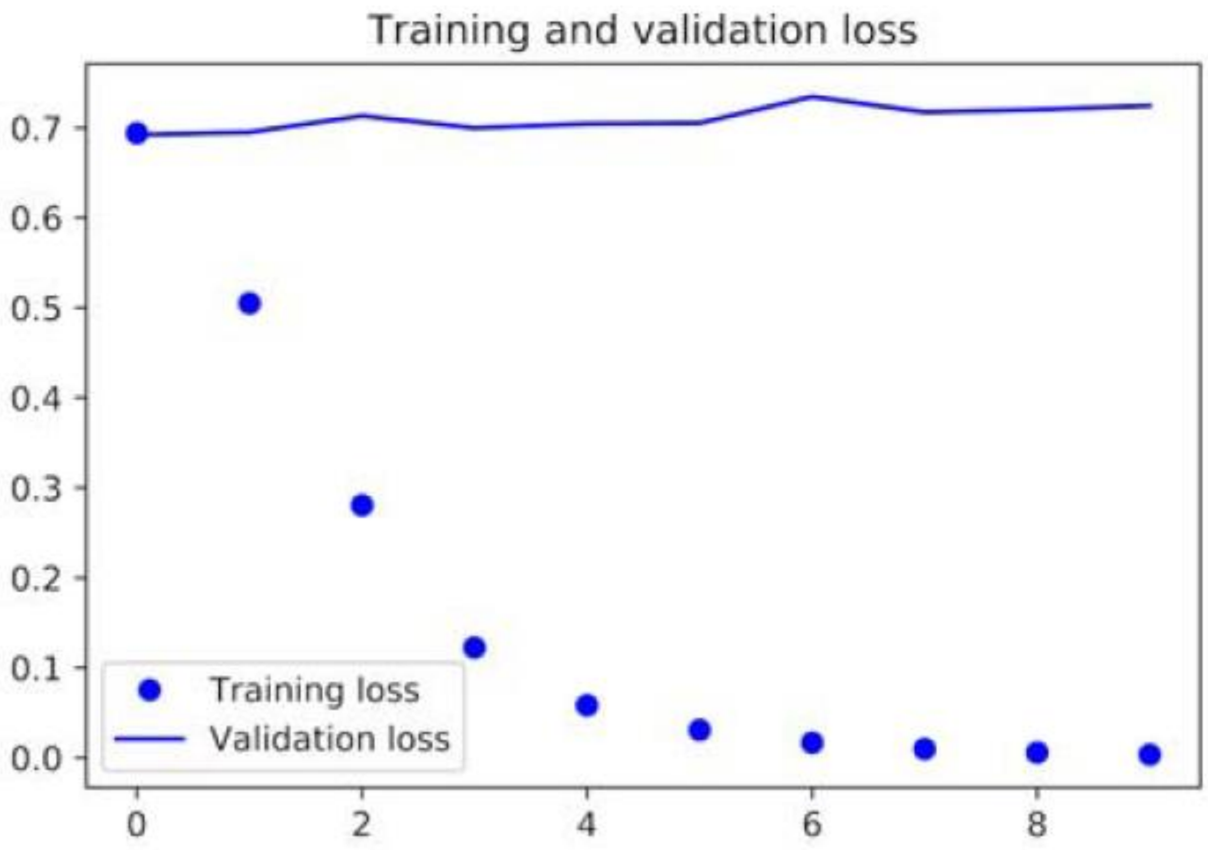


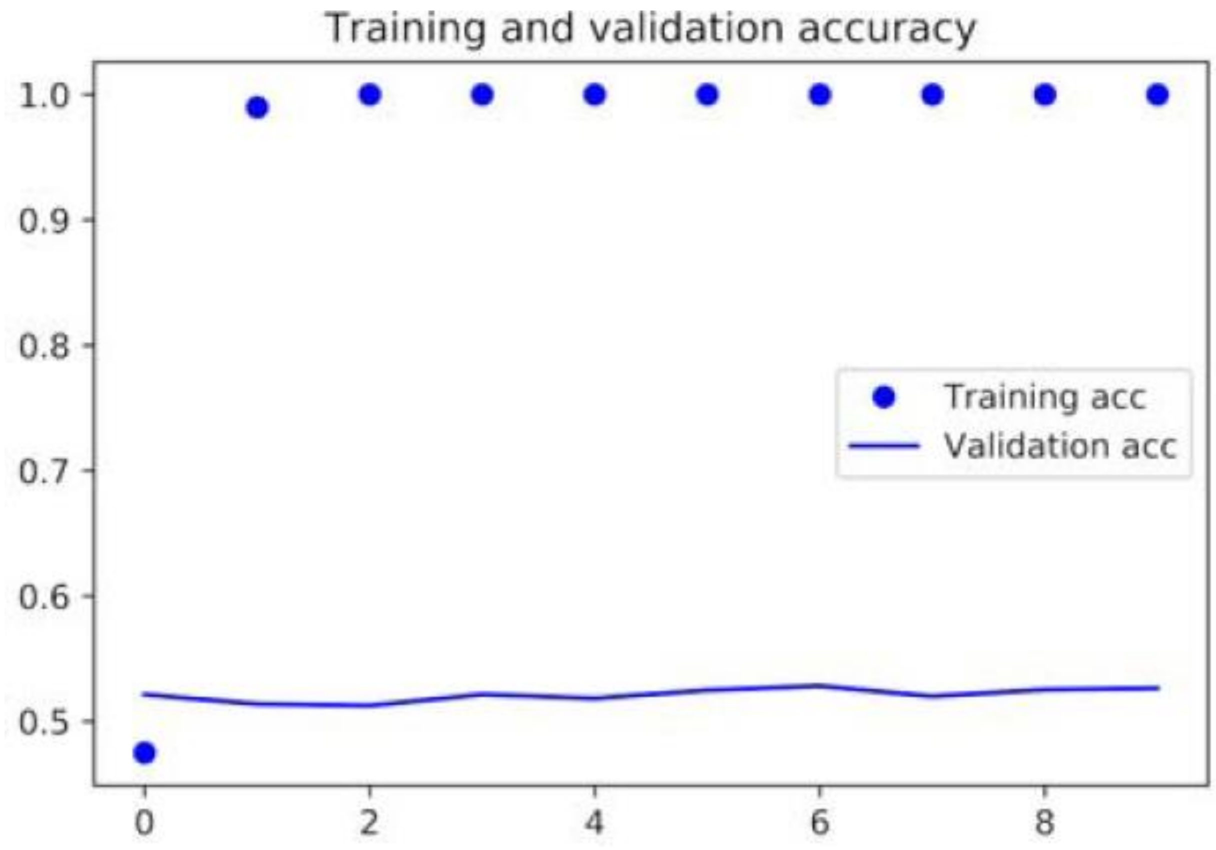
Accuracy in the mid-50s

Training the Same Model Without Pretrained Word Embeddings

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(x_val, y_val))
```

Loss and Accuracy





Accuracy in the low-50s

Tokenizing the Test Set

```
test_dir = os.path.join(imdb_dir, 'test')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(test_dir, label_type)
    for fname in sorted(os.listdir(dir_name)):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)
```

Evaluating the Model with Pretrained Embeddings

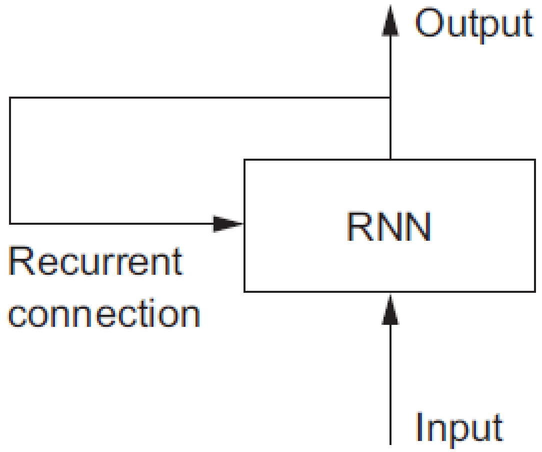
56% accuracy [okay, given only 200 training observations]

```
model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)
```

Wrapping Up

- Turn raw text into something a neural network can process
- Use the Embedding layer in a Keras model to learn task-specific token embeddings
- Use pretrained word embeddings to get an extra boost on small-data natural language-processing problems

Recurrent Network: a Network with a Loop



Pseudocode RNN

```
state_t = 0                                 The state at t
for input_t in input_sequence:              Iterates over sequence elements
    output_t = f(input_t, state_t)
    state_t = output_t                      The previous output becomes the state for the next iteration.
```

More Detailed Pseudocode for the RNN

```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

Numpy Implementation of a Simple RNN

Number of timesteps in the input sequence

Dimensionality of the input feature space

Dimensionality of the output feature space

Input data: random noise for the sake of the example

Initial state: an all-zero vector

Creates random weight matrices

input_t is a vector of shape (input_features,).

```
import numpy as np

timesteps = 100
input_features = 32
output_features = 64

inputs = np.random.random((timesteps, input_features))

state_t = np.zeros((output_features,))

W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))

successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)

    successive_outputs.append(output_t)

    state_t = output_t

final_output_sequence = np.concatenate(successive_outputs, axis=0)
```

Stores this output in a list

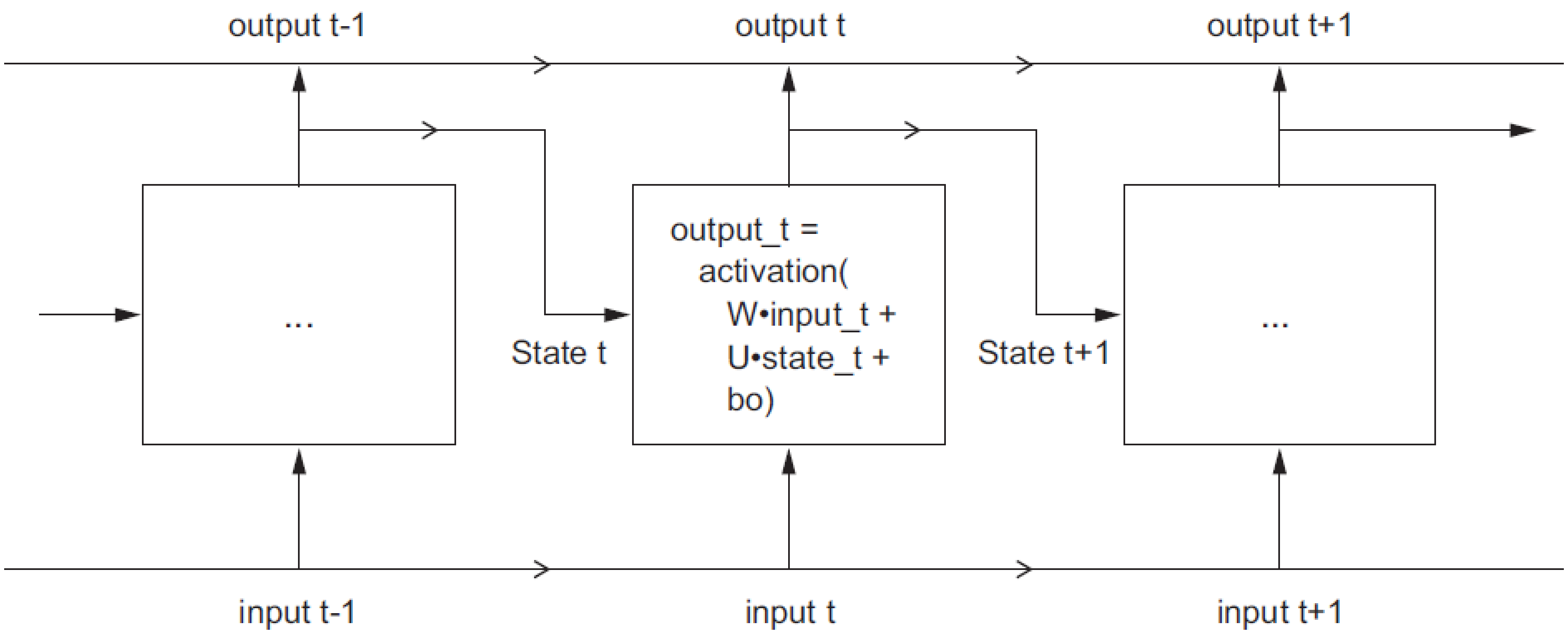
The final output is a 2D tensor of shape (timesteps, output_features).

