Chapter 3: Getting started with neural networks

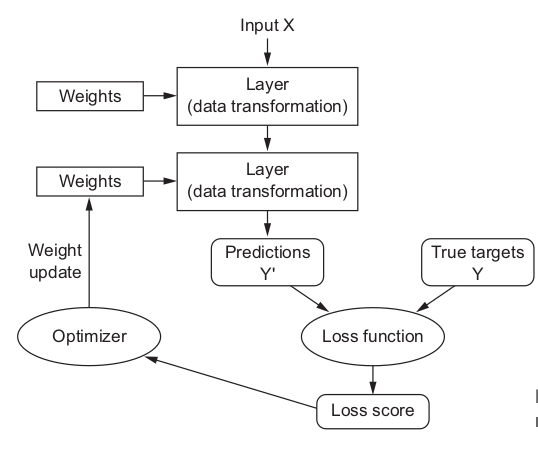
1. Components of NN

 **Layers**, which are combined into a network (or **model**)

 The **input data** and corresponding **targets**

 The **loss function**, which defines the feedback signal used for learning

 The **optimizer**, which determines how learning proceeds



2. **Network’s knowledge** contains **weight** of the **layers.** Different layers will have different tensor format for different type of data.

Ex: 2D tensors is often stored in **densely connected** layers (or fully connected)

3D tensor (sequence Data) (samples, timesteps, features) is typically processed by **recurrent** layers such as **LSTM layer**

4D tensors (image data…) is processed by **2D convolution** layer (**Conv2D)**

3. Building model is Keras is clipping together compatible layers to form useful data-transfomation pipelines.

Or **layer compatibility** => compatible input and output tensors.

layer = layers.Dense(32, input\_shape=(784,)) #**A dense layer with 32 output units**

**#Input: 2D tensor (first dimension: 784)**

**#Output: tensor (first dimension: 32)**

from keras import models  
from keras import layers  
model = models.Sequential()  
model.add(layers.Dense(32, input\_shape=(784,)))  
model.add(layers.Dense(32))  
The second layer didn’t receive an input shape argument—instead, it automatically  
inferred its input shape as being the output shape of the layer that came before.

4. **Models: networks of layers**

Most common: linear stack of layers (mapping 1 input to single output)

More: - Two-branch network

* Multihead networks
* Inception block

5. **Hypothesis space:** by choosing a network topology, you constrain your **space of possibilities (hypothesis space)** => then the jobs left is searching for good set of weights.

YUP

Picking the right network architecture is more an art than a science; and although  
there are some best practices and principles you can rely on, only practice can help  
you become a proper neural-network architect. The next few chapters will both teach  
you explicit principles for building neural networks and help you develop intuition as  
to what works or doesn’t work for specific problems.

6. Once the network architecture is defined, you still have to choose two more things:

VERY IMPORTANT!!!  
 ***Loss function (objective function)*—**The quantity that will be minimized during  
training. It represents a measure of success for the task at hand.  
 ***Optimize****r*—Determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD)

NN with multiple outputs will have multiple loss functions (one per output), but gradient-descent process is based on just 1 single scalar loss value => combine all loss value.

7. Fortunately, when it comes to common problems such as classification, regression,  
and sequence prediction, there are simple guidelines you can follow to choose the  
correct loss. For instance, you’ll use binary crossentropy for a two-class classification  
problem, categorical crossentropy for a many-class classification problem, meansquared error for a regression problem, connectionist temporal classification (CTC)  
for a sequence-learning problem, and so on. Only when you’re working on truly new  
research problems will you have to develop your own objective functions. In the next  
few chapters, we’ll detail explicitly which loss functions to choose for a wide range of  
common tasks.

