Chapter 11 – Training Deep Neural Networks

This notebook contains all the sample code and solutions to the exercises in chapter 11.

[Run in Google Colab (https://colab.research.google.com/github/ageron/handson-ml2/blob/master/11_training_deep_neural_networks.ipynb)](https://colab.research.google.com/github/ageron/handson-ml2/blob/master/11_training_deep_neural_networks.ipynb)

Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20 and TensorFlow ≥2.0.

In [1]:
```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
except Exception:
    pass

# TensorFlow ≥2.0 is required
import tensorflow as tf
from tensorflow import keras
assert tf.__version__ >= "2.0"

%load_ext tensorboard

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "deep"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=30
0):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

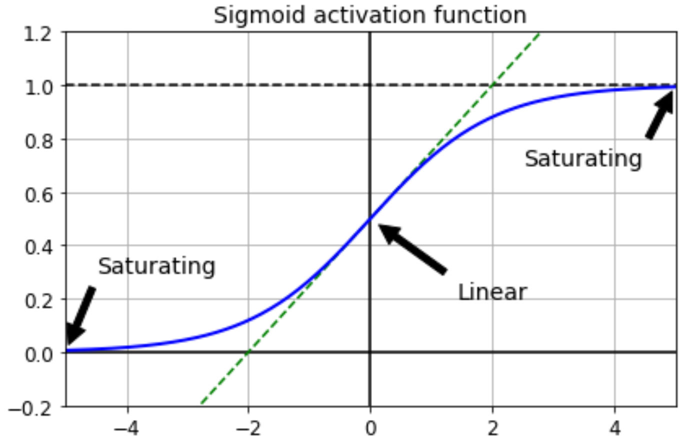
1. Vanishing/Exploding Gradients Problem

The saturating activation functions can be problematic and lead to vanishing/exploding gradients problem.

In [2]:
```
def logit(z):
    return 1 / (1 + np.exp(-z))
```

In [3]:
```
z = np.linspace(-5, 5, 200)

plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([-5, 5], [1, 1], 'k--')
plt.plot([0, 0], [-0.2, 1.2], 'k-')
plt.plot([-5, 5], [-3/4, 7/4], 'g--')
plt.plot(z, logit(z), "b-", linewidth=2)
props = dict(facecolor='black', shrink=0.1)
plt.annotate('Saturating', xytext=(3.5, 0.7), xy=(5, 1), arrowprops=props, f
ontsize=14, ha="center")
plt.annotate('Saturating', xytext=(-3.5, 0.3), xy=(-5, 0), arrowprops=props,
fontsize=14, ha="center")
plt.annotate('Linear', xytext=(2, 0.2), xy=(0, 0.5), arrowprops=props, fonts
ize=14, ha="center")
plt.grid(True)
plt.title("Sigmoid activation function", fontsize=14)
plt.axis([-5, 5, -0.2, 1.2])

save_fig("sigmoid_saturation_plot")
plt.show()
```

```
Saving figure sigmoid_saturation_plot
```



1.1 Xavier and He Initialization

Using the Xavier or He initialization for the weights helps prevent the vanishing/exploding gradient problem.

In [4]:
```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Out[4]:
```
<tensorflow.python.keras.layers.core.Dense at 0x1a343418948>
```

1.2 Nonsaturating Activation Functions

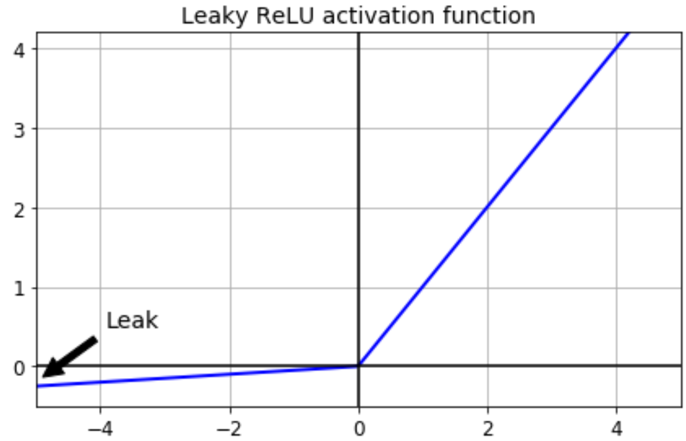
Using non-saturating activation functions helps with the vanishing/exploding gradient problem.

Leaky ReLU

In [5]:
```
def leaky_relu(z, alpha=0.01):
    return np.maximum(alpha*z, z)
```

In [6]:
```
plt.plot(z, leaky_relu(z, 0.05), "b-", linewidth=2)
plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([0, 0], [-0.5, 4.2], 'k-')
plt.grid(True)
props = dict(facecolor='black', shrink=0.1)
plt.annotate('Leak', xytext=(-3.5, 0.5), xy=(-5, -0.2), arrowprops=props, fo
ntsize=14, ha="center")
plt.title("Leaky ReLU activation function", fontsize=14)
plt.axis([-5, 5, -0.5, 4.2])

save_fig("leaky_relu_plot")
plt.show()
```

```
Saving figure leaky_relu_plot
```



Let's train a neural network on Fashion MNIST using the Leaky ReLU:

In [7]:
```
(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnis
t.load_data()
X_train_full = X_train_full / 255.0
X_test = X_test / 255.0
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

In [8]:
```
tf.random.set_seed(42)
np.random.seed(42)

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(),
    keras.layers.Dense(100, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(),
    keras.layers.Dense(10, activation="softmax")
])
```

In [9]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

In [10]:
```
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid))
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 4s 78us/sample - loss: 1.2810
- accuracy: 0.6205 - val_loss: 0.8869 - val_accuracy: 0.7160
Epoch 2/10
55000/55000 [==============================] - 3s 62us/sample - loss: 0.7952
- accuracy: 0.7368 - val_loss: 0.7132 - val_accuracy: 0.7626
Epoch 3/10
55000/55000 [==============================] - 4s 64us/sample - loss: 0.6817
- accuracy: 0.7726 - val_loss: 0.6385 - val_accuracy: 0.7896
Epoch 4/10
55000/55000 [==============================] - 3s 60us/sample - loss: 0.6219
- accuracy: 0.7942 - val_loss: 0.5931 - val_accuracy: 0.8016
Epoch 5/10
55000/55000 [==============================] - 4s 70us/sample - loss: 0.5829
- accuracy: 0.8074 - val_loss: 0.5607 - val_accuracy: 0.8164
Epoch 6/10
55000/55000 [==============================] - 4s 65us/sample - loss: 0.5552
- accuracy: 0.8173 - val_loss: 0.5355 - val_accuracy: 0.8240
Epoch 7/10
55000/55000 [==============================] - 3s 59us/sample - loss: 0.5338
- accuracy: 0.8225 - val_loss: 0.5166 - val_accuracy: 0.8300
Epoch 8/10
55000/55000 [==============================] - 3s 62us/sample - loss: 0.5172
- accuracy: 0.8261 - val_loss: 0.5043 - val_accuracy: 0.8356
Epoch 9/10
55000/55000 [==============================] - 3s 53us/sample - loss: 0.5039
- accuracy: 0.8305 - val_loss: 0.4889 - val_accuracy: 0.8386
Epoch 10/10
55000/55000 [==============================] - 3s 55us/sample - loss: 0.4923
- accuracy: 0.8333 - val_loss: 0.4816 - val_accuracy: 0.8396
```

Now let's try PReLU:

In [11]:
```
tf.random.set_seed(42)
np.random.seed(42)