Combines the input with the current state (the previous output) to obtain the current output

Updates the state of the network for the next timestep

A Simple RNN, Unrolled Over Time

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```



SimpleRNN: Only Returning Last Output

```
>>> from keras.models import Sequential
>>> from keras.layers import Embedding, SimpleRNN
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32))
>>> model.summary()
```

```
________________________________________________________________
Layer (type)                     Output Shape          Param #
================================================================
embedding_22 (Embedding)         (None, None, 32)      320000
________________________________________________________________
simplernn_10 (SimpleRNN)         (None, 32)            2080
================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

SimpleRNN: Returning All Outputs

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.summary()
________________________________________________________________
Layer (type)                     Output Shape          Param #
================================================================
embedding_23 (Embedding)         (None, None, 32)      320000
________________________________________________________________
simplernn_11 (SimpleRNN)         (None, None, 32)      2080
================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

Stacking SimpleRNN Layers [must return sequences to stack]

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32))
>>> model.summary()
```

Last layer only returns the last output

```
________________________________________________________________
Layer (type)                     Output Shape          Param #
================================================================
embedding_24 (Embedding)         (None, None, 32)      320000
________________________________________________________________
simplernn_12 (SimpleRNN)         (None, None, 32)      2080
________________________________________________________________
simplernn_13 (SimpleRNN)         (None, None, 32)      2080
________________________________________________________________
simplernn_14 (SimpleRNN)         (None, None, 32)      2080
________________________________________________________________
simplernn_15 (SimpleRNN)         (None, 32)            2080
================================================================
Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0
```

Preparing the IMDB Data

```
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 10000
maxlen = 500
batch_size = 32

print('Loading data...')
(input_train, y_train), (input_test, y_test) = imdb.load_data(
     num_words=max_features)
print(len(input_train), 'train sequences')
print(len(input_test), 'test sequences')

print('Pad sequences (samples x time)')
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train shape:', input_train.shape)
print('input_test shape:', input_test.shape)
```

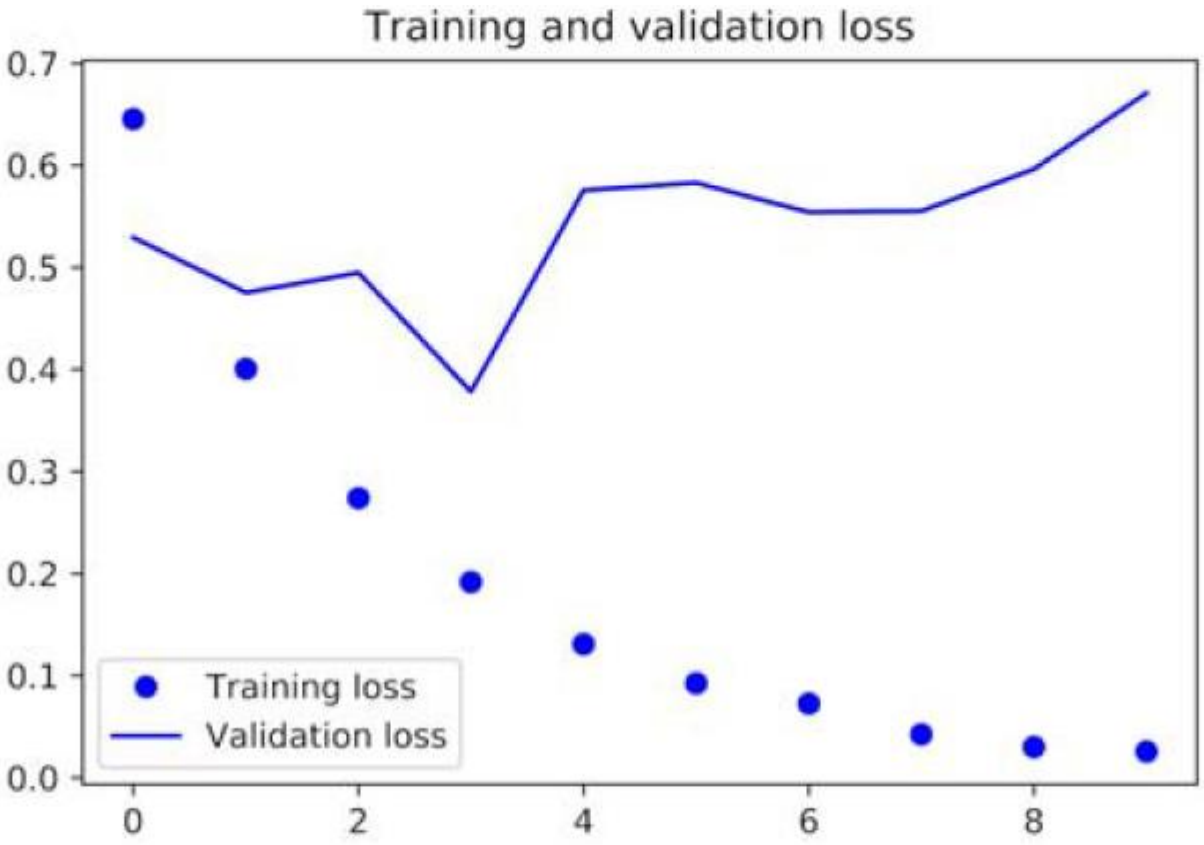
Number of words to consider as features

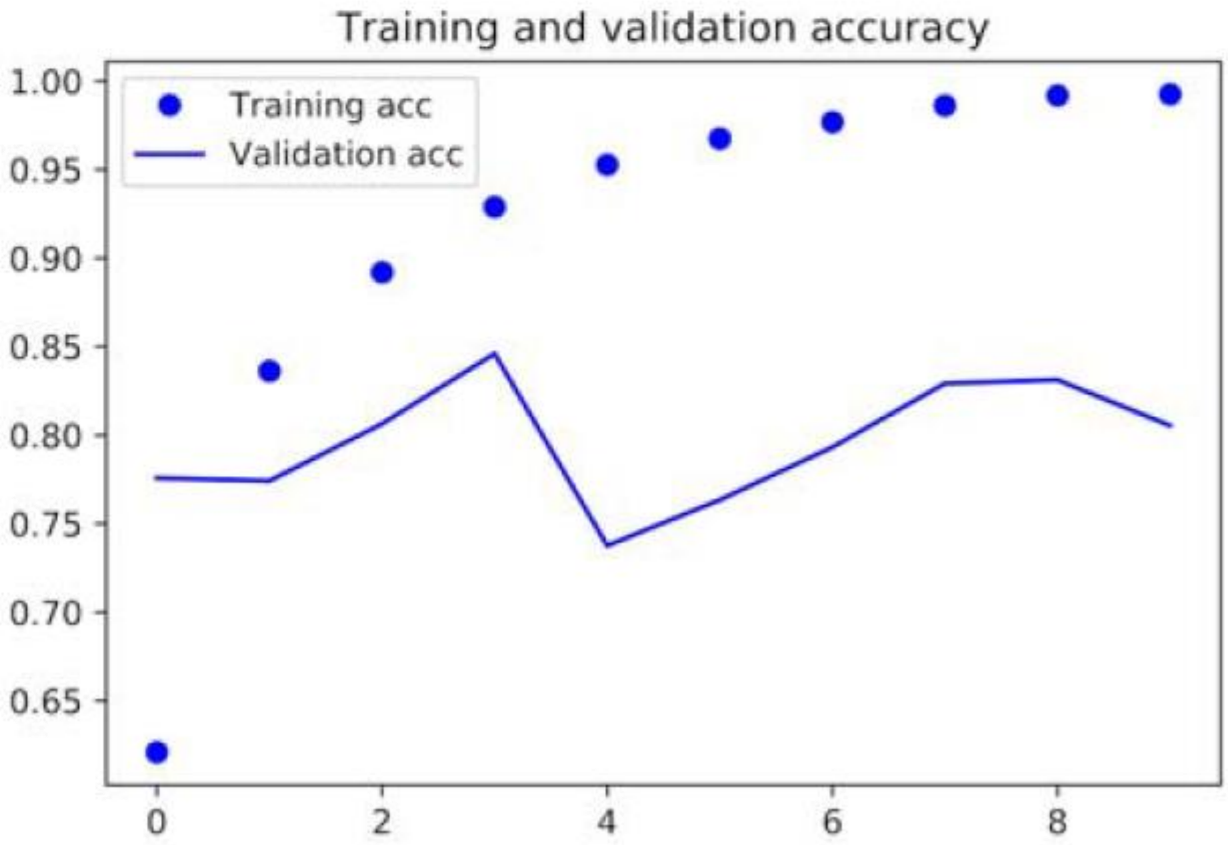
Cuts off texts after this many words (among the max_features most common words)

Training the Model with Embedding and SimpleRNN Layers

```
from keras.layers import Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                     validation_split=0.2)
```