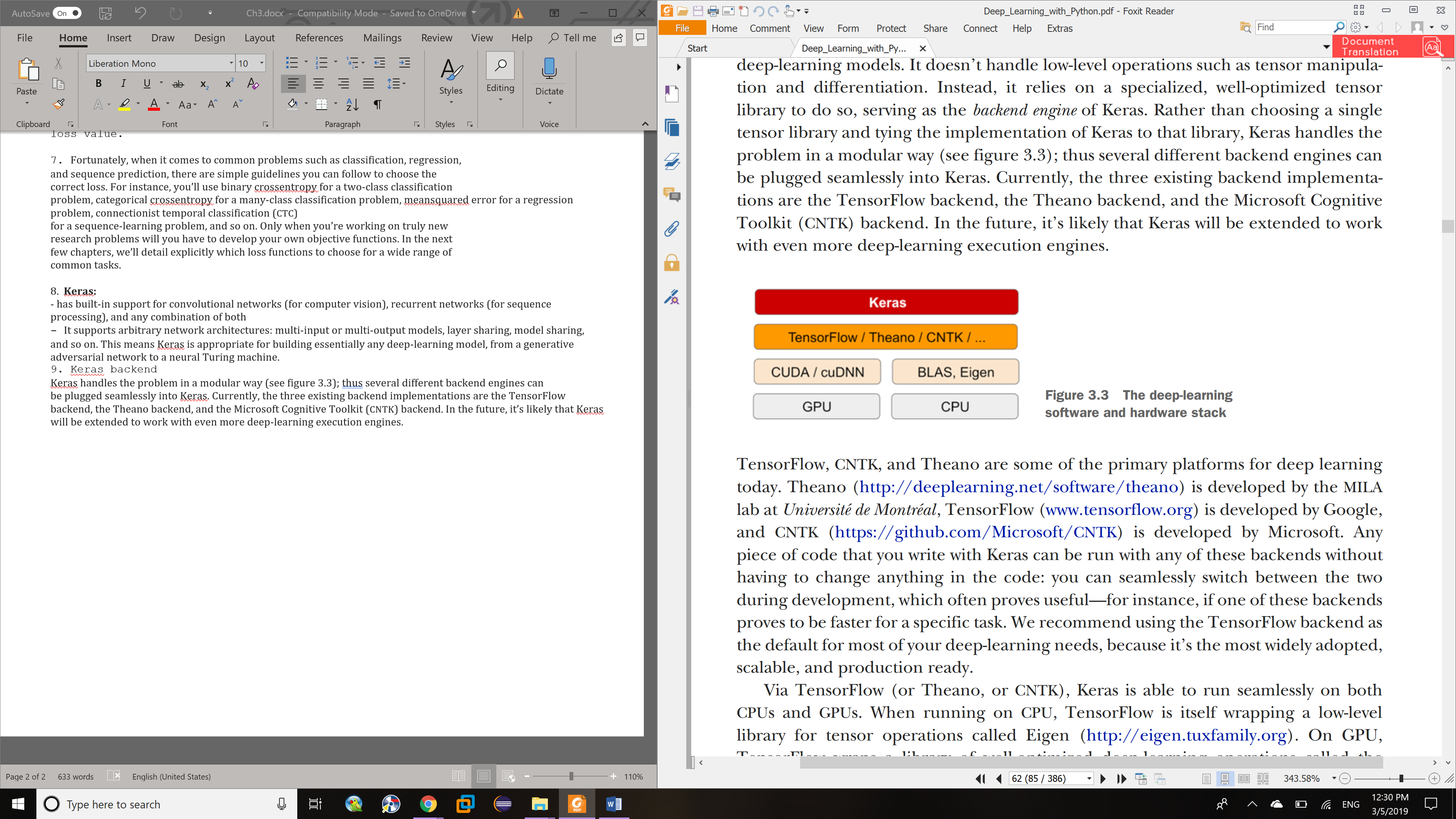
8. **Keras:**

- has built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both

- It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, and so on. This means Keras is appropriate for building essentially any deep-learning model, from a generative adversarial network to a neural Turing machine.

9. Keras backend

Keras handles the problem in a modular way (see figure 3.3); thus several different backend engines can  
be plugged seamlessly into Keras. Currently, the three existing backend implementations are the TensorFlow backend, the Theano backend, and the Microsoft Cognitive Toolkit (CNTK) backend. In the future, it’s likely that Keras will be extended to work with even more deep-learning execution engines.