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, kernel_initializer="he_normal"),
    keras.layers.PReLU(),
    keras.layers.Dense(100, kernel_initializer="he_normal"),
    keras.layers.PReLU(),
    keras.layers.Dense(10, activation="softmax")
])
```

In [12]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

In [13]:
```
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid))
```

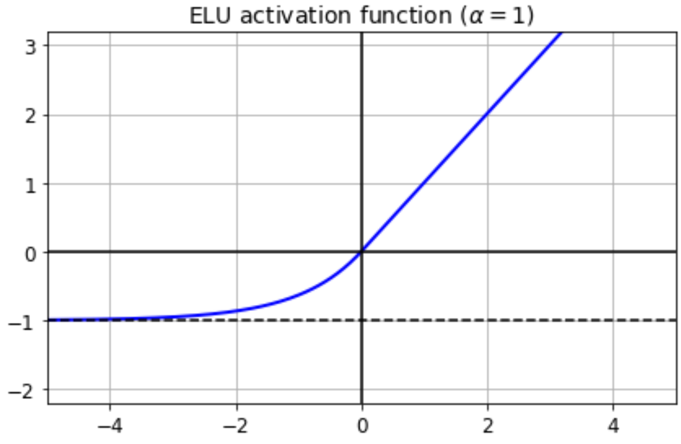
```
Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 3s 63us/sample - loss: 1.3452
- accuracy: 0.6203 - val_loss: 0.9241 - val_accuracy: 0.7170
Epoch 2/10
55000/55000 [==============================] - 3s 60us/sample - loss: 0.8196
- accuracy: 0.7364 - val_loss: 0.7314 - val_accuracy: 0.7602
Epoch 3/10
55000/55000 [==============================] - 3s 63us/sample - loss: 0.6970
- accuracy: 0.7701 - val_loss: 0.6517 - val_accuracy: 0.7878
Epoch 4/10
55000/55000 [==============================] - 4s 66us/sample - loss: 0.6333
- accuracy: 0.7915 - val_loss: 0.6032 - val_accuracy: 0.8056
Epoch 5/10
55000/55000 [==============================] - 4s 64us/sample - loss: 0.5917
- accuracy: 0.8049 - val_loss: 0.5689 - val_accuracy: 0.8162
Epoch 6/10
55000/55000 [==============================] - 4s 66us/sample - loss: 0.5619
- accuracy: 0.8143 - val_loss: 0.5417 - val_accuracy: 0.8222
Epoch 7/10
55000/55000 [==============================] - 4s 66us/sample - loss: 0.5392
- accuracy: 0.8205 - val_loss: 0.5213 - val_accuracy: 0.8298
Epoch 8/10
55000/55000 [==============================] - 5s 94us/sample - loss: 0.5215
- accuracy: 0.8257 - val_loss: 0.5075 - val_accuracy: 0.8352
Epoch 9/10
55000/55000 [==============================] - 5s 96us/sample - loss: 0.5071
- accuracy: 0.8287 - val_loss: 0.4917 - val_accuracy: 0.8384
Epoch 10/10
55000/55000 [==============================] - 4s 78us/sample - loss: 0.4946
- accuracy: 0.8322 - val_loss: 0.4839 - val_accuracy: 0.8378
```

ELU

In [14]:
```
def elu(z, alpha=1):
    return np.where(z < 0, alpha * (np.exp(z) - 1), z)
```

In [15]:
```
plt.plot(z, elu(z), "b-", linewidth=2)
plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([-5, 5], [-1, -1], 'k--')
plt.plot([0, 0], [-2.2, 3.2], 'k-')
plt.grid(True)
plt.title(r"ELU activation function ($\alpha=1$)", fontsize=14)
plt.axis([-5, 5, -2.2, 3.2])

save_fig("elu_plot")
plt.show()
```

```
Saving figure elu_plot
```



Implementing ELU in TensorFlow is trivial, just specify the activation function when building each layer:

In [16]:
```
keras.layers.Dense(10, activation="elu")
```

Out[16]:
```
<tensorflow.python.keras.layers.core.Dense at 0x1a347039088>
```

SELU

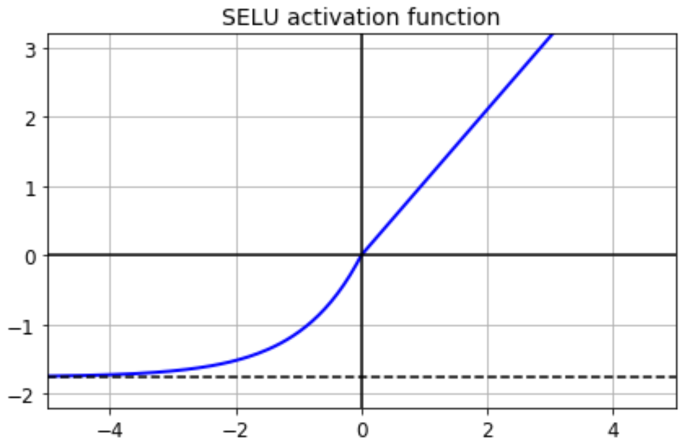
This activation function was proposed in this great paper (https://arxiv.org/pdf/1706.02515.pdf) by Günter Klambauer, Thomas Unterthiner and Andreas Mayr, published in June 2017. During training, a neural network composed exclusively of a stack of dense layers using the SELU activation function and LeCun initialization will self-normalize: the output of each layer will tend to preserve the same mean and variance during training, which solves the vanishing/exploding gradients problem. As a result, this activation function outperforms the other activation functions very significantly for such neural nets, so you should really try it out. Unfortunately, the self-normalizing property of the SELU activation function is easily broken: you cannot use ℓ_1 or ℓ_2 regularization, regular dropout, max-norm, skip connections or other non-sequential topologies (so recurrent neural networks won't self-normalize). However, in practice it works quite well with sequential CNNs. If you break self-normalization, SELU will not necessarily outperform other activation functions.

In [17]:
```
from scipy.special import erfc

# alpha and scale to self normalize with mean 0 and standard deviation 1
# (see equation 14 in the paper):
alpha_0_1 = -np.sqrt(2 / np.pi) / (erfc(1/np.sqrt(2)) * np.exp(1/2) - 1)
scale_0_1 = (1 - erfc(1 / np.sqrt(2)) * np.sqrt(np.e)) * np.sqrt(2 * np.pi)
* (2 * erfc(np.sqrt(2))*np.e**2 + np.pi*erfc(1/np.sqrt(2))**2*np.e - 2*(2+n
p.pi)*erfc(1/np.sqrt(2))*np.sqrt(np.e)+np.pi+2)**(-1/2)
```

In [18]:
```
def selu(z, scale=scale_0_1, alpha=alpha_0_1):
    return scale * elu(z, alpha)
```

In [19]:
```
plt.plot(z, selu(z), "b-", linewidth=2)
plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([-5, 5], [-1.758, -1.758], 'k--')
plt.plot([0, 0], [-2.2, 3.2], 'k-')
plt.grid(True)
plt.title("SELU activation function", fontsize=14)
plt.axis([-5, 5, -2.2, 3.2])

save_fig("selu_plot")
plt.show()
```

```
Saving figure selu_plot
```



By default, the SELU hyperparameters (`scale` and `alpha`) are tuned in such a way that the mean output of each neuron remains close to 0, and the standard deviation remains close to 1 (assuming the inputs are standardized with mean 0 and standard deviation 1 too). Using this activation function, even a 1,000 layer deep neural network preserves roughly mean 0 and standard deviation 1 across all layers, avoiding the exploding/vanishing gradients problem:

In [20]:
```
np.random.seed(42)
Z = np.random.normal(size=(500, 100)) # standardized inputs
for layer in range(1000):
    W = np.random.normal(size=(100, 100), scale=np.sqrt(1 / 100)) # LeCun in
itialization
    Z = selu(np.dot(Z, W))
    means = np.mean(Z, axis=0).mean()
    stds = np.std(Z, axis=0).mean()
    if layer % 100 == 0:
        print("Layer {}: mean {:.2f}, std deviation {:.2f}".format(layer, me
ans, stds))
```

```
Layer 0: mean -0.00, std deviation 1.00
Layer 100: mean 0.02, std deviation 0.96
Layer 200: mean 0.01, std deviation 0.90
Layer 300: mean -0.02, std deviation 0.92
Layer 400: mean 0.05, std deviation 0.89
Layer 500: mean 0.01, std deviation 0.93
Layer 600: mean 0.02, std deviation 0.92
Layer 700: mean -0.02, std deviation 0.90
Layer 800: mean 0.05, std deviation 0.83
Layer 900: mean 0.02, std deviation 1.00
```

Using SELU is easy:

In [21]:
```
keras.layers.Dense(10, activation="selu",
                   kernel_initializer="lecun_normal")
```

Out[21]:
```
<tensorflow.python.keras.layers.core.Dense at 0x1a343414f88>
```

Let's create a neural net for Fashion MNIST with 100 hidden layers, using the SELU activation function:

In [22]:
```
np.random.seed(42)
tf.random.set_seed(42)
```

In [23]:
```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="selu",
                             kernel_initializer="lecun_normal"))
for layer in range(99):
    model.add(keras.layers.Dense(100, activation="selu",
                                 kernel_initializer="lecun_normal"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

In [24]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

Now let's train it. Do not forget to scale the inputs to mean 0 and standard deviation 1:

In [25]:
```
pixel_means = X_train.mean(axis=0, keepdims=True)
pixel_stds = X_train.std(axis=0, keepdims=True)
X_train_scaled = (X_train - pixel_means) / pixel_stds
X_valid_scaled = (X_valid - pixel_means) / pixel_stds
X_test_scaled = (X_test - pixel_means) / pixel_stds
```