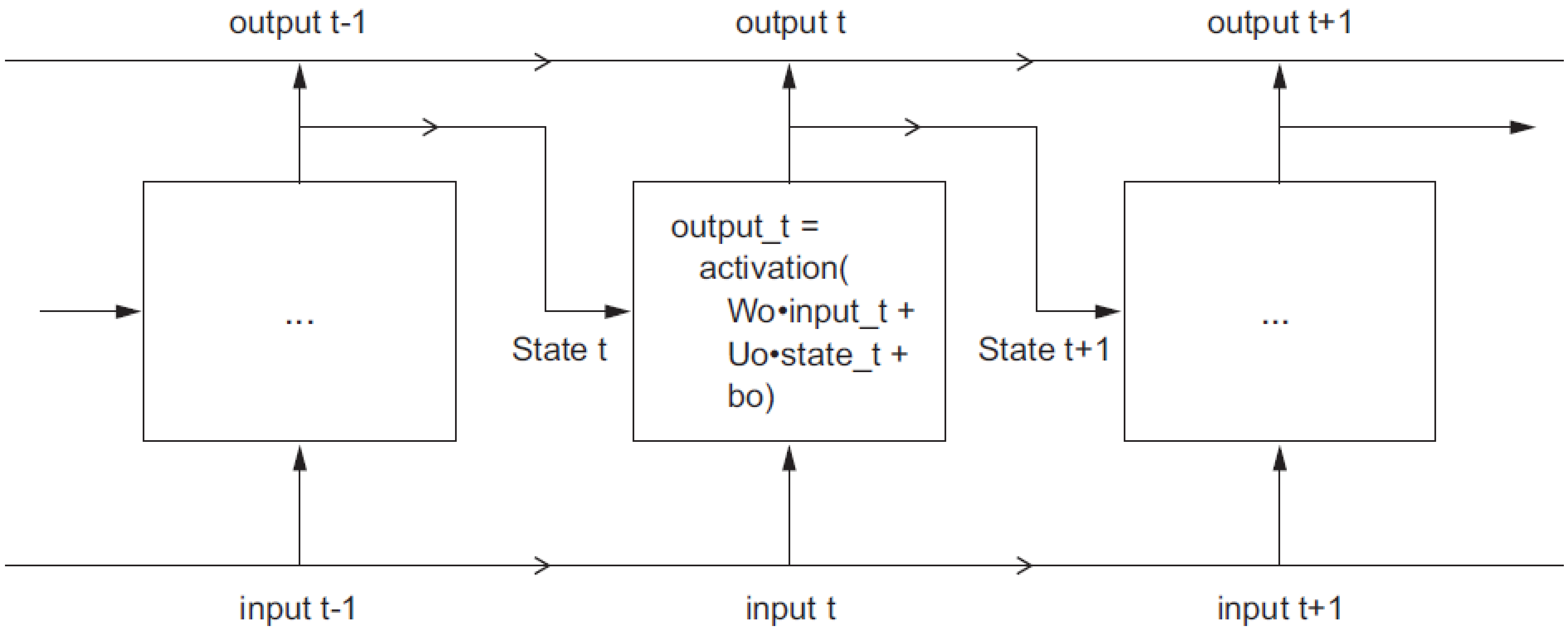
Loss and Accuracy



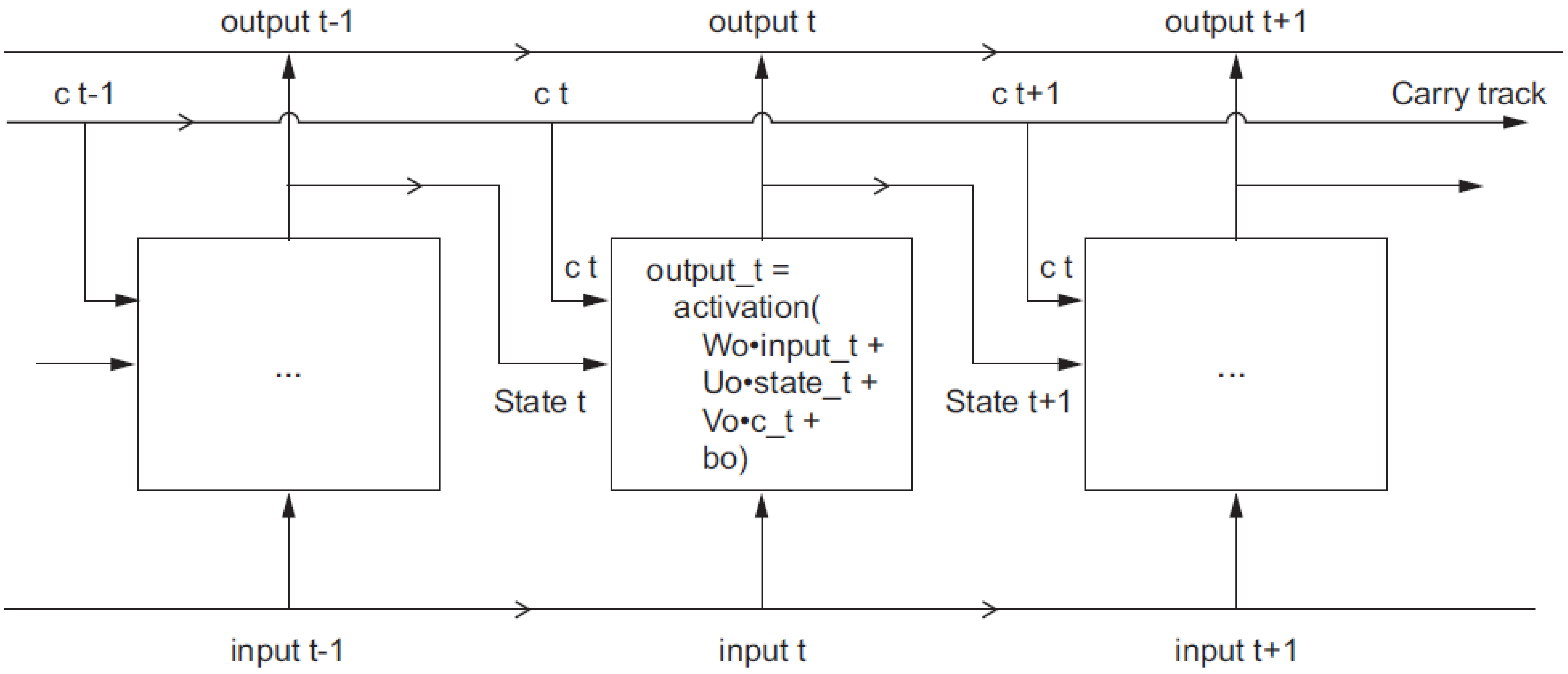


85% accuracy [compare to 88% accuracy in Chapter 3]

Visualizing a SimpleRNN



Going from SimpleRNN to LSTM: Adding a Carry Track



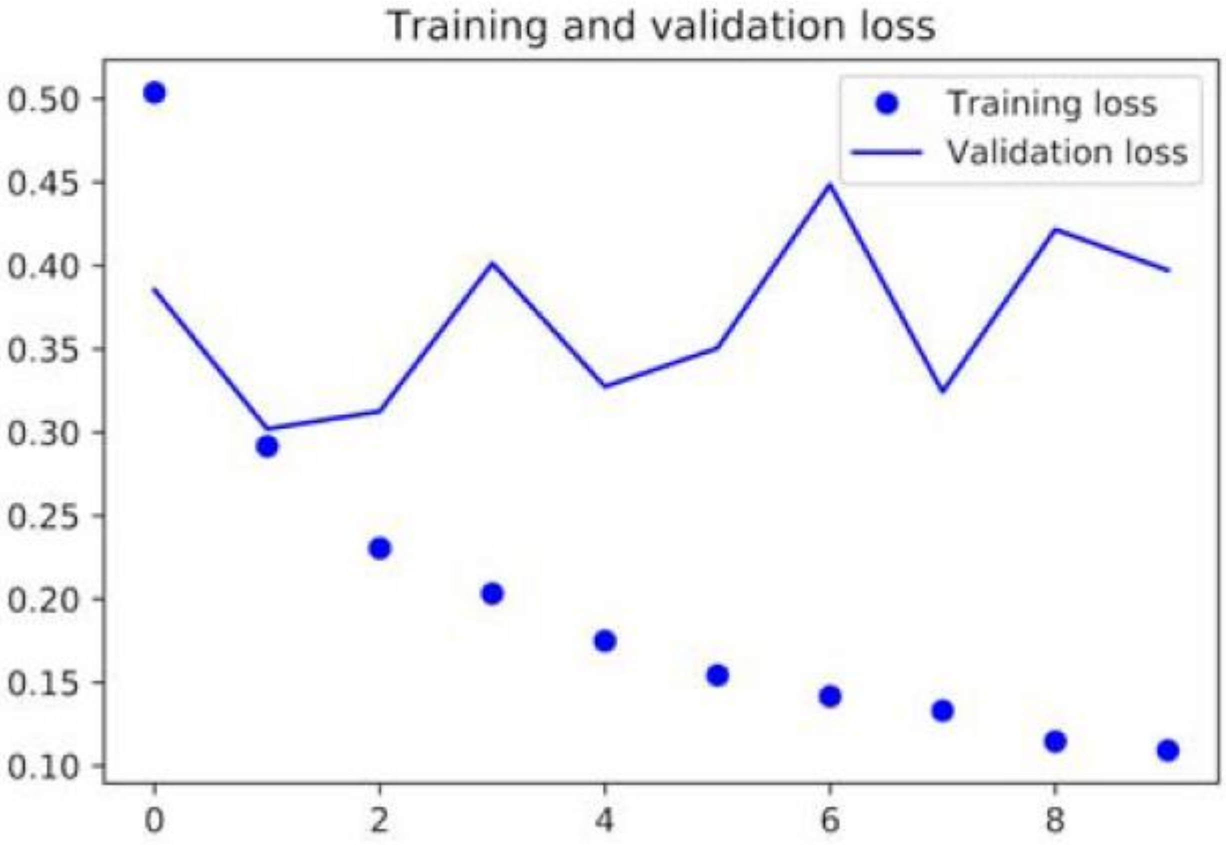
Pseudocode Details of the LSTM Architecture

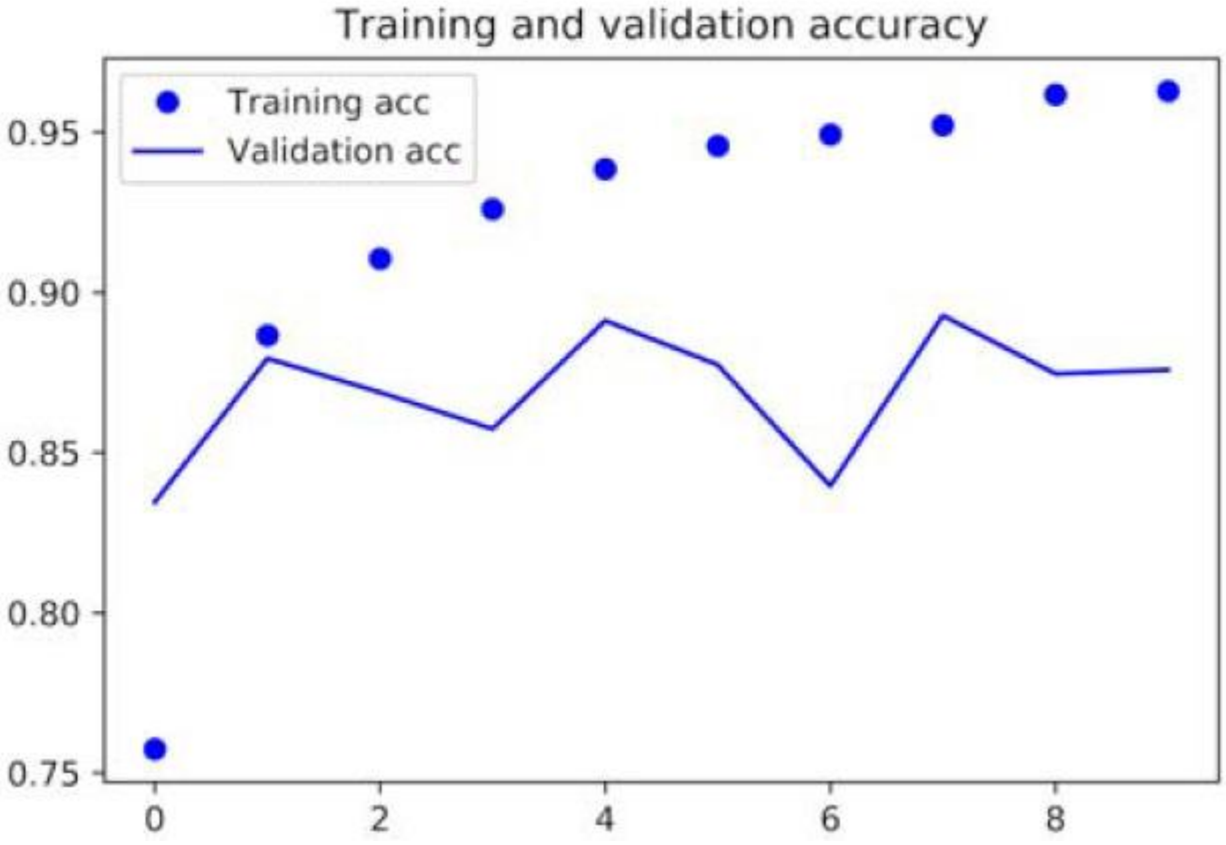
```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(C_t, Vo) + bo)

i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)

c_t+1 = i_t * k_t + c_t * f_t
```

Using the LSTM Layer in Keras

```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

Loss and Accuracy





89% Accuracy!

Wrapping Up

- What RNNs are and how they work
- What LSTM is, and why it works better on long sequences than a naive RNN
- How to use Keras RNN layers to process sequence data

Techniques

- Recurrent Dropout
- Stacking Recurrent Layers
- Bidirectional Recurrent Layers

Weather Data from Jena Germany

```
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
```

Inspecting the Jena Weather Data

420,551 lines; 15 columns