10. **Workflow**

1. Define your training data: input tensors and target tensors.  
2. Define a network of layers (or *model* ) that maps your inputs to your targets.3 Configure the learning process by choosing a loss function, an optimizer, and  
some metrics to monitor.  
4 Iterate on your training data by calling the fit() method of your model

11. **2 ways of defining a model:**

**- Sequential** class (linear stack of layers)

from keras import models

from keras import layers

model = models.Sequential()

model.add(layers.Dense(32, activation='relu', input\_shape=(784,)))

model.add(layers.Dense(10, activation='softmax'))

12. **Functional API** (for directed acyclic graphs of layers, which lets you build completely arbitrary architectures)

input\_tensor = layers.Input(shape=(784,))

x = layers.Dense(32, activation='relu')(input\_tensor)

output\_tensor = layers.Dense(10, activation='softmax')(x)

model = models.Model(inputs=input\_tensor, outputs=output\_tensor)

Here’s an example with a single loss function, which is by far the most common case:

from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),

loss='mse',

metrics=['accuracy'])

model.fit(input\_tensor, target\_tensor, batch\_size=128, epochs=10) #Input,output is numpy array

**Example IMDB:**

**Preparing data**

*from keras.datasets import imdb*

*(train\_data, train\_labels), (test\_data, test\_labels) = imdb.load\_data(*

*num\_words=10000)*

*>>> train\_data[0]*

*[1, 14, 22, 16, ... 178, 32]*

*>>> train\_labels[0]*

*1*

*>>> max([max(sequence) for sequence in train\_data])*

*9999*

**#decode reviews back to English words**

*word\_index = imdb.get\_word\_index()* **#dictionary**

*reverse\_word\_index = dict([(value, key) for (key, value) in word\_index.items()])* **#reverse the dict key and value**

*decoded\_review = ' '.join([reverse\_word\_index.get(i - 3, '?') for i in train\_data[0]])*

**Have to turn your lists into tensors. There are two ways to do that:**

1. Pad your lists so that they all have the same length, turn them into an integer

tensor of shape (samples, word\_indices) , and then use as the first layer in

your network a layer capable of handling such integer tensors (the Embedding

layer, which we’ll cover in detail later in the book).

2. One-hot encode your lists to turn them into vectors of 0s and 1s. This would

mean, for instance, turning the sequence [3, 5] into a 10,000-dimensional vec-

tor that would be all 0s except for indices 3 and 5, which would be 1s. Then you

could use as the first layer in your network a Dense layer, capable of handling

floating-point vector data.

Way2:

**Encoding the integer sequences into a binary matrix**

*import numpy as np*

*def vectorize\_sequences(sequences, dimension=10000):*

*results = np.zeros((len(sequences), dimension))*

*for i, sequence in enumerate(sequences):*

*results[i, sequence] = 1.*

*return results*

*x\_train = vectorize\_sequences(train\_data)*

*x\_test = vectorize\_sequences(test\_data)*

*#vectorize the labels*

*y\_train = np.asarray(train\_labels).astype('float32')*

*y\_test = np.asarray(test\_labels).astype('float32')*

**The input data is vectors, and the labels are scalars (1s and 0s): this is the easiest setup**

**you’ll ever encounter. A type of network that performs well on such a problem is**

**a simple stack of fully connected ( Dense ) layers with relu activations: Dense(16,**

**activation='relu') .**

13. **Hidden unit: arguments in Dense(16), is a dimensionn in the representation space of the l**

**14. There are two key architecture decisions to be made about such a stack of Dense layers:**

** How many layers to use**

** How many hidden units to choose for each layer**

**15.** Sigmoid activation: output a probability (0 => 1)

*model = models.Sequential()*

*model.add(layers.Dense(16, activation='relu', input\_shape=(10000,)))*

*model.add(layers.Dense(16, activation='relu'))*

*model.add(layers.Dense(1, activation='sigmoid'))*

16. **binary\_crossentropy:** loss function that output probabilities. **Crossentropy is a quantity from the field of Information Theory** that measure distance between probability distribution, bw ground-truth distribution and your prediction.