In [26]:
```
history = model.fit(X_train_scaled, y_train, epochs=5,
                    validation_data=(X_valid_scaled, y_valid))
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/5
55000/55000 [==============================] - 36s 657us/sample - loss: 0.993
3 - accuracy: 0.6273 - val_loss: 0.7413 - val_accuracy: 0.7384
Epoch 2/5
55000/55000 [==============================] - 29s 535us/sample - loss: 0.647
0 - accuracy: 0.7691 - val_loss: 0.6637 - val_accuracy: 0.7672
Epoch 3/5
55000/55000 [==============================] - 29s 526us/sample - loss: 0.566
8 - accuracy: 0.8016 - val_loss: 0.4969 - val_accuracy: 0.8300
Epoch 4/5
55000/55000 [==============================] - 29s 530us/sample - loss: 0.511
3 - accuracy: 0.8227 - val_loss: 0.4977 - val_accuracy: 0.8314
Epoch 5/5
55000/55000 [==============================] - 35s 630us/sample - loss: 0.470
8 - accuracy: 0.8372 - val_loss: 0.4595 - val_accuracy: 0.8364
```

1.3 Batch Normalization

Sometimes applying batch normalization before the activation function works better (there's a debate on this topic). Moreover, the layer before a `BatchNormalization` layer does not need to have bias terms, since the `BatchNormalization` layer some as well, it would be a waste of parameters, so you can set `use_bias=False` when creating those layers:

In [27]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("relu"),
    keras.layers.Dense(100, use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

In [28]:
```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

In [29]:
```
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid))
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 9s 162us/sample - loss: 1.0829
- accuracy: 0.6616 - val_loss: 0.6702 - val_accuracy: 0.7964
Epoch 2/10
55000/55000 [==============================] - 8s 138us/sample - loss: 0.6749
- accuracy: 0.7862 - val_loss: 0.5470 - val_accuracy: 0.8230
Epoch 3/10
55000/55000 [==============================] - 8s 143us/sample - loss: 0.5893
- accuracy: 0.8075 - val_loss: 0.4938 - val_accuracy: 0.8412
Epoch 4/10
55000/55000 [==============================] - 8s 138us/sample - loss: 0.5459
- accuracy: 0.8195 - val_loss: 0.4601 - val_accuracy: 0.8492
Epoch 5/10
55000/55000 [==============================] - 8s 147us/sample - loss: 0.5127
- accuracy: 0.8272 - val_loss: 0.4379 - val_accuracy: 0.8556
Epoch 6/10
55000/55000 [==============================] - 8s 150us/sample - loss: 0.4941
- accuracy: 0.8311 - val_loss: 0.4213 - val_accuracy: 0.8586
Epoch 7/10
55000/55000 [==============================] - 8s 152us/sample - loss: 0.4739
- accuracy: 0.8390 - val_loss: 0.4083 - val_accuracy: 0.8616
Epoch 8/10
55000/55000 [==============================] - 8s 148us/sample - loss: 0.4595
- accuracy: 0.8411 - val_loss: 0.3974 - val_accuracy: 0.8662
Epoch 9/10
55000/55000 [==============================] - 10s 180us/sample - loss: 0.444
2 - accuracy: 0.8472 - val_loss: 0.3884 - val_accuracy: 0.8656
Epoch 10/10
55000/55000 [==============================] - 10s 179us/sample - loss: 0.433
7 - accuracy: 0.8496 - val_loss: 0.3817 - val_accuracy: 0.8676
```

1.4 Gradient Clipping

All Keras optimizers accept `clipnorm` or `clipvalue` arguments:

In [30]:
```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
```

In [31]:
```
optimizer = keras.optimizers.SGD(clipnorm=1.0)
```

2. Reusing Pretrained Layers

Reusing a Keras model

Let's split the fashion MNIST training set in two:

- `X_train_A` : all images of all items except for sandals and shirts (classes 5 and 6).
- `X_train_B` : a much smaller training set of just the first 200 images of sandals or shirts.

The validation set and the test set are also split this way, but without restricting the number of images.

We will train a model on set A (classification task with 8 classes), and try to reuse it to tackle set B (binary classification). We hope to transfer a little bit of knowledge from task A to task B, since classes in set A (sneakers, ankle boots, coats, t-shirts, etc.) are somewhat similar to classes in set B (sandals and shirts). However, since we are using `Dense` layers, only patterns that occur at the same location can be reused (in contrast, convolutional layers will transfer much better, since learned patterns can be detected anywhere on the image, as we will see in the CNN chapter).

In [32]:
```
def split_dataset(X, y):
    y_5_or_6 = (y == 5) | (y == 6) # sandals or shirts
    y_A = y[~y_5_or_6]
    y_A[y_A > 6] -= 2 # class indices 7, 8, 9 should be moved to 5, 6, 7
    y_B = (y[y_5_or_6] == 6).astype(np.float32) # binary classification tas
k: is it a shirt (class 6)?
    return ((X[~y_5_or_6], y_A),
            (X[y_5_or_6], y_B))

(X_train_A, y_train_A), (X_train_B, y_train_B) = split_dataset(X_train, y_tr
ain)
(X_valid_A, y_valid_A), (X_valid_B, y_valid_B) = split_dataset(X_valid, y_va
lid)
(X_test_A, y_test_A), (X_test_B, y_test_B) = split_dataset(X_test, y_test)
X_train_B = X_train_B[:200]
y_train_B = y_train_B[:200]
```

In [33]:
```
X_train_A.shape
```

Out[33]:
```
(43986, 28, 28)
```

In [34]:
```
X_train_B.shape
```

Out[34]:
```
(200, 28, 28)
```

In [35]:
```
y_train_A[:30]
```

Out[35]:
```
array([4, 0, 5, 7, 7, 7, 4, 4, 3, 4, 0, 1, 6, 3, 4, 3, 2, 6, 5, 3, 4, 5,
       1, 3, 4, 2, 0, 6, 7, 1], dtype=uint8)
```

In [36]:
```
y_train_B[:30]
```

Out[36]:
```
array([1., 1., 0., 0., 0., 0., 1., 1., 1., 0., 0., 1., 1., 0., 0., 0., 0.,
       0., 0., 1., 1., 0., 0., 1., 1., 0., 1., 1., 1., 1.], dtype=float32)
```

In [37]:
```
tf.random.set_seed(42)
np.random.seed(42)
```

In [38]:
```
model_A = keras.models.Sequential()
model_A.add(keras.layers.Flatten(input_shape=[28, 28]))
for n_hidden in (300, 100, 50, 50, 50):
    model_A.add(keras.layers.Dense(n_hidden, activation="selu"))
model_A.add(keras.layers.Dense(8, activation="softmax"))
```

In [39]:
```
model_A.compile(loss="sparse_categorical_crossentropy",
                optimizer=keras.optimizers.SGD(lr=1e-3),
                metrics=["accuracy"])
```

In [40]:
```
history = model_A.fit(X_train_A, y_train_A, epochs=20,
                      validation_data=(X_valid_A, y_valid_A))
```