```
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:]

print(header)
print(len(lines))
```

Jena Weather Columns

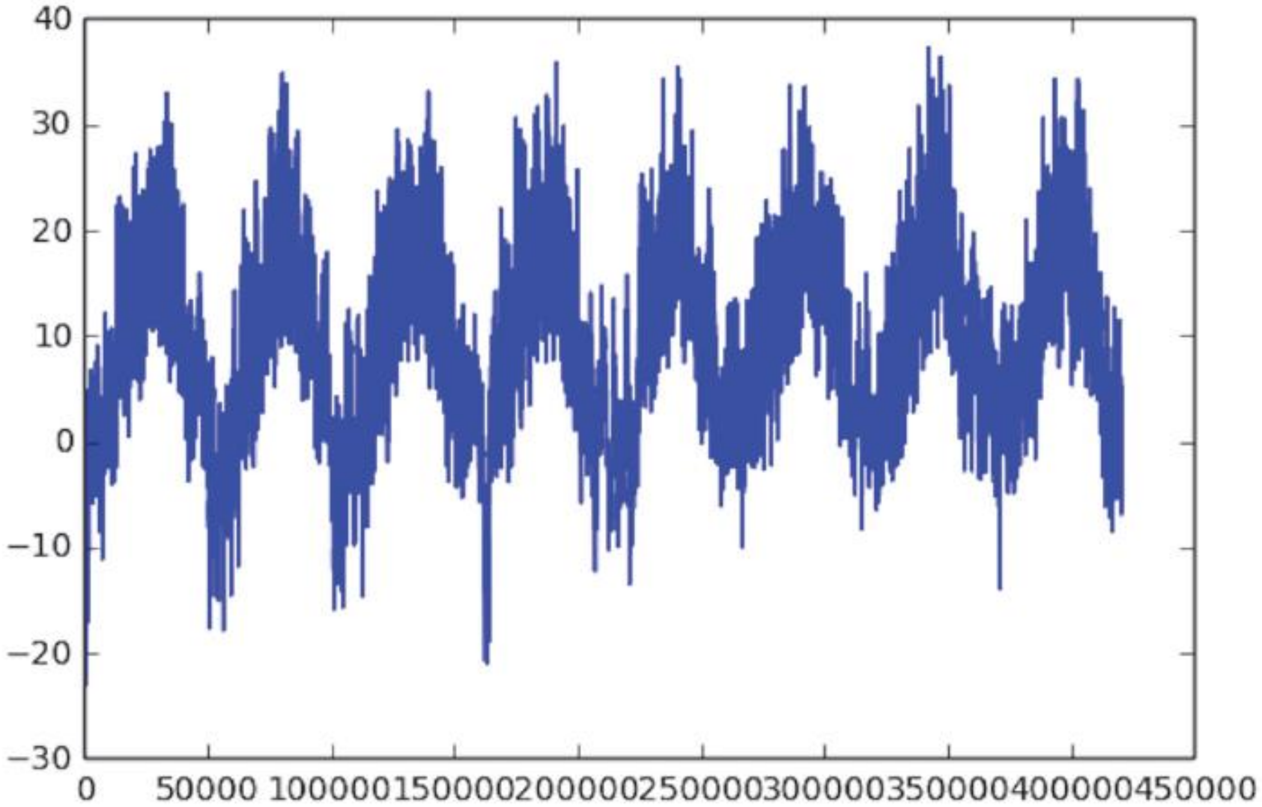
1. Date Time
2. p (mbar): atmospheric pressure in millibars
3. T (degC): temperature in degrees Celsius
4. Tpot (K): potential temperature (for reference pressure) on Kelvin scale
5. Tdew (degC): dewpoint temperature in degrees Celsius
6. rh (%): relative humidity
7. VPmax (mbar): maximum water vapor pressure
8. VPact (mbar): actual water vapor pressure
9. VPdef (mbar): water vapor pressure deficit
10. sh (g/kg): specific humidity
11. H2OC (mmol/mol): water vapor concentration
12. rho (g/m**3): air density
13. wv (m/s): wind velocity
14. max. wv (m/s): maximum wind velocity
15. wd (deg): wind direction

Parsing the Data

```
import numpy as np

float_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(',')[1:]]
    float_data[i, :] = values
```

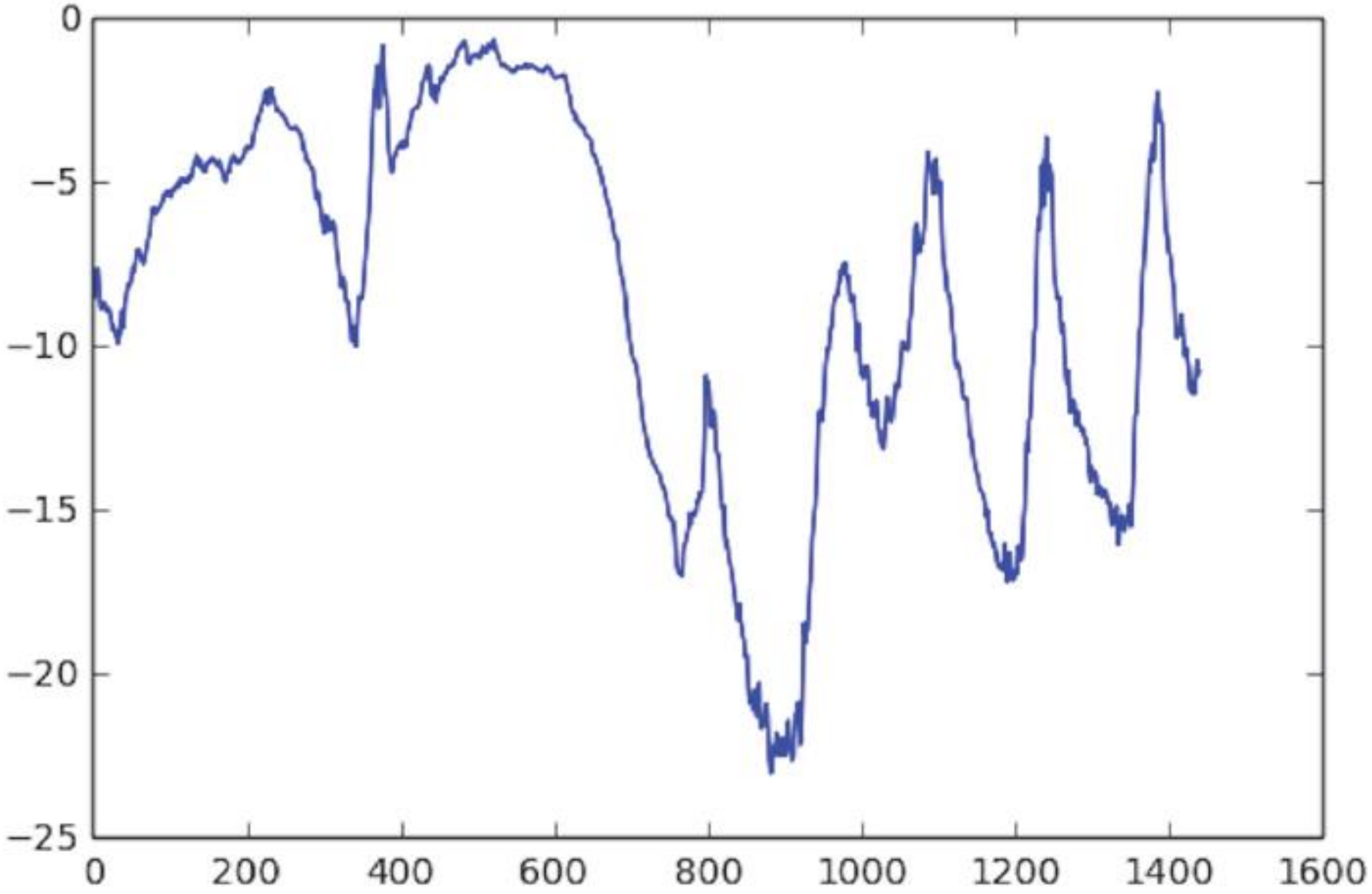
Plotting the Temperature Timeseries

```
from matplotlib import pyplot as plt

temp = float_data[:, 1]        <—— temperature (in degrees Celsius)
plt.plot(range(len(temp)), temp)
```



Plotting the First 10 Days of the Temperature Timeseries

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



Preparing the Data

- Given: 10 minutes between consecutive observations
- Parameters:
  - lookback = 720—Observations will go back 5 days
  - steps = 6—Observations will be sampled at one data point per hour
  - delay = 144—Targets will be 24 hours in the future
- Preprocess the data to a format a neural network can ingest: normalize each timeseries independently so that they all take small values on a similar scale
- Write a Python generator that takes the current array of float data and yields batches of data from the recent past, along with a target temperature in the future

Normalizing the Data

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

Data Generator

```
def generator(data, lookback, delay, min_index, max_index,
              shuffle=False, batch_size=128, step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(
                min_index + lookback, max_index, size=batch_size)
        else:
            if i + batch_size >= max_index:
                i = min_index + lookback
            rows = np.arange(i, min(i + batch_size, max_index))
            i += len(rows)

        samples = np.zeros((len(rows),
                            lookback // step,
                            data.shape[-1]))
        targets = np.zeros((len(rows),))
        for j, row in enumerate(rows):
            indices = range(rows[j] - lookback, rows[j], step)
            samples[j] = data[indices]
            targets[j] = data[rows[j] + delay][1]
        yield samples, targets
```

Data Generator Arguments

- `data`—The original array of floating-point data, which you normalized in listing 6.32.
- `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`—Indices in the `data` array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another for testing.
- `shuffle`—Whether to shuffle the samples or draw them in chronological order.
- `batch_size`—The number of samples per batch.
- `step`—The period, in timesteps, at which you sample data. You'll set it to 6 in order to draw one data point every hour.

Preparing the Train Data Generator

```
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)
```

Preparing the Val and Test Data Generators

```
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

test_steps = (len(float_data) - 300001 - lookback) // batch_size
```

How many steps to draw from val_gen in order to see the entire validation set

How many steps to draw from test_gen in order to see the entire test set

Mean Absoute Error (MAE) Evaluation Metric [and loss function!]

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

Estimating a Baseline [last temperature from observations]

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

evaluate_naive_method()
```

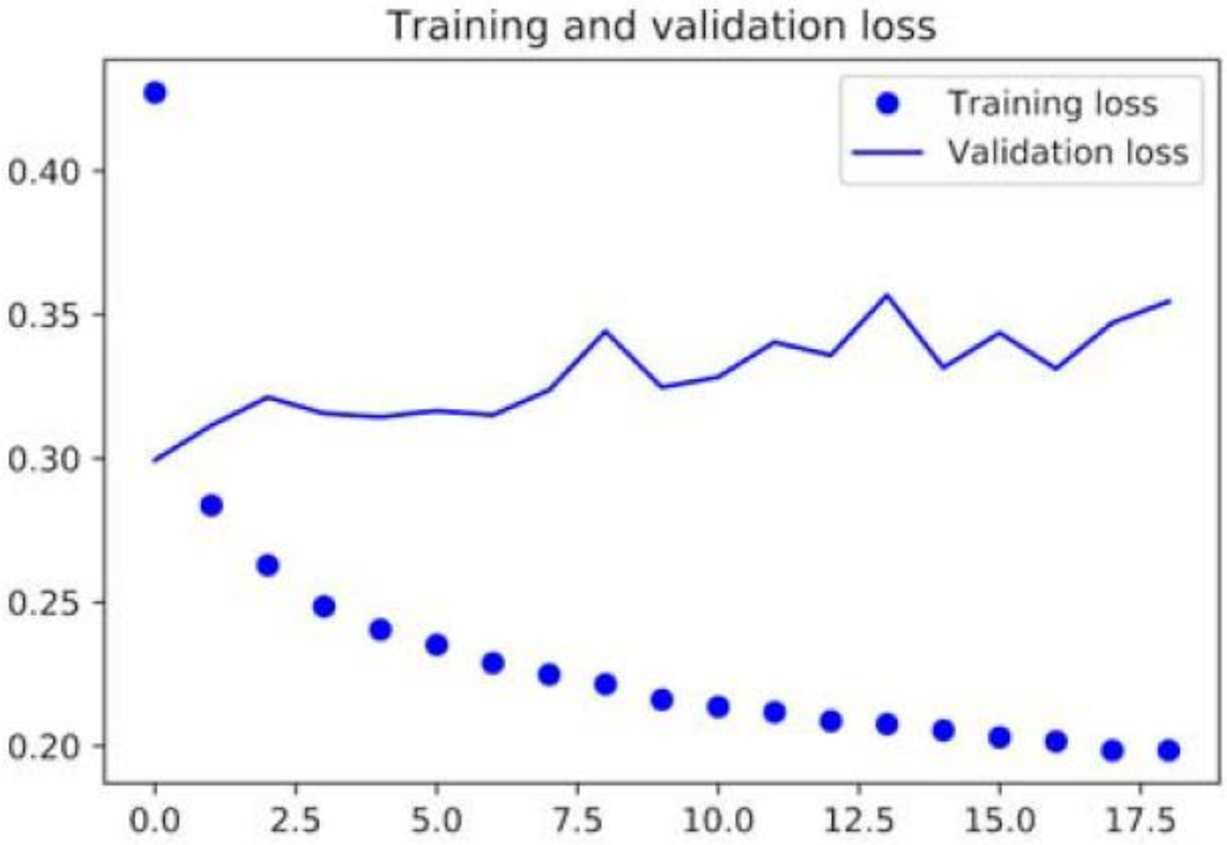
MAE of 0.29: celsius_mae = 0.29 * std[1] = 2.57 degrees Celsius