*model.compile(optimizer='rmsprop',*

*loss='binary\_crossentropy',*

*metrics=['accuracy'])*

17. **Validating your approach:** setting apart samples from original training data.

Ex: x\_val = x\_train[:10000]

partial\_x\_train = x\_train[10000:]

#same with y

18. **Epochs : iterations over all samples in training tensors**

**19. the training loss decreases with every epoch, and the training accuracy**

**increases with every epoch. That’s what you would expect when running gradient-**

**descent optimization—the quantity you’re trying to minimize should be less with**

**every iteration**

**20. SUMMARY**

**- You usually need to do quite a bit of preprocessing on your raw data in order to be able to feed it—as tensors—into a neural network. Sequences of words can**

**be encoded as binary vectors, but there are other encoding options, too.**

**- Stacks of Dense layers with relu activations can solve a wide range of problems**

**(including sentiment classification), and you’ll likely use them frequently.**

**- In a binary classification problem (two output classes), your network should**

**end with a Dense layer with one unit and a sigmoid activation: the output of**

**your network should be a scalar between 0 and 1, encoding a probability.**

**- With such a scalar sigmoid output on a binary classification problem, the loss**

**function you should use is binary\_crossentropy .**

**- The rmsprop optimizer is generally a good enough choice, whatever your prob-**

**lem. That’s one less thing for you to worry about.**

**- As they get better on their training data, neural networks eventually start over-**

**fitting and end up obtaining increasingly worse results on data they’ve never**

**seen before. Be sure to always monitor performance on data that is outside of**

**the training set.**

**21. Single/multi-label, multiclass classification**

**22. One-hot encoding (categorical encoding)**

***def to\_one\_hot(labels, dimension=46):***

***results = np.zeros((len(labels), dimension))***

***for i, label in enumerate(labels):***

***results[i, label] = 1.***

***return results***

***one\_hot\_train\_labels = to\_one\_hot(train\_labels)***

***one\_hot\_test\_labels = to\_one\_hot(test\_labels)***

***model.add(layers.Dense(64, activation='relu', input\_shape=(10000,)))***

***model.add(layers.Dense(64, activation='relu'))***

***model.add(layers.Dense(46, activation='softmax'))***

***23.* Softmax** activation: network will output a probability distribution over 46 different output class.

24. **categorical\_crossentropy:** measures the distance between two probability distributions (between the prediction and real labels)

SUMMARY OF NEW WIRES

Here’s what you should take away from this example:

- If you’re trying to classify data points among N classes, your network should end

with a Dense layer of size N .

- In a single-label, multiclass classification problem, your network should end

with a softmax activation so that it will output a probability distribution over the N output classes.

- Categorical crossentropy is almost always the loss function you should use for

such problems. It minimizes the distance between the probability distributions

output by the network and the true distribution of the targets.

- There are two ways to handle labels in multiclass classification:

– Encoding the labels via categorical encoding (also known as one-hot encod-

ing) and using categorical\_crossentropy as a loss function

– Encoding the labels as integers and using the sparse\_categorical\_crossentropy

loss function

- If you need to classify data into a large number of categories, you should avoid

creating information bottlenecks in your network due to intermediate layers

that are too small.

25. **Normalize data**

***mean = train\_data.mean(axis=0)***

***train\_data -= mean***

***std = train\_data.std(axis=0)***

***train\_data /= std***

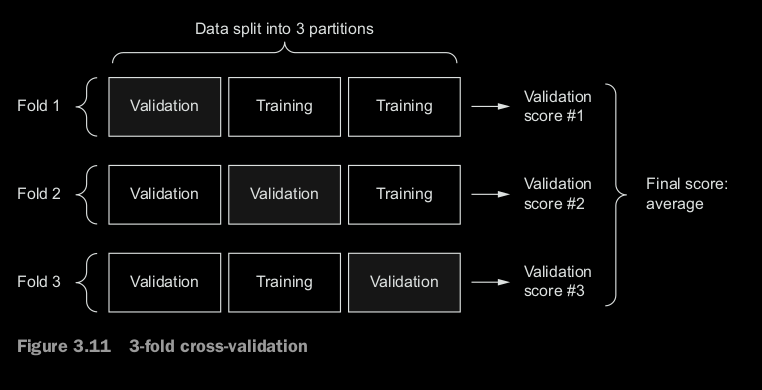
***test\_data -= mean***

***test\_data /= std***

**26. Mean squared error (mse):** loss function for regression problem

27. **mean absolute error (mae)**

**28. K-fold validation: It consists of splitting the available data into K partitions (typically K = 4 or 5), instanti-**