```
Train on 43986 samples, validate on 4014 samples
Epoch 1/20
43986/43986 [==============================] - 6s 126us/sample - loss: 0.5902
- accuracy: 0.8133 - val_loss: 0.3782 - val_accuracy: 0.8692
Epoch 2/20
43986/43986 [==============================] - 5s 104us/sample - loss: 0.3518
- accuracy: 0.8783 - val_loss: 0.3370 - val_accuracy: 0.8839
Epoch 3/20
43986/43986 [==============================] - 5s 110us/sample - loss: 0.3163
- accuracy: 0.8896 - val_loss: 0.3019 - val_accuracy: 0.8956
Epoch 4/20
43986/43986 [==============================] - 5s 116us/sample - loss: 0.2969
- accuracy: 0.8973 - val_loss: 0.2912 - val_accuracy: 0.9013
Epoch 5/20
43986/43986 [==============================] - 6s 139us/sample - loss: 0.2831
- accuracy: 0.9027 - val_loss: 0.2816 - val_accuracy: 0.9016
Epoch 6/20
43986/43986 [==============================] - 5s 117us/sample - loss: 0.2725
- accuracy: 0.9065 - val_loss: 0.2736 - val_accuracy: 0.9073
Epoch 7/20
43986/43986 [==============================] - 4s 85us/sample - loss: 0.2644
- accuracy: 0.9094 - val_loss: 0.2649 - val_accuracy: 0.9093
Epoch 8/20
43986/43986 [==============================] - 4s 90us/sample - loss: 0.2577
- accuracy: 0.9117 - val_loss: 0.2579 - val_accuracy: 0.9131
Epoch 9/20
43986/43986 [==============================] - 4s 94us/sample - loss: 0.2517
- accuracy: 0.9137 - val_loss: 0.2581 - val_accuracy: 0.9133
Epoch 10/20
43986/43986 [==============================] - 5s 106us/sample - loss: 0.2466
- accuracy: 0.9152 - val_loss: 0.2521 - val_accuracy: 0.9150
Epoch 11/20
43986/43986 [==============================] - 4s 92us/sample - loss: 0.2420
- accuracy: 0.9178 - val_loss: 0.2489 - val_accuracy: 0.9160
Epoch 12/20
43986/43986 [==============================] - 4s 95us/sample - loss: 0.2381
- accuracy: 0.9191 - val_loss: 0.2454 - val_accuracy: 0.9173
Epoch 13/20
43986/43986 [==============================] - 5s 104us/sample - loss: 0.2348
- accuracy: 0.9197 - val_loss: 0.2448 - val_accuracy: 0.9193
Epoch 14/20
43986/43986 [==============================] - 4s 97us/sample - loss: 0.2312
- accuracy: 0.9202 - val_loss: 0.2431 - val_accuracy: 0.9175
Epoch 15/20
43986/43986 [==============================] - 4s 88us/sample - loss: 0.2282
- accuracy: 0.9220 - val_loss: 0.2430 - val_accuracy: 0.9178
Epoch 16/20
43986/43986 [==============================] - 3s 78us/sample - loss: 0.2256
- accuracy: 0.9228 - val_loss: 0.2413 - val_accuracy: 0.9155
Epoch 17/20
43986/43986 [==============================] - 6s 127us/sample - loss: 0.2229
- accuracy: 0.9229 - val_loss: 0.2368 - val_accuracy: 0.9180
Epoch 18/20
43986/43986 [==============================] - 5s 115us/sample - loss: 0.2202
- accuracy: 0.9243 - val_loss: 0.2433 - val_accuracy: 0.9175
Epoch 19/20
43986/43986 [==============================] - 4s 88us/sample - loss: 0.2177
- accuracy: 0.9250 - val_loss: 0.2609 - val_accuracy: 0.9053
Epoch 20/20
43986/43986 [==============================] - 4s 88us/sample - loss: 0.2159
- accuracy: 0.9265 - val_loss: 0.2328 - val_accuracy: 0.9205
```

In [41]:
```
model_A.save("my_model_A.h5")
```

In [42]:
```
model_B = keras.models.Sequential()
model_B.add(keras.layers.Flatten(input_shape=[28, 28]))
for n_hidden in (300, 100, 50, 50, 50):
    model_B.add(keras.layers.Dense(n_hidden, activation="selu"))
model_B.add(keras.layers.Dense(1, activation="sigmoid"))
```

In [43]:
```
model_B.compile(loss="binary_crossentropy",
                optimizer=keras.optimizers.SGD(lr=1e-3),
                metrics=["accuracy"])
```

In [44]:
```
history = model_B.fit(X_train_B, y_train_B, epochs=20,
                      validation_data=(X_valid_B, y_valid_B))
```

```
Train on 200 samples, validate on 986 samples
Epoch 1/20
200/200 [==============================] - 1s 3ms/sample - loss: 0.9509 - acc
uracy: 0.4800 - val_loss: 0.6533 - val_accuracy: 0.5568
Epoch 2/20
200/200 [==============================] - 0s 469us/sample - loss: 0.5837 - a
ccuracy: 0.7100 - val_loss: 0.4825 - val_accuracy: 0.8479
Epoch 3/20
200/200 [==============================] - 0s 509us/sample - loss: 0.4527 - a
ccuracy: 0.8750 - val_loss: 0.4097 - val_accuracy: 0.8945
Epoch 4/20
200/200 [==============================] - 0s 534us/sample - loss: 0.3869 - a
ccuracy: 0.9050 - val_loss: 0.3630 - val_accuracy: 0.9209
Epoch 5/20
200/200 [==============================] - 0s 538us/sample - loss: 0.3404 - a
ccuracy: 0.9300 - val_loss: 0.3302 - val_accuracy: 0.9280
Epoch 6/20
200/200 [==============================] - 0s 559us/sample - loss: 0.3073 - a
ccuracy: 0.9350 - val_loss: 0.3026 - val_accuracy: 0.9381
Epoch 7/20
200/200 [==============================] - 0s 514us/sample - loss: 0.2797 - a
ccuracy: 0.9400 - val_loss: 0.2790 - val_accuracy: 0.9452
Epoch 8/20
200/200 [==============================] - 0s 519us/sample - loss: 0.2554 - a
ccuracy: 0.9450 - val_loss: 0.2595 - val_accuracy: 0.9473
Epoch 9/20
200/200 [==============================] - 0s 532us/sample - loss: 0.2355 - a
ccuracy: 0.9600 - val_loss: 0.2439 - val_accuracy: 0.9493
Epoch 10/20
200/200 [==============================] - 0s 547us/sample - loss: 0.2187 - a
ccuracy: 0.9650 - val_loss: 0.2293 - val_accuracy: 0.9523
Epoch 11/20
200/200 [==============================] - 0s 538us/sample - loss: 0.2041 - a
ccuracy: 0.9650 - val_loss: 0.2162 - val_accuracy: 0.9544
Epoch 12/20
200/200 [==============================] - 0s 499us/sample - loss: 0.1906 - a
ccuracy: 0.9650 - val_loss: 0.2049 - val_accuracy: 0.9574
Epoch 13/20
200/200 [==============================] - 0s 494us/sample - loss: 0.1791 - a
ccuracy: 0.9700 - val_loss: 0.1946 - val_accuracy: 0.9594
Epoch 14/20
200/200 [==============================] - 0s 509us/sample - loss: 0.1686 - a
ccuracy: 0.9750 - val_loss: 0.1856 - val_accuracy: 0.9615
Epoch 15/20
200/200 [==============================] - 0s 499us/sample - loss: 0.1591 - a
ccuracy: 0.9750 - val_loss: 0.1765 - val_accuracy: 0.9655
Epoch 16/20
200/200 [==============================] - 0s 494us/sample - loss: 0.1502 - a
ccuracy: 0.9900 - val_loss: 0.1695 - val_accuracy: 0.9655
Epoch 17/20
200/200 [==============================] - 0s 475us/sample - loss: 0.1424 - a
ccuracy: 0.9900 - val_loss: 0.1624 - val_accuracy: 0.9686
Epoch 18/20
200/200 [==============================] - 0s 464us/sample - loss: 0.1351 - a
ccuracy: 0.9900 - val_loss: 0.1567 - val_accuracy: 0.9686
Epoch 19/20
200/200 [==============================] - 0s 534us/sample - loss: 0.1290 - a
ccuracy: 0.9900 - val_loss: 0.1513 - val_accuracy: 0.9696
Epoch 20/20
200/200 [==============================] - 0s 469us/sample - loss: 0.1229 - a
ccuracy: 0.9900 - val_loss: 0.1450 - val_accuracy: 0.9696
```

In [45]:
```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten_3 (Flatten)          (None, 784)               0         
_________________________________________________________________
batch_normalization (BatchNo (None, 784)               3136      
_________________________________________________________________
dense_110 (Dense)            (None, 300)               235200    
_________________________________________________________________
batch_normalization_1 (Batch (None, 300)               1200      
_________________________________________________________________
activation (Activation)      (None, 300)               0         
_________________________________________________________________
dense_111 (Dense)            (None, 100)               30000     
_________________________________________________________________
batch_normalization_2 (Batch (None, 100)               400       
_________________________________________________________________
activation_1 (Activation)    (None, 100)               0         
_________________________________________________________________
dense_112 (Dense)            (None, 10)                1010      
=================================================================
Total params: 270,946
Trainable params: 268,578
Non-trainable params: 2,368
_________________________________________________________________
```

In [46]:
```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