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])))
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,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

Training and Validation Loss (MAE)

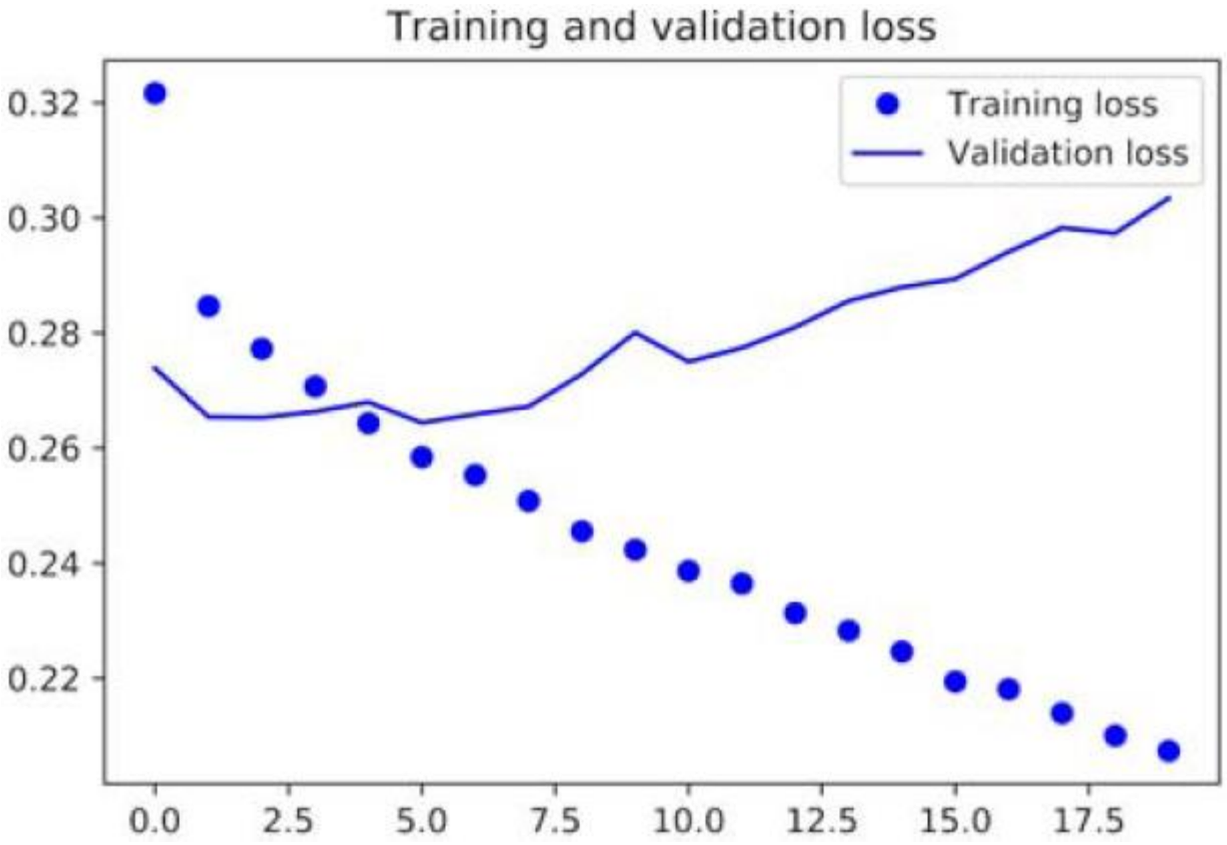


We are *not* beating the baseline

Training and Evaluating a GRU-Based Model

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

model = Sequential()
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)
```

Beating the Baseline!

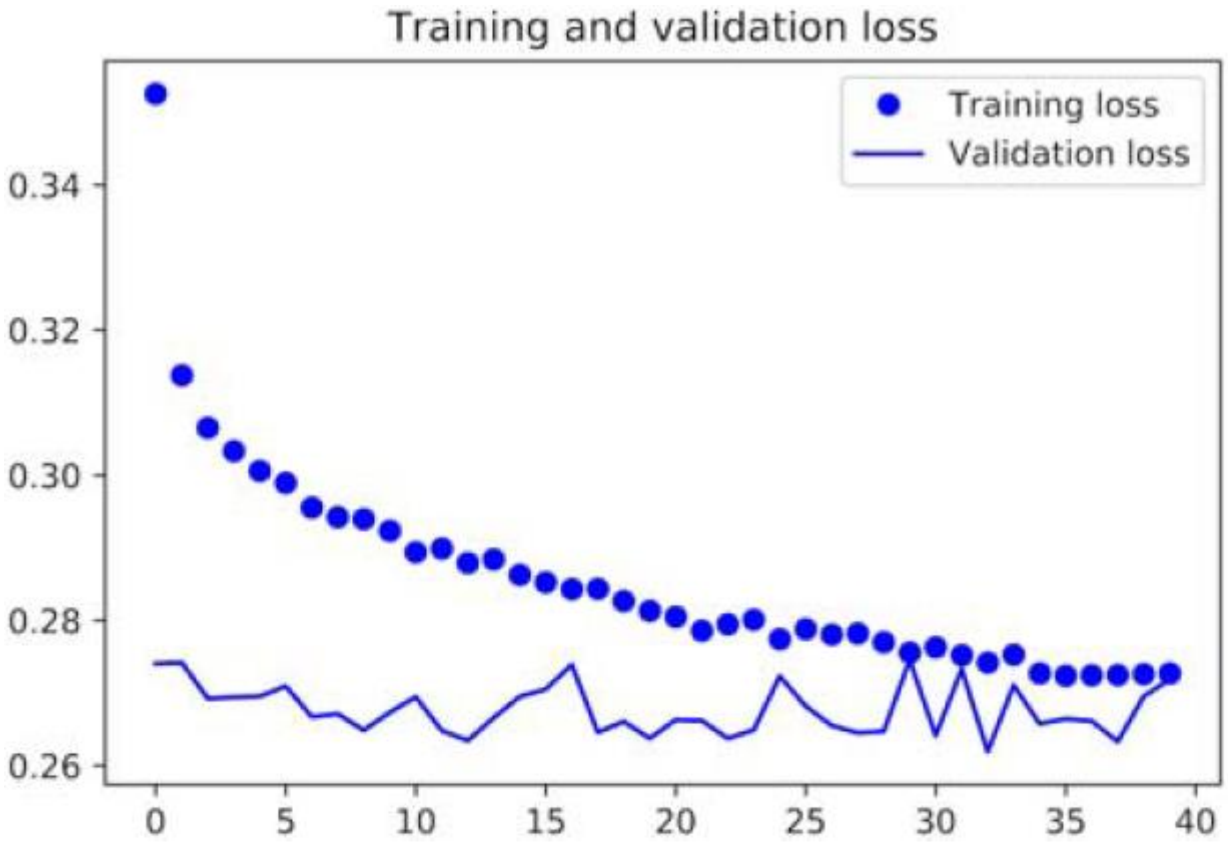


MAE around 0.265

Training and Evaluating a GRU-Based Model with Dropout

```
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)
```

Training and Validation Loss with Dropout

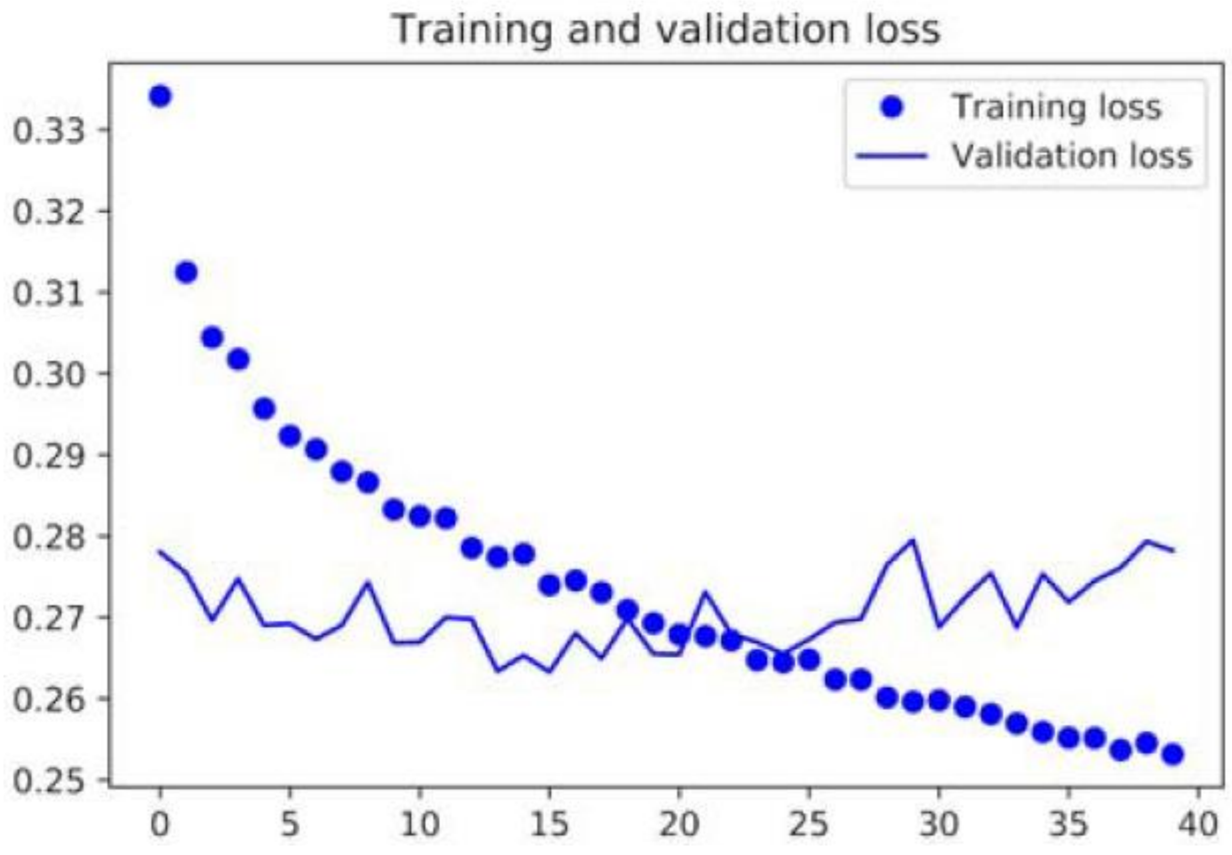


Not much better than before; but no longer overfitting

Training and Evaluating Stacked 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=0.1,
                     recurrent_dropout=0.5,
                     return_sequences=True,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
                     dropout=0.1,
                     recurrent_dropout=0.5))
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)
```