**ating K identical models, and training each one on K – 1 partitions while evaluating on the remaining partition. The validation score for the model used is then the average of the K validation scores obtained. In terms of code, this is straightforward.**

***import numpy as np***

***k = 4***

***num\_val\_samples = len(train\_data) // k***

***num\_epochs = 100***

***all\_scores = []***

***for i in range(k):***

***print('processing fold #', i)***

***val\_data = train\_data[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]***

***val\_targets = train\_targets[i \* num\_val\_samples: (i + 1) \* num\_val\_samples]***

***partial\_train\_data = np.concatenate(***

***[train\_data[:i \* num\_val\_samples],***

***train\_data[(i + 1) \* num\_val\_samples:]],***

***axis=0)***

***partial\_train\_targets = np.concatenate(***

***[train\_targets[:i \* num\_val\_samples],***

***train\_targets[(i + 1) \* num\_val\_samples:]],***

***axis=0)***

***model = build\_model()***

***model.fit(partial\_train\_data, partial\_train\_targets,***

***epochs=num\_epochs, batch\_size=1, verbose=0)***

***val\_mse, val\_mae = model.evaluate(val\_data, val\_targets, verbose=0)***

***all\_scores.append(val\_mae)***

**29. Smooth curve: to obtain the smooth curve for data observation**

**Replace each point with an exponential moving average of the previous points,**

**to obtain a smooth curve**

***def smooth\_curve(points, factor=0.9):***

***smoothed\_points = []***

***for point in points:***

***if smoothed\_points:***

***previous = smoothed\_points[-1]***

***smoothed\_points.append(previous \* factor + point \* (1 - factor))***

***else:***

***smoothed\_points.append(point)***

***return smoothed\_points***

***smooth\_mae\_history = smooth\_curve(average\_mae\_history[10:])***

***plt.plot(range(1, len(smooth\_mae\_history) + 1), smooth\_mae\_history)***

***plt.xlabel('Epochs')***

***plt.ylabel('Validation MAE')***

***plt.show()***

**SUMMARY ON REGRESSION PROBLEM:**

**-Regression is done using different loss functions than what we used for classification. Mean squared error ( MSE ) is a loss function commonly used for regression.**

**-Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally, the concept of accuracy doesn’t apply for regression. A common regression metric is mean absolute error ( MAE ).**

**-When features in the input data have values in different ranges, each feature**

**should be scaled independently as a preprocessing step.**

**-When there is little data available, using K-fold validation is a great way to reliably evaluate a model.**

**-When little training data is available, it’s preferable to use a small network with few hidden layers (typically only one or two), in order to avoid severe overfitting.**

**CHAPTER SUMMARY**

** You’re now able to handle the most common kinds of machine-learning**

**tasks on vector data: binary classification, multiclass classification, and sca-**

**lar regression. The “Wrapping up” sections earlier in the chapter summa-**

**rize the important points you’ve learned regarding these types of tasks.**

** You’ll usually need to preprocess raw data before feeding it into a neural**

**network.**

** When your data has features with different ranges, scale each feature**

**independently as part of preprocessing.**

** As training progresses, neural networks eventually begin to overfit and**

**obtain worse results on never-before-seen data.**

** If you don’t have much training data, use a small network with only one or**

**two hidden layers, to avoid severe overfitting.**

** If your data is divided into many categories, you may cause information**

**bottlenecks if you make the intermediate layers too small.**

** Regression uses different loss functions and different evaluation metrics**

**than classification.**

** When you’re working with little data, K-fold validation can help reliably**

**evaluate your model.**