In [47]:
```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

In [48]:
```
model_B_on_A.compile(loss="binary_crossentropy",
                     optimizer=keras.optimizers.SGD(lr=1e-3),
                     metrics=["accuracy"])
```

In [49]:
```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

```
Train on 200 samples, validate on 986 samples
Epoch 1/16
200/200 [==============================] - 1s 3ms/sample - loss: 0.4905 - acc
uracy: 0.7550 - val_loss: 0.4044 - val_accuracy: 0.8063
Epoch 2/16
200/200 [==============================] - 0s 404us/sample - loss: 0.3263 - a
ccuracy: 0.8700 - val_loss: 0.3033 - val_accuracy: 0.8854
Epoch 3/16
200/200 [==============================] - 0s 414us/sample - loss: 0.2430 - a
ccuracy: 0.9400 - val_loss: 0.2436 - val_accuracy: 0.9371
Epoch 4/16
200/200 [==============================] - 0s 434us/sample - loss: 0.1929 - a
ccuracy: 0.9650 - val_loss: 0.2047 - val_accuracy: 0.9533
Epoch 5/16
200/200 [==============================] - 0s 459us/sample - loss: 0.1596 - a
ccuracy: 0.9800 - val_loss: 0.1756 - val_accuracy: 0.9655
Epoch 6/16
200/200 [==============================] - 0s 451us/sample - loss: 0.1345 - a
ccuracy: 0.9800 - val_loss: 0.1545 - val_accuracy: 0.9716
Epoch 7/16
200/200 [==============================] - 0s 414us/sample - loss: 0.1164 - a
ccuracy: 0.9900 - val_loss: 0.1392 - val_accuracy: 0.9777
Epoch 8/16
200/200 [==============================] - 0s 437us/sample - loss: 0.1031 - a
ccuracy: 0.9900 - val_loss: 0.1269 - val_accuracy: 0.9807
Epoch 9/16
200/200 [==============================] - 0s 420us/sample - loss: 0.0924 - a
ccuracy: 0.9950 - val_loss: 0.1169 - val_accuracy: 0.9828
Epoch 10/16
200/200 [==============================] - 0s 429us/sample - loss: 0.0838 - a
ccuracy: 0.9950 - val_loss: 0.1086 - val_accuracy: 0.9838
Epoch 11/16
200/200 [==============================] - 0s 441us/sample - loss: 0.0768 - a
ccuracy: 1.0000 - val_loss: 0.1017 - val_accuracy: 0.9868
Epoch 12/16
200/200 [==============================] - 0s 419us/sample - loss: 0.0707 - a
ccuracy: 1.0000 - val_loss: 0.0952 - val_accuracy: 0.9888
Epoch 13/16
200/200 [==============================] - 0s 395us/sample - loss: 0.0651 - a
ccuracy: 1.0000 - val_loss: 0.0902 - val_accuracy: 0.9888
Epoch 14/16
200/200 [==============================] - 0s 424us/sample - loss: 0.0606 - a
ccuracy: 1.0000 - val_loss: 0.0853 - val_accuracy: 0.9899
Epoch 15/16
200/200 [==============================] - 0s 419us/sample - loss: 0.0565 - a
ccuracy: 1.0000 - val_loss: 0.0814 - val_accuracy: 0.9899
Epoch 16/16
200/200 [==============================] - 0s 404us/sample - loss: 0.0529 - a
ccuracy: 1.0000 - val_loss: 0.0780 - val_accuracy: 0.9899
```

So, what's the final verdict?

In [50]:
```
model_B.evaluate(X_test_B, y_test_B)
```

```
2000/2000 [==============================] - 0s 55us/sample - loss: 0.1426 -
accuracy: 0.9695
```

Out[50]:
```
[0.1426312597990036, 0.9695]
```

In [51]:
```
model_B_on_A.evaluate(X_test_B, y_test_B)
```

```
2000/2000 [==============================] - 0s 61us/sample - loss: 0.0725 - accuracy: 0.9925
```

Out[51]:
```
[0.0724904710650444, 0.9925]
```

Great! We got quite a bit of transfer: the error rate dropped by a factor of 4!

In [52]:
```
(100 - 96.95) / (100 - 99.25)
```

Out[52]:
```
4.066666666666663
```

Faster Optimizers

Momentum optimization

In [53]:
```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

Nesterov Accelerated Gradient

In [54]:
```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

AdaGrad

In [55]:
```
optimizer = keras.optimizers.Adagrad(lr=0.001)
```

RMSProp

In [56]:
```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Adam Optimization

In [57]:
```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Adamax Optimization

In [58]:
```
optimizer = keras.optimizers.Adamax(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Nadam Optimization

In [59]:
```
optimizer = keras.optimizers.Nadam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

4. Learning Rate Scheduling

tf.keras schedulers

In [60]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="selu", kernel_initializer="lecun_nor
mal"),
    keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_nor
mal"),
    keras.layers.Dense(10, activation="softmax")
])
s = 20 * len(X_train) // 32 # number of steps in 20 epochs (batch size = 32)
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer, m
etrics=["accuracy"])
n_epochs = 25
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid))
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/25
55000/55000 [==============================] - 5s 86us/sample - loss: 0.4842
- accuracy: 0.8318 - val_loss: 0.4174 - val_accuracy: 0.8556
Epoch 2/25
55000/55000 [==============================] - 4s 80us/sample - loss: 0.3787
- accuracy: 0.8648 - val_loss: 0.3772 - val_accuracy: 0.8688
Epoch 3/25
55000/55000 [==============================] - 5s 86us/sample - loss: 0.3451
- accuracy: 0.8768 - val_loss: 0.3684 - val_accuracy: 0.8700
Epoch 4/25
55000/55000 [==============================] - 5s 85us/sample - loss: 0.3235
- accuracy: 0.8842 - val_loss: 0.3519 - val_accuracy: 0.8776
Epoch 5/25
55000/55000 [==============================] - 4s 80us/sample - loss: 0.3067
- accuracy: 0.8910 - val_loss: 0.3438 - val_accuracy: 0.8818
Epoch 6/25
55000/55000 [==============================] - 5s 83us/sample - loss: 0.2942
- accuracy: 0.8945 - val_loss: 0.3414 - val_accuracy: 0.8814
Epoch 7/25
55000/55000 [==============================] - 4s 80us/sample - loss: 0.2832
- accuracy: 0.8990 - val_loss: 0.3360 - val_accuracy: 0.8848
Epoch 8/25
55000/55000 [==============================] - 4s 81us/sample - loss: 0.2742
- accuracy: 0.9022 - val_loss: 0.3309 - val_accuracy: 0.8848
Epoch 9/25
55000/55000 [==============================] - 4s 81us/sample - loss: 0.2664
- accuracy: 0.9041 - val_loss: 0.3279 - val_accuracy: 0.8898
Epoch 10/25
55000/55000 [==============================] - 5s 85us/sample - loss: 0.2595
- accuracy: 0.9071 - val_loss: 0.3295 - val_accuracy: 0.8866
Epoch 11/25
55000/55000 [==============================] - 5s 88us/sample - loss: 0.2537
- accuracy: 0.9094 - val_loss: 0.3244 - val_accuracy: 0.8888
Epoch 12/25
55000/55000 [==============================] - 6s 101us/sample - loss: 0.2486
- accuracy: 0.9109 - val_loss: 0.3234 - val_accuracy: 0.8910
Epoch 13/25
55000/55000 [==============================] - 5s 99us/sample - loss: 0.2442
- accuracy: 0.9124 - val_loss: 0.3230 - val_accuracy: 0.8896
Epoch 14/25
55000/55000 [==============================] - 4s 81us/sample - loss: 0.2404
- accuracy: 0.9148 - val_loss: 0.3235 - val_accuracy: 0.8914
Epoch 15/25
55000/55000 [==============================] - 5s 83us/sample - loss: 0.2370
- accuracy: 0.9164 - val_loss: 0.3201 - val_accuracy: 0.8904
Epoch 16/25
55000/55000 [==============================] - 5s 85us/sample - loss: 0.2337
- accuracy: 0.9167 - val_loss: 0.3209 - val_accuracy: 0.8910
Epoch 17/25
55000/55000 [==============================] - 5s 86us/sample - loss: 0.2313
- accuracy: 0.9185 - val_loss: 0.3189 - val_accuracy: 0.8914
Epoch 18/25
55000/55000 [==============================] - 5s 85us/sample - loss: 0.2284
- accuracy: 0.9187 - val_loss: 0.3212 - val_accuracy: 0.8898
Epoch 19/25
55000/55000 [==============================] - 5s 84us/sample - loss: 0.2265
- accuracy: 0.9199 - val_loss: 0.3198 - val_accuracy: 0.8912
Epoch 20/25
55000/55000 [==============================] - 5s 92us/sample - loss: 0.2246
- accuracy: 0.9213 - val_loss: 0.3183 - val_accuracy: 0.8930
Epoch 21/25
55000/55000 [==============================] - 4s 82us/sample - loss: 0.2229
- accuracy: 0.9218 - val_loss: 0.3180 - val_accuracy: 0.8906
Epoch 22/25
55000/55000 [==============================] - 5s 83us/sample - loss: 0.2213
- accuracy: 0.9228 - val_loss: 0.3176 - val_accuracy: 0.8904
Epoch 23/25
```