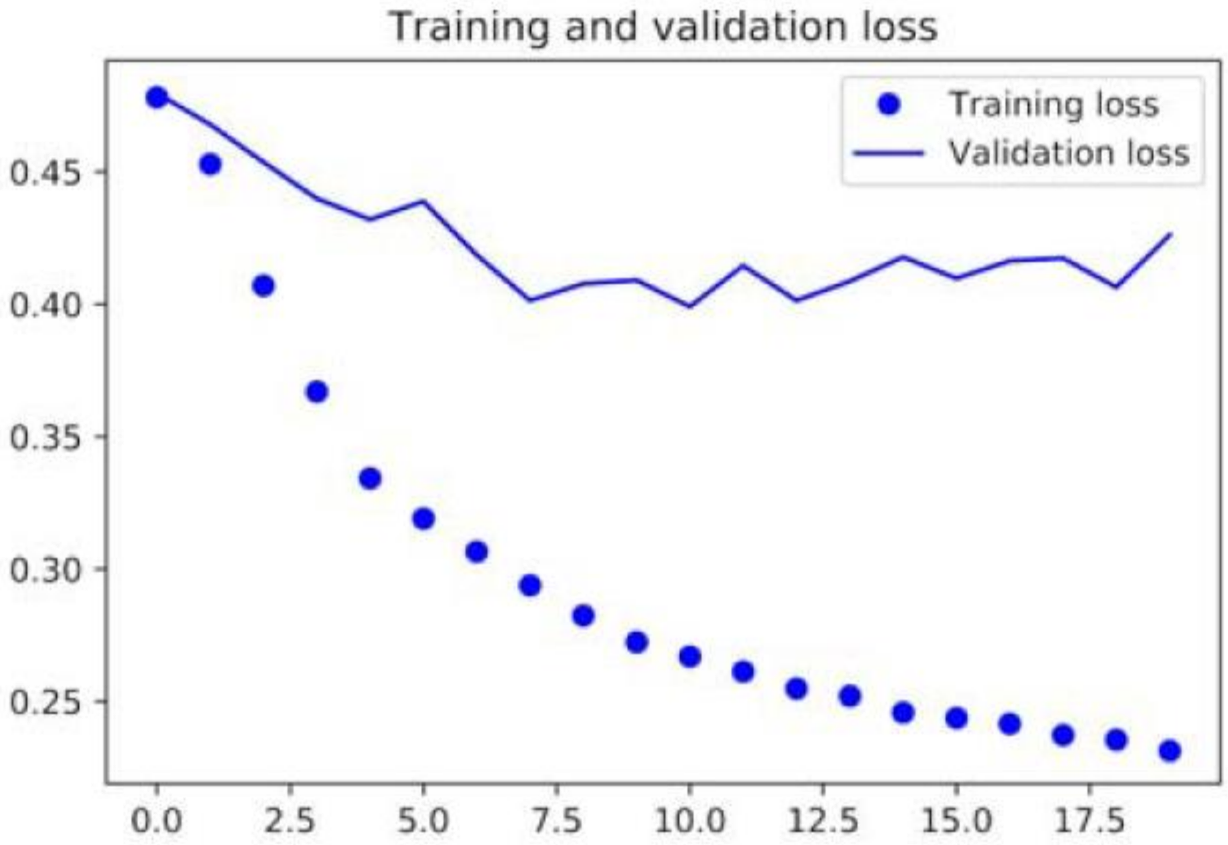
Training and Validation Loss for Stacked GRU-Based Model



Adding a layer did not help much: diminishing returns from increasing network capacity

Training and Validation Loss for Reversed Sequences using GRU Cell

Reversed order sequences underperform: last values processed by the GRU is the furthest away from the temperature prediction time



Training an LSTM Using Reversed Sequences

Nearly identical performance compared to LSTM with chronologically-ordered sequences

```
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras import layers
from keras.models import Sequential

max_features = 10000
maxlen = 500

(x_train, y_train), (x_test, y_test) = imdb.load_data(
    num_words=max_features)

x_train = [x[::-1] for x in x_train]
x_test = [x[::-1] for x in x_test]

x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)

model = Sequential()
model.add(layers.Embedding(max_features, 128))
model.add(layers.LSTM(32))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

Number of words to consider as features

Cuts off texts after this number of words (among the max_features most common words)

Loads data

Reverses sequences

Pads sequences

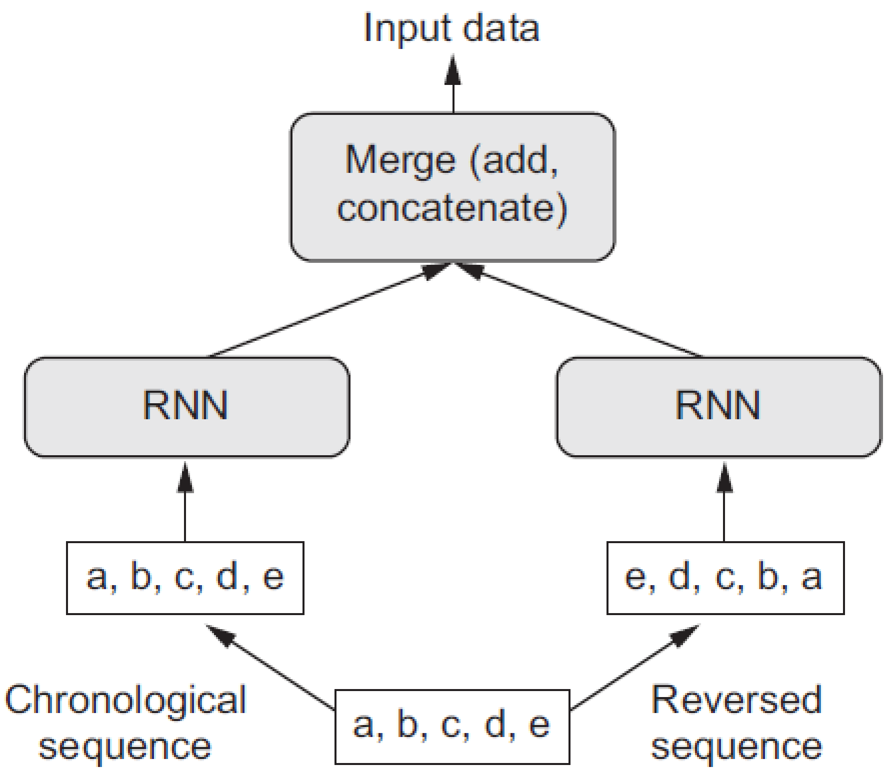
Bidirectional LSTM

Sometimes useful when applied to text

- Forward: tokens that come "before" are useful for understanding current token
- Backward: tokens that come "after" are useful for understanding current token

```
model.add(Bidirectional(LSTM(64)))    # creates 2 LSTM cells
```

Visualizing a Bidirectional RNN



Training a Bidirectional LSTM for IMDB

```
model = Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

Over 89% accuracy!

Training a Bidirectional GRU for Temperature Prediction

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

model = Sequential()
model.add(layers.Bidirectional(
    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=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

Performs about as well as the model with the forward GRU layer

Suggestions for Improving Temperature Predictions

- Adjust the number of units in each recurrent layer in the stacked setup. The current choices are largely arbitrary and thus probably suboptimal.
- Adjust the learning rate used by the RMSprop optimizer.
- Try using LSTM layers instead of GRU layers.
- Try using a bigger densely connected regressor on top of the recurrent layers: that is, a bigger Dense layer or even a stack of Dense layers.
- Don’t forget to eventually run the best-performing models (in terms of validation MAE) on the test set! Otherwise, you’ll develop architectures that are overfitting to the validation set.

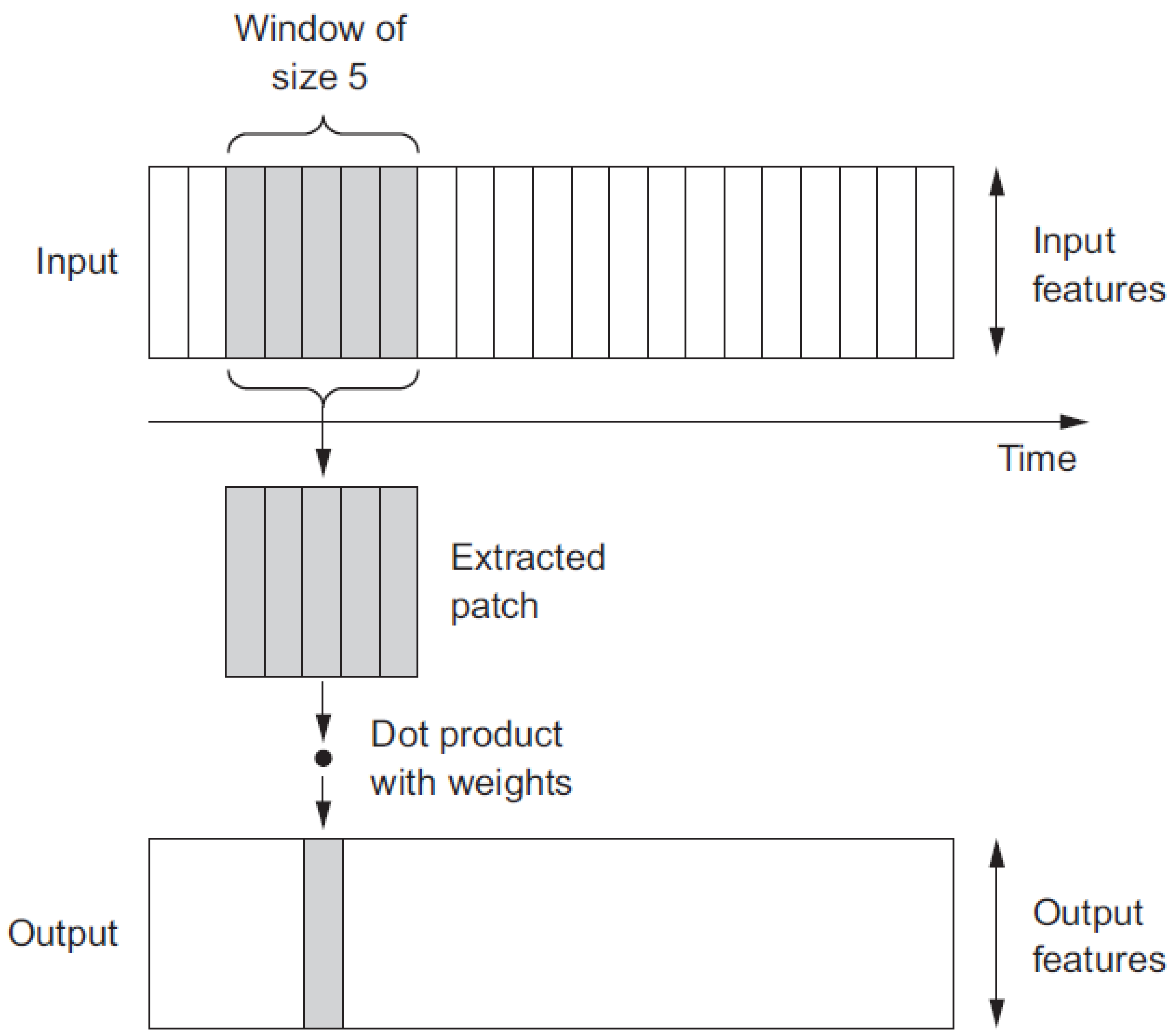
Wrapping Up

- When approaching a new problem, it's good to first establish common-sense baselines for your metric of choice. If you don't have a baseline to beat, you can't tell whether you're making real progress.
- Try simple models before expensive ones, to justify the additional expense. Sometimes a simple model will turn out to be your best option.
- When you have data where temporal ordering matters, recurrent networks are a great fit and easily outperform models that first flatten the temporal data.
- To use dropout with recurrent networks, you should use a time-constant dropout mask and recurrent dropout mask. These are built into Keras recurrent layers, so all you have to do is use the dropout and recurrent_dropout arguments of recurrent layers.
- Stacked RNNs provide more representational power than a single RNN layer. They're also much more expensive and thus not always worth it. Although they offer clear gains on complex problems (such as machine translation), they may not always be relevant to smaller, simpler problems.
- Bidirectional RNNs, which look at a sequence both ways, are useful on natural-language processing problems. But they aren't strong performers on sequence data where the recent past is much more informative than the beginning of the sequence.

Markets and Machine Learning

- Markets have *very different statistical characteristics* than natural phenomena such as weather patterns. Trying to use machine learning to beat markets, when you only have access to publicly available data, is a difficult endeavor, and you're likely to waste your time and resources with nothing to show for it.
- Always remember that when it comes to markets, past performance is *not* a good predictor of future returns—looking in the rear-view mirror is a bad way to drive. Machine learning, on the other hand, is applicable to datasets where the past *is* a good predictor of the future.