```
55000/55000 [==============================] - 5s 88us/sample - loss: 0.2200
- accuracy: 0.9226 - val_loss: 0.3177 - val_accuracy: 0.8922
Epoch 24/25
55000/55000 [==============================] - 5s 85us/sample - loss: 0.2187
- accuracy: 0.9235 - val_loss: 0.3184 - val_accuracy: 0.8908
Epoch 25/25
55000/55000 [==============================] - 6s 106us/sample - loss: 0.2178
- accuracy: 0.9237 - val_loss: 0.3175 - val_accuracy: 0.8908
```

5. Avoiding Overfitting Through Regularization

5.1 ℓ_1 and ℓ_2 regularization

In [61]:

```
# or L1(0.1) for l1 regularization with a factor of 0.1
# or L1_L2(0.1, 0.01) for both L1 and L2 regularization, with factors 0.1 an
d 0.01 respectively

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="elu",
                       kernel_initializer="he_normal",
                       kernel_regularizer=keras.regularizers.l2(0.01)),
    keras.layers.Dense(100, activation="elu",
                       kernel_initializer="he_normal",
                       kernel_regularizer=keras.regularizers.l2(0.01)),
    keras.layers.Dense(10, activation="softmax",
                       kernel_regularizer=keras.regularizers.l2(0.01))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", met
rics=["accuracy"])
n_epochs = 2
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid))
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/2
55000/55000 [==============================] - 9s 165us/sample - loss: 1.5853
- accuracy: 0.8134 - val_loss: 0.7360 - val_accuracy: 0.8208
Epoch 2/2
55000/55000 [==============================] - 8s 149us/sample - loss: 0.7192
- accuracy: 0.8259 - val_loss: 0.6969 - val_accuracy: 0.8322
```

5.2 Dropout

In [62]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accuracy"])
n_epochs = 2
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid))
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/2
55000/55000 [==============================] - 11s 193us/sample - loss: 0.5730 - accuracy: 0.8030 - val_loss: 0.3922 - val_accuracy: 0.8592
Epoch 2/2
55000/55000 [==============================] - 9s 159us/sample - loss: 0.4248 - accuracy: 0.8441 - val_loss: 0.3390 - val_accuracy: 0.8752
```

5.3 Max norm

MaxNorm constrains the weights incident to each hidden unit to have a norm less than or equal to a desired value.

In [63]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="selu", kernel_initializer="lecun_normal",
                           kernel_constraint=keras.constraints.max_norm(1.)),
    keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_normal",
                           kernel_constraint=keras.constraints.max_norm(1.)),
    keras.layers.Dense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accuracy"])
n_epochs = 2
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid))
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/2
55000/55000 [==============================] - 9s 164us/sample - loss: 0.4757 - accuracy: 0.8352 - val_loss: 0.3956 - val_accuracy: 0.8620
Epoch 2/2
55000/55000 [==============================] - 9s 159us/sample - loss: 0.3591 - accuracy: 0.8688 - val_loss: 0.3386 - val_accuracy: 0.8766
```

Chapter 14 – Deep Computer Vision Using Convolutional Neural Networks

This notebook contains all the sample code in chapter 14.

Run in Google Colab (https://colab.research.google.com/github/ageron/handson-ml2/blob/master/14_deep_computer_vision_with_cnns.ipynb)

Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20 and TensorFlow ≥2.0.

In [1]:
```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
    IS_COLAB = True
except Exception:
    IS_COLAB = False

# TensorFlow ≥2.0 is required
import tensorflow as tf
from tensorflow import keras
assert tf.__version__ >= "2.0"

if not tf.config.list_physical_devices('GPU'):
    print("No GPU was detected. CNNs can be very slow without a GPU.")
    if IS_COLAB:
        print("Go to Runtime > Change runtime and select a GPU hardware acce
lerator.")

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)
tf.random.set_seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "cnn"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=30
0):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

```
No GPU was detected. CNNs can be very slow without a GPU.
```

A couple utility functions to plot grayscale and RGB images:

In [2]:
```
def plot_image(image):
    plt.imshow(image, cmap="gray", interpolation="nearest")
    plt.axis("off")

def plot_color_image(image):
    plt.imshow(image, interpolation="nearest")
    plt.axis("off")
```

What is a Convolution?

In [3]:
```
import numpy as np
from sklearn.datasets import load_sample_image

# Load sample images
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1  # vertical line
filters[3, :, :, 1] = 1  # horizontal line

outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")

plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature
map
plt.axis("off") # Not shown in the book
plt.show()
```

In [4]:
```
for image_index in (0, 1):
    for feature_map_index in (0, 1):
        plt.subplot(2, 2, image_index * 2 + feature_map_index + 1)
        plot_image(outputs[image_index, :, :, feature_map_index])

plt.show()
```

In [5]:
```
def crop(images):
    return images[150:220, 130:250]
```

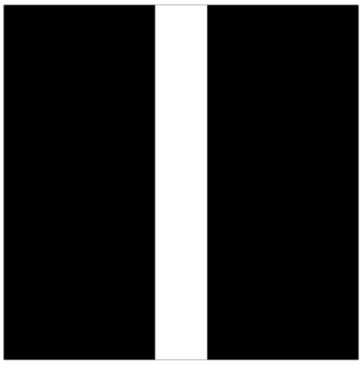
In [6]:
```
plot_image(crop(images[0, :, :, 0]))
save_fig("china_original", tight_layout=False)
plt.show()

for feature_map_index, filename in enumerate(["china_vertical", "china_horiz
ontal"]):
    plot_image(crop(outputs[0, :, :, feature_map_index]))
    save_fig(filename, tight_layout=False)
    plt.show()
```

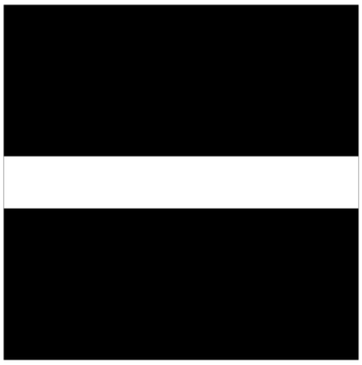
```
Saving figure china_original
```

```
Saving figure china_vertical
```

```
Saving figure china_horizontal
```

In [7]:
```
plot_image(filters[:, :, 0, 0])
plt.show()
plot_image(filters[:, :, 0, 1])
plt.show()
```





Convolutional Layer

Using `keras.layers.Conv2D()` :

In [8]:
```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="SAME", activation="relu")
```

VALID vs SAME padding

Confusingly, `"VALID"` padding means no padding at all.

In [10]:
```
kernel_size = 7
strides = 2

conv_valid = keras.layers.Conv2D(filters=1, kernel_size=kernel_size, strides
=strides, padding="VALID")
conv_same = keras.layers.Conv2D(filters=1, kernel_size=kernel_size, strides=
strides, padding="SAME")
```

Pooling layer

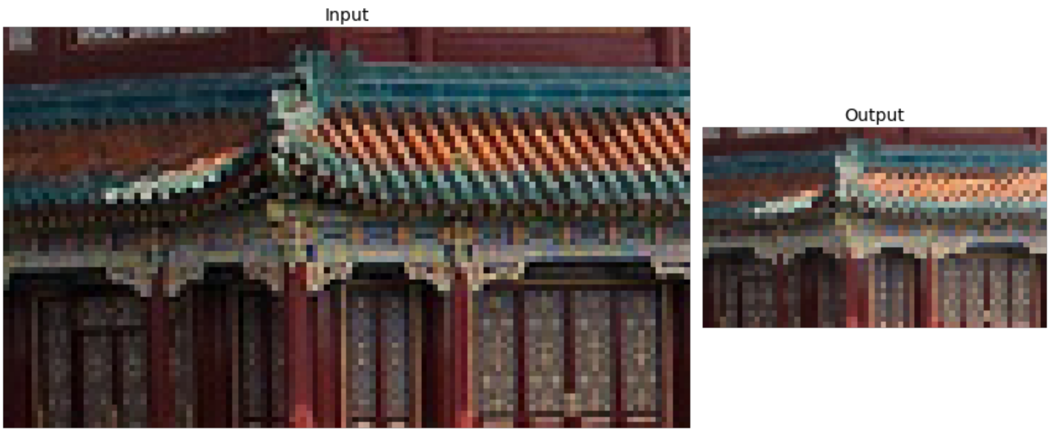
Max pooling

In [11]:
```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

In [12]:
```
cropped_images = np.array([crop(image) for image in images], dtype=np.float3
2)
output = max_pool(cropped_images)
```

In [13]:
```
fig = plt.figure(figsize=(12, 8))
gs = mpl.gridspec.GridSpec(nrows=1, ncols=2, width_ratios=[2, 1])

ax1 = fig.add_subplot(gs[0, 0])
ax1.set_title("Input", fontsize=14)
ax1.imshow(cropped_images[0])  # plot the 1st image
ax1.axis("off")
ax2 = fig.add_subplot(gs[0, 1])
ax2.set_title("Output", fontsize=14)
ax2.imshow(output[0])  # plot the output for the 1st image
ax2.axis("off")
save_fig("china_max_pooling")
plt.show()
```

```
Saving figure china_max_pooling
```

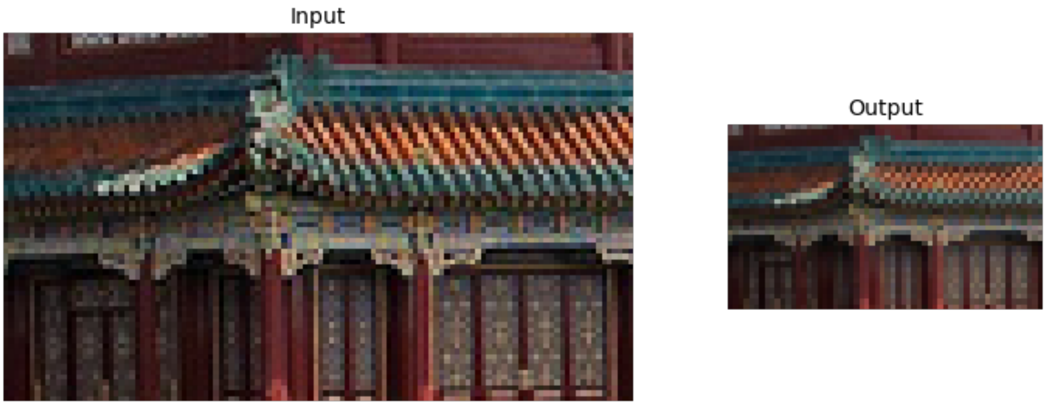


Average pooling

In [14]:
```
avg_pool = keras.layers.AvgPool2D(pool_size=2)
```

In [15]:
```
output_avg = avg_pool(cropped_images)
```

In [16]:
```
fig = plt.figure(figsize=(12, 8))
gs = mpl.gridspec.GridSpec(nrows=1, ncols=2, width_ratios=[2, 1])

ax1 = fig.add_subplot(gs[0, 0])
ax1.set_title("Input", fontsize=14)
ax1.imshow(cropped_images[0])  # plot the 1st image
ax1.axis("off")
ax2 = fig.add_subplot(gs[0, 1])
ax2.set_title("Output", fontsize=14)
ax2.imshow(output_avg[0])  # plot the output for the 1st image
ax2.axis("off")
plt.show()
```



Tackling Fashion MNIST With a CNN

In [17]:
```
(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()
X_train, X_valid = X_train_full[:-5000], X_train_full[-5000:]
y_train, y_valid = y_train_full[:-5000], y_train_full[-5000:]

X_mean = X_train.mean(axis=0, keepdims=True)
X_std = X_train.std(axis=0, keepdims=True) + 1e-7
X_train = (X_train - X_mean) / X_std
X_valid = (X_valid - X_mean) / X_std
X_test = (X_test - X_mean) / X_std

X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]
```

Note : Partial functions allow one to derive a function with x parameters to a function with fewer parameters and fixed values set for the more limited function.

In [18]:
```
from functools import partial

DefaultConv2D = partial(keras.layers.Conv2D,
                        kernel_size=3, activation='relu', padding="SAME")

model = keras.models.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Flatten(),
    keras.layers.Dense(units=128, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units=64, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units=10, activation='softmax'),
])
```

In [19]:
```
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", met
rics=["accuracy"])
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid, y
_valid))
score = model.evaluate(X_test, y_test)
X_new = X_test[:10] # pretend we have new images
y_pred = model.predict(X_new)
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 435s 8ms/sample - loss: 0.7362
- accuracy: 0.7460 - val_loss: 0.3829 - val_accuracy: 0.8654
Epoch 2/10
55000/55000 [==============================] - 492s 9ms/sample - loss: 0.4270
- accuracy: 0.8576 - val_loss: 0.3241 - val_accuracy: 0.8802
Epoch 3/10
55000/55000 [==============================] - 498s 9ms/sample - loss: 0.3709
- accuracy: 0.8749 - val_loss: 0.3086 - val_accuracy: 0.8888
Epoch 4/10
55000/55000 [==============================] - 477s 9ms/sample - loss: 0.3346
- accuracy: 0.8892 - val_loss: 0.2978 - val_accuracy: 0.8894
Epoch 5/10
55000/55000 [==============================] - 477s 9ms/sample - loss: 0.3108
- accuracy: 0.8948 - val_loss: 0.2948 - val_accuracy: 0.8946
Epoch 6/10
55000/55000 [==============================] - 471s 9ms/sample - loss: 0.2981
- accuracy: 0.9005 - val_loss: 0.2871 - val_accuracy: 0.9024
Epoch 7/10
55000/55000 [==============================] - 470s 9ms/sample - loss: 0.2887
- accuracy: 0.9039 - val_loss: 0.2801 - val_accuracy: 0.8980
Epoch 8/10
55000/55000 [==============================] - 551s 10ms/sample - loss: 0.272
6 - accuracy: 0.9074 - val_loss: 0.2889 - val_accuracy: 0.9016
Epoch 9/10
55000/55000 [==============================] - 497s 9ms/sample - loss: 0.2634
- accuracy: 0.9116 - val_loss: 0.2937 - val_accuracy: 0.9000
Epoch 10/10
55000/55000 [==============================] - 534s 10ms/sample - loss: 0.252
9 - accuracy: 0.9153 - val_loss: 0.2959 - val_accuracy: 0.8956
10000/10000 [==============================] - 25s 2ms/sample - loss: 0.3124
- accuracy: 0.8961
```

Using a Pretrained Model

In [32]:
```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

In [33]:
```
images_resized = tf.image.resize(images, [224, 224])
plot_color_image(images_resized[0])
plt.show()
```

In [34]:
```
images_resized = tf.image.resize_with_pad(images, 224, 224, antialias=True)
plot_color_image(images_resized[0])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
```

In [35]:
```
images_resized = tf.image.resize_with_crop_or_pad(images, 224, 224)
plot_color_image(images_resized[0])
plt.show()
```

In [36]:
```
china_box = [0, 0.03, 1, 0.68]
flower_box = [0.19, 0.26, 0.86, 0.7]
images_resized = tf.image.crop_and_resize(images, [china_box, flower_box],
[0, 1], [224, 224])
plot_color_image(images_resized[0])
plt.show()
plot_color_image(images_resized[1])
plt.show()
```

In [37]:
```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
Y_proba = model.predict(inputs)
```

In [38]:
```
Y_proba.shape
```

Out[38]:
```
(2, 1000)
```

In [39]:
```
top_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print("Image #{}".format(image_index))
    for class_id, name, y_proba in top_K[image_index]:
        print("  {} - {:12s} {:.2f}%".format(class_id, name, y_proba * 100))
    print()
```

```
Image #0
  n03877845 - palace       43.39%
  n02825657 - bell_cote    43.08%
  n03781244 - monastery    11.69%

Image #1
  n04522168 - vase         53.97%
  n07930864 - cup          9.52%
  n11939491 - daisy        4.96%
```

In [ ]:

In [1]:
```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Data augmentation

View on TensorFlow.org (https://www.tensorflow.org/tutorials/images/data_augmentation)

Run in Google Colab (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/images/data_augmentation.ipynb)

View source on GitHub (https://github.com/tensorflow/docs/blob/master/site/en/tutorials/images/data_augmentation.ipynb)

Download notebook (https://storage.googleapis.com/tensorflow_docs/docs/site/en/tutorials/images/data_augmentation.ipynb)

Overview

This tutorial demonstrates manual image manipulations and augmentation using `tf.image`.

Data augmentation is a common technique to improve results and avoid overfitting, see Overfitting and Underfitting (../keras/overfit_and_underfit.ipynb) for others.

Setup

In [2]:
```
!pip install -q git+https://github.com/tensorflow/docs
```

In [3]:
```
import urllib

import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras import layers
AUTOTUNE = tf.data.experimental.AUTOTUNE

import tensorflow_docs as tfdocs
import tensorflow_docs.plots

import tensorflow_datasets as tfds

import PIL.Image

import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (12, 5)