Visualizing 1D Convolution



Note the vertical arrows for input features

1D Pooling

- This is the 1D equivalent of the 2D versions
- Used for subsampling: giving access to the bigger picture [all pun intended]
- Common flavors include:
  - Max pooling
  - Average pooling

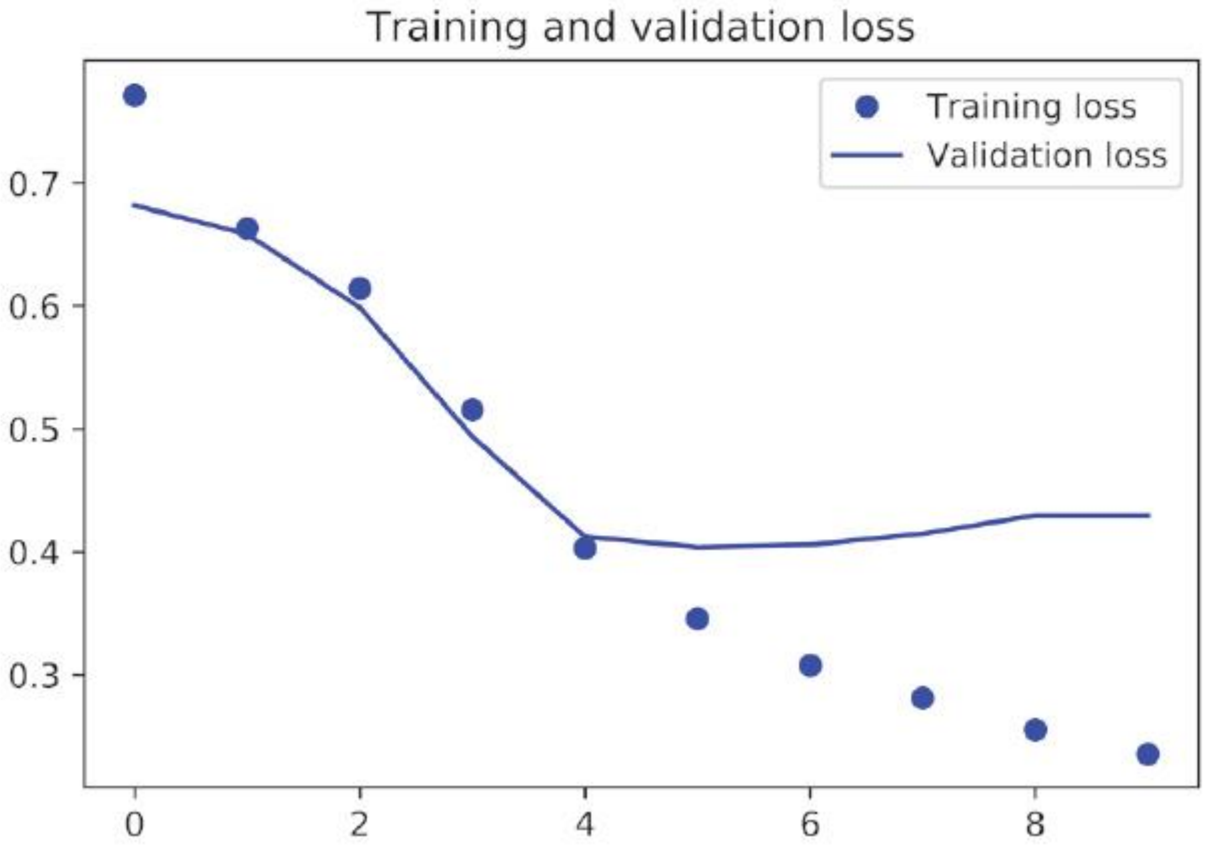
Preparing the IMDB Data

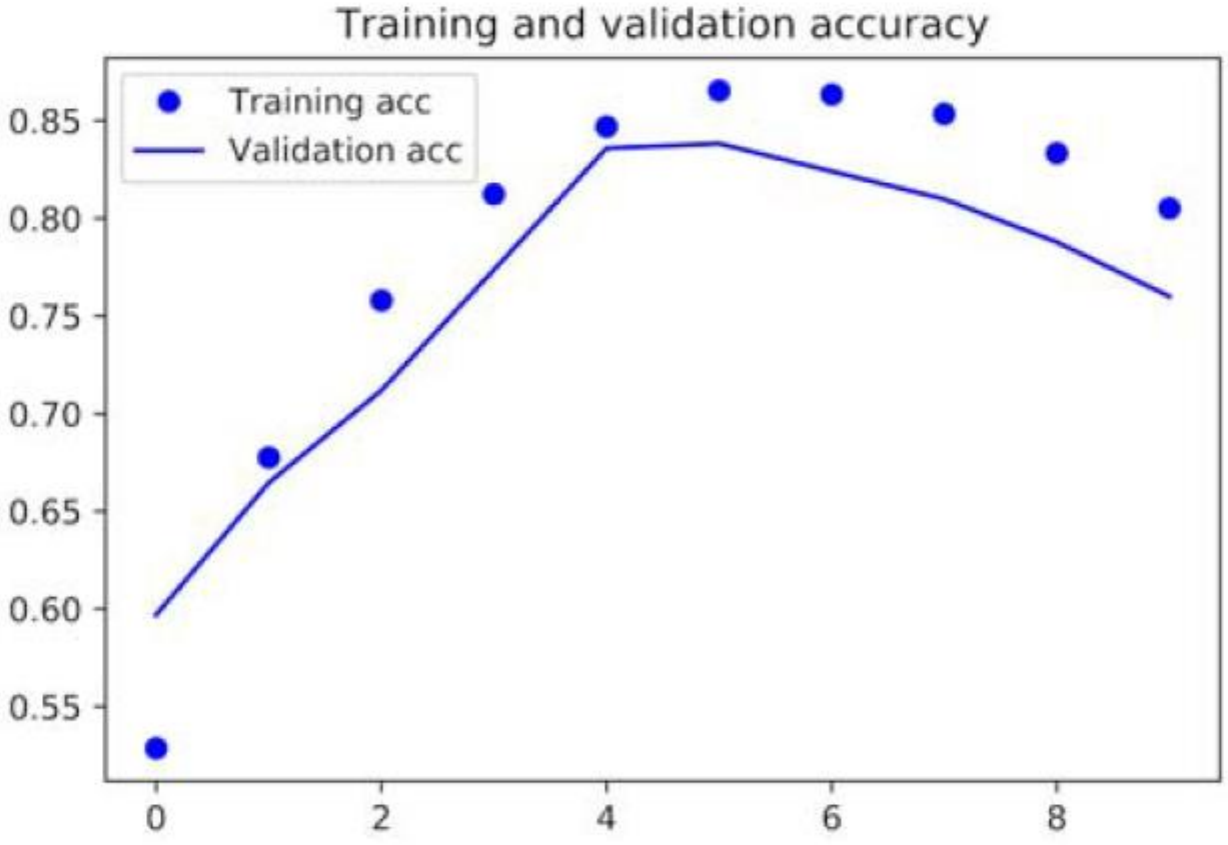
```
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 10000
max_len = 500

print('Loading data...')
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences')
print(len(x_test), 'test sequences')

print('Pad sequences (samples x time)')
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
```

Simple 1D ConvNet for the IMDB Data

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

model = Sequential()
model.add(layers.Embedding(max_features, 128, input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(lr=1e-4),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

Loss and Accuracy for the IMDB ConvNet



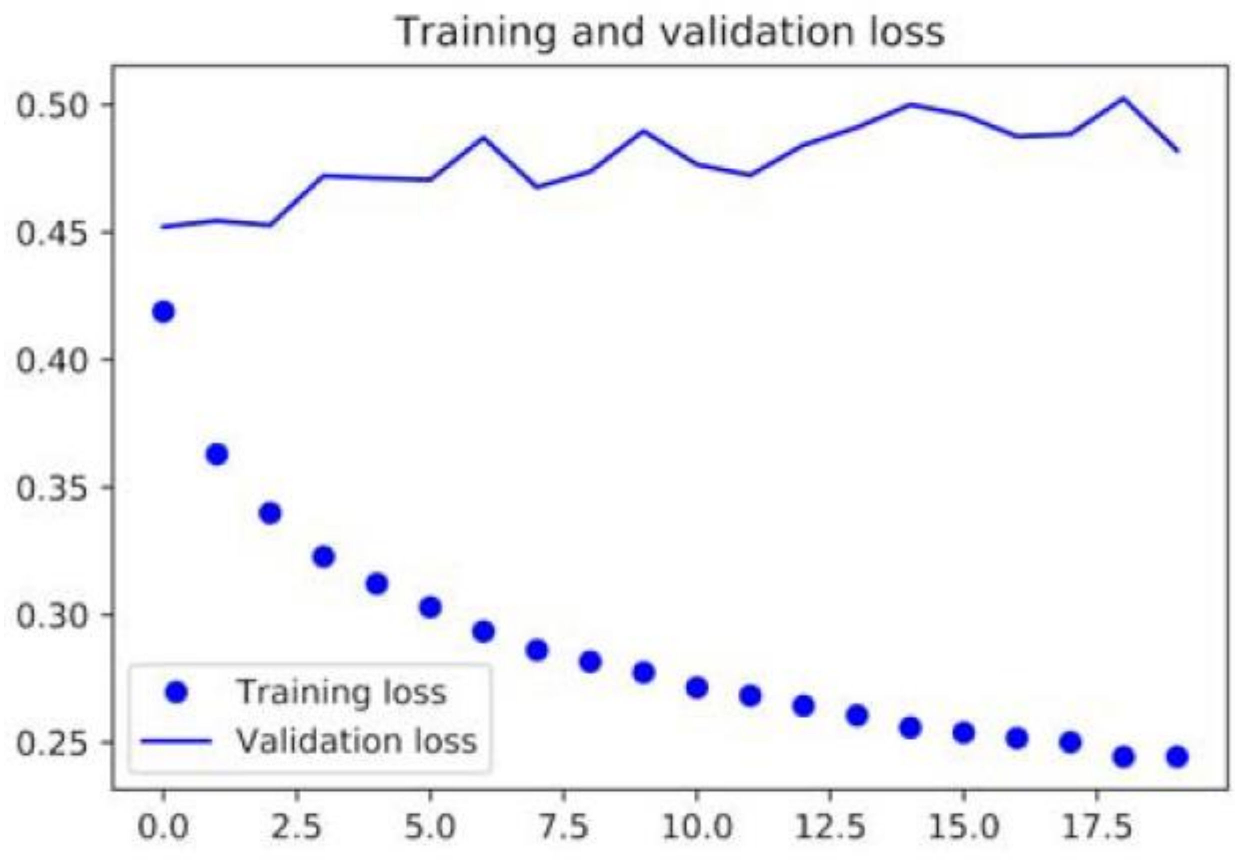


Accuracy is not as good as the LSTM, but it runs faster

Simple 1D ConvNet for the Jena Weather Data

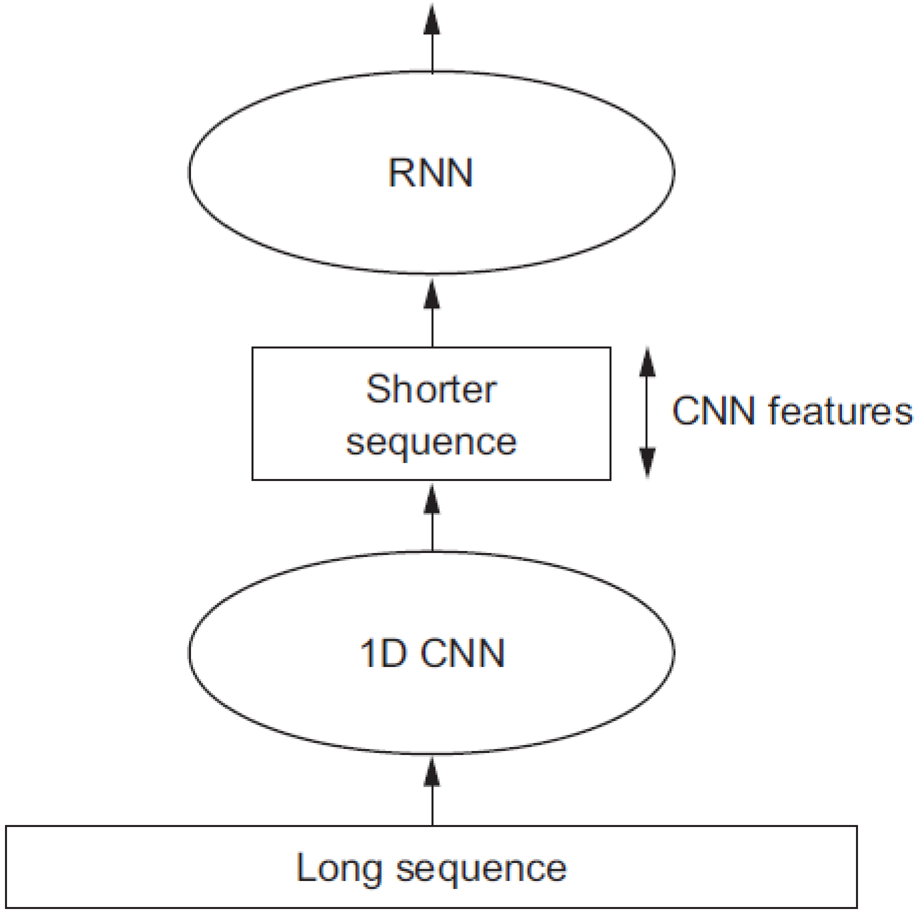
```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                        input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
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)
```

MAE Loss on the Jena Weather Data



Model has no knowledge of temporal position; e.g. toward the beginning, toward the end, etc.

Combining a 1D ConvNet and an RNN for Processing Sequences



Higher-Resolution Data Generators for the Jena Weather Data

```
step = 3
lookback = 720
delay = 144

train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=200000,
                      shuffle=True,
                      step=step)
val_gen = generator(float_data,
                    lookback=lookback,
                    delay=delay,
                    min_index=200001,
                    max_index=300000,
                    step=step)
test_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=300001,
                     max_index=None,
                     step=step)
val_steps = (300000 - 200001 - lookback) // 128
test_steps = (len(float_data) - 300001 - lookback) // 128
```

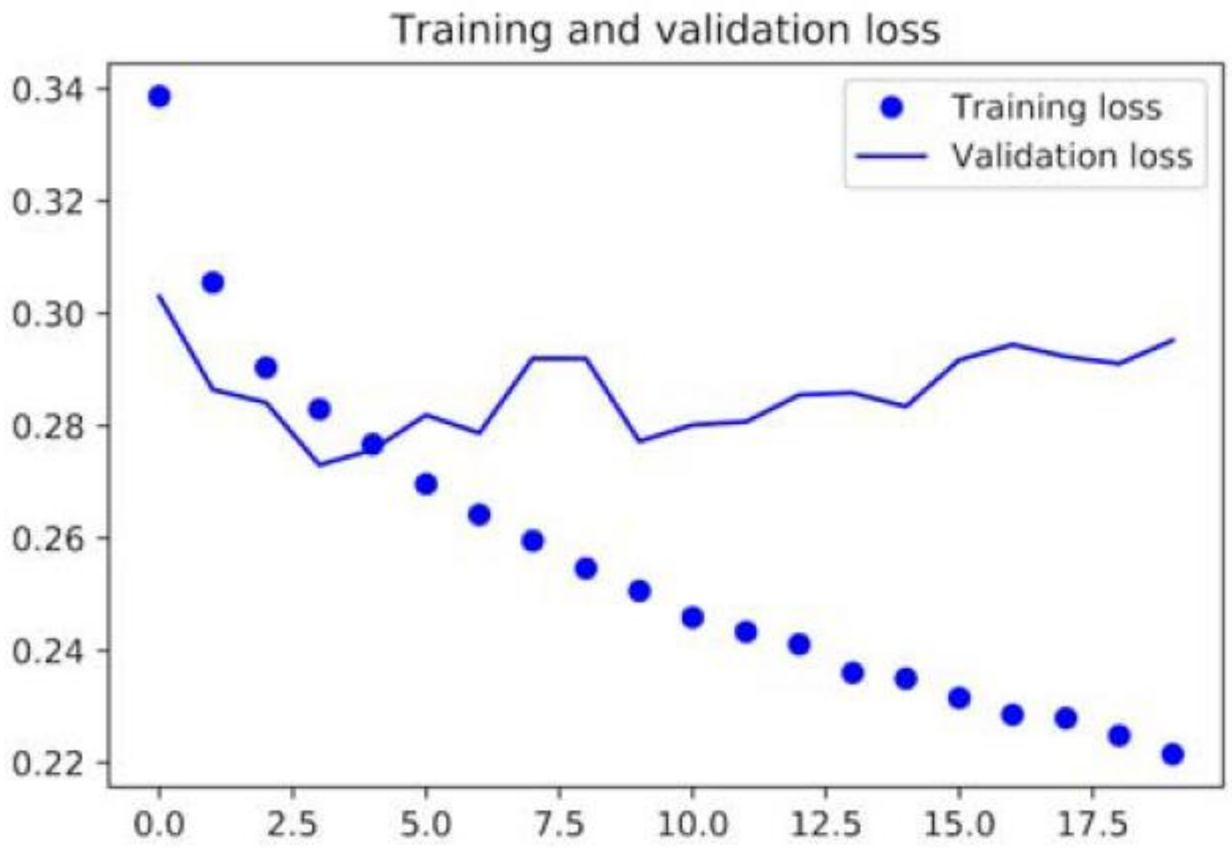
Previously set to 6 (1 point per hour); now 3 (1 point per 30 min)

Unchanged

1D ConvNet + GRU for the Jena Weather Data

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

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                        input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GRU(32, dropout=0.1, recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.summary()

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)
```

MAE Loss for the Jena Weather Data



Not as good as the GRU alone, but it's significantly faster

Wrapping Up

- In the same way that 2D convnets perform well for processing visual patterns in 2D space, 1D convnets perform well for processing temporal patterns. They offer a faster alternative to RNNs on some problems, in particular natural language processing tasks.
- Typically, 1D convnets are structured much like their 2D equivalents from the world of computer vision: they consist of stacks of Conv1D layers and Max-Pooling1D layers, ending in a global pooling operation or flattening operation.
- Because RNNs are extremely expensive for processing very long sequences, but 1D convnets are cheap, it can be a good idea to use a 1D convnet as a preprocessing step before an RNN, shortening the sequence and extracting useful representations for the RNN to process.

Chapter Summary: 1 of 2

- In this chapter, you learned the following techniques, which are widely applicable to any dataset of sequence data, from text to timeseries:
  - How to tokenize text
  - What word embeddings are, and how to use them
  - What recurrent networks are, and how to use them
  - How to stack RNN layers and use bidirectional RNNs to build more-powerful sequence-processing models
  - How to use 1D convnets for sequence processing
  - How to combine 1D convnets and RNNs to process long sequences
- You can use RNNs for timeseries regression ("predicting the future"), timeseries classification, anomaly detection in timeseries, and sequence labeling (such as identifying names or dates in sentences).

Chapter Summary: 2 of 2

- Similarly, you can use 1D convnets for machine translation (sequence-to-sequence convolutional models, like SliceNet[a]), document classification, and spelling correction.
- If *global order matters* in your sequence data, then it's preferable to use a recurrent network to process it. This is typically the case for timeseries, where the recent past is likely to be more informative than the distant past.
- If *global ordering isn't fundamentally meaningful*, then 1D convnets will turn out to work at least as well and are cheaper. This is often the case for text data, where a keyword found at the beginning of a sentence is just as meaningful as a keyword found at the end.

Parameter Counts for Recurrent Cells

- Number of Parameters =

  (1 + numGates)*recurrentCellSize*(previousLayerElementSize + recurrentCellSize + 1)

- numGates =
  - 0 for Simple Recurrent Neural Network (RNN) Cell
  - 2 for Gated Recurrent Unit (GRU) Cell
  - 3 for Long Short-Term Memory (LSTM) Cell
- "+ 1": assumes we're including a bias term for the cell's features
- Runtime Complexity: the cell is invoked for each element in a sequence and each sequence in a batch

Simple Recurrent Neural Network (RNN) Cell

- Under "class SimpleRNNCell", look for "def call"

```
h = backend.dot(inputs, self.kernel)
h = backend.bias_add(h, self.bias)
output = h + backend.dot(prev_output, self.recurrent_kernel)
output = self.activation(output)
```

- For each sequence position: features for previous output added to features for current input

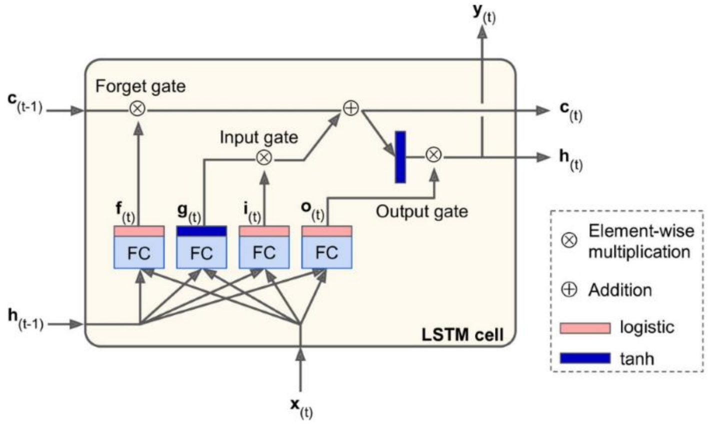
  2 weight matrices and 1 bias vector [same 2 weight matrices and 1 bias vector used for all positions in the sequence]

- Note: It's possible to simply concatenate the inputs and prev_output, so we could have 1 weight matrix

https://github.com/tensorflow/tensorflow/blob/v2.5.0/tensorflow/python/keras/layers/recurrent.py

source for image: http://shop.oreilly.com/product/0636920052289.do

Long Short-Term Memory (LSTM) Cell

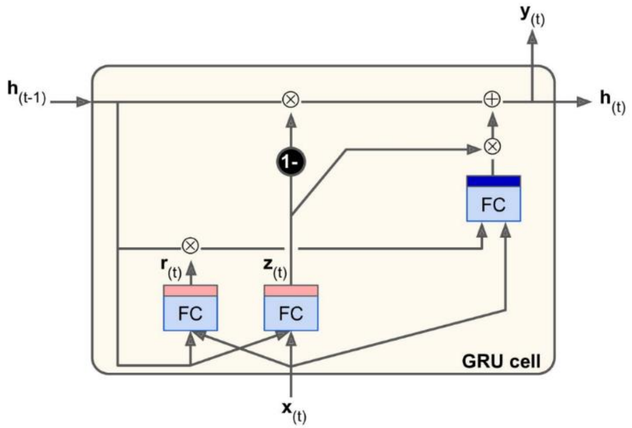



```
z = backend.dot(inputs, self.kernel)
z += backend.dot(h_tm1, self.recurrent_kernel)
z = backend.bias_add(z, self.bias)
z0, z1, z2, z3 = array_ops.split(z, num_or_size_splits=4, axis=1)
i = self.recurrent_activation(z0)
f = self.recurrent_activation(z1)
c = f * c_tm1 + i * self.activation(z2)    # context ‘c’
o = self.recurrent_activation(z3)    # hidden output ‘h’
# recurrent activation: sigmoid
# activation: tanh
```

https://github.com/tensorflow/tensorflow/blob/v2.5.0/tensorflow/python/keras/layers/recurrent.py

source for image: http://shop.oreilly.com/product/0636920052289.do (super close to reflecting source code: “1-”)

Gated Recurrent Unit (GRU) Cell




```
matrix_x = backend.dot(inputs, self.kernel)
matrix_x = backend.bias_add(matrix_x, input_bias)
x_z, x_r, x_h = array_ops.split(matrix_x, 3, axis=-1)
matrix_inner = backend.dot(h_tm1, self.recurrent_kernel[:, :2 * self.units])
recurrent_z, recurrent_r, recurrent_h = array_ops.split(matrix_inner, [self.units,
self.units, -1], axis=-1)
z = self.recurrent_activation(x_z + recurrent_z)
r = self.recurrent_activation(x_r + recurrent_r)
recurrent_h = backend.dot(r * h_tm1, self.recurrent_kernel[:, 2 * self.units:])
hh = self.activation(x_h + recurrent_h)
# previous and candidate state mixed by update gate
h = z * h_tm1 + (1 - z) * hh
```

https://github.com/tensorflow/tensorflow/blob/v2.5.0/tensorflow/python/keras/layers/recurrent.py