import numpy as np
```

Let's check the data augmentation features on an image and then augment a whole dataset later to train a model.

Download [this image (https://commons.wikimedia.org/wiki/File:Felis_catus-cat_on_snow.jpg)](https://commons.wikimedia.org/wiki/File:Felis_catus-cat_on_snow.jpg), by Von.grzanka, for augmentation.

In [4]:
```
image_path = tf.keras.utils.get_file("cat.jpg", "https://storage.googleapis.com/download.tensorflow.org/example_images/320px-Felis_catus-cat_on_snow.jpg")
PIL.Image.open(image_path)
```

Out[4]:

Read and decode the image to tensor format.

In [5]:
```
image_string=tf.io.read_file(image_path)
image=tf.image.decode_jpeg(image_string,channels=3)
```

A function to visualize and compare the original and augmented image side by side.

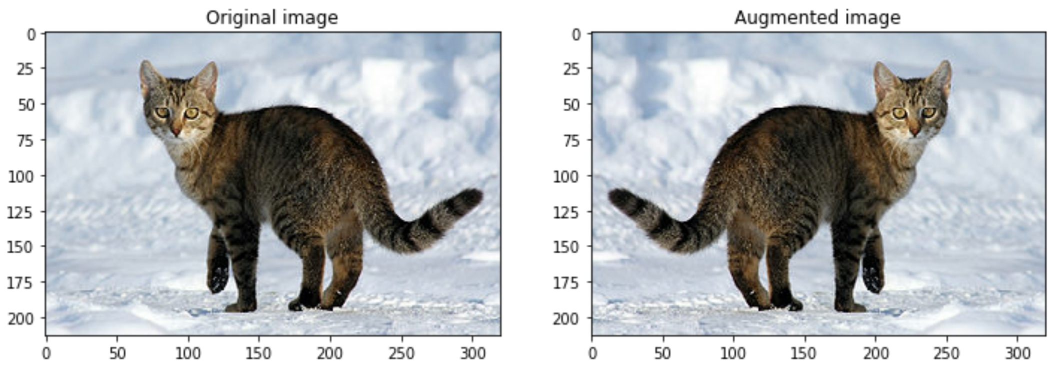
In [6]:
```
def visualize(original, augmented):
  fig = plt.figure()
  plt.subplot(1,2,1)
  plt.title('Original image')
  plt.imshow(original)

  plt.subplot(1,2,2)
  plt.title('Augmented image')
  plt.imshow(augmented)
```

Augment a single image

Flipping the image

Flip the image either vertically or horizontally.

In [7]:
```
flipped = tf.image.flip_left_right(image)
visualize(image, flipped)
```

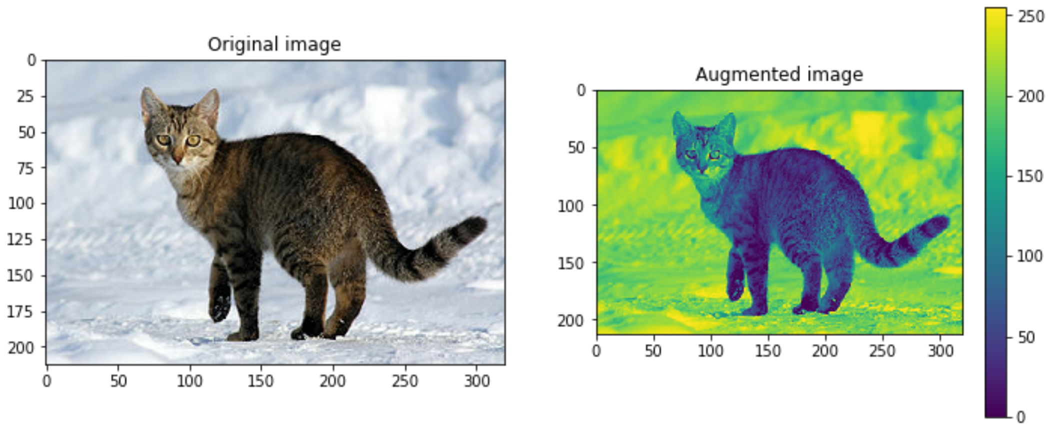


Grayscale the image

Grayscale an image.

In [8]:
```
grayscaled = tf.image.rgb_to_grayscale(image)
visualize(image, tf.squeeze(grayscaled))
plt.colorbar()
```

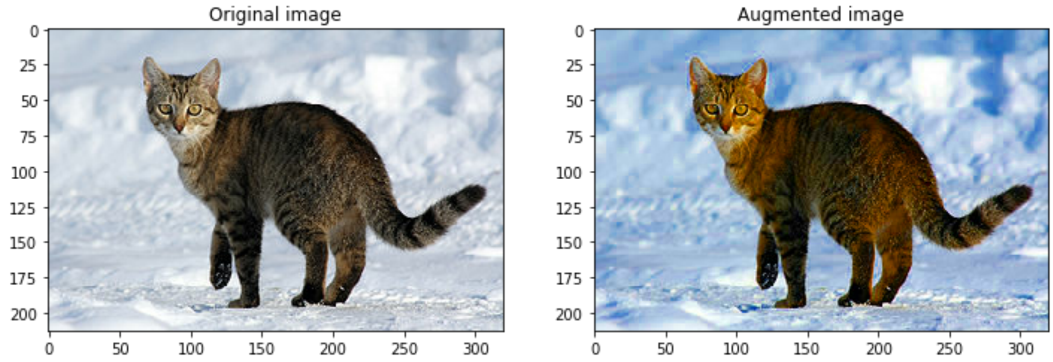
Out[8]: <matplotlib.colorbar.Colorbar at 0x1693a591dc8>



Saturate the image

Saturate an image by providing a saturation factor.

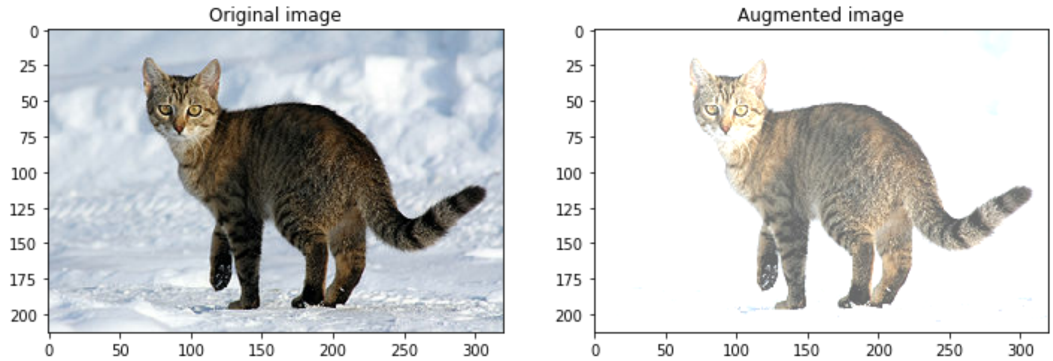
In [9]:
```
saturated = tf.image.adjust_saturation(image, 3)
visualize(image, saturated)
```



Change image brightness

Change the brightness of image by providing a brightness factor.

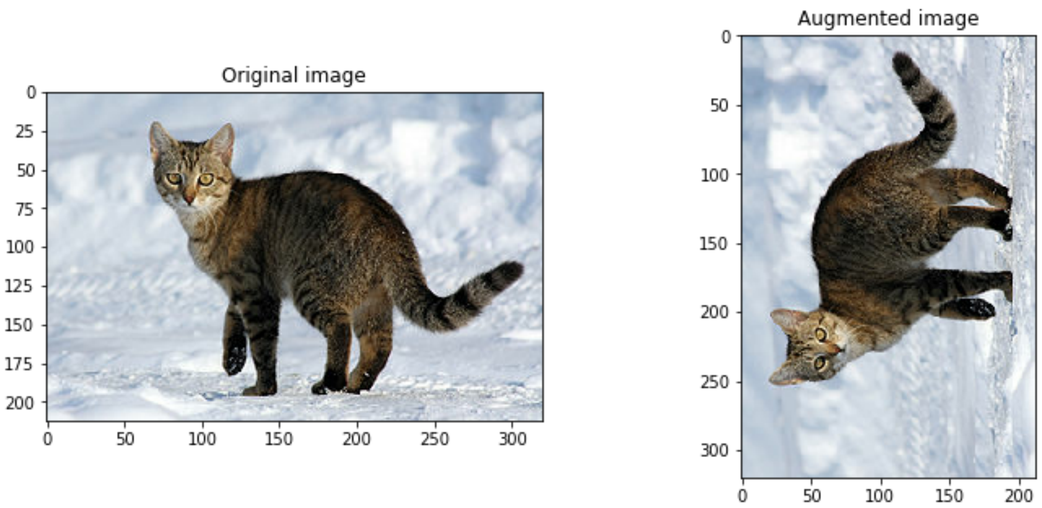
In [10]:
```
bright = tf.image.adjust_brightness(image, 0.4)
visualize(image, bright)
```



Rotate the image

Rotate an image by 90 degrees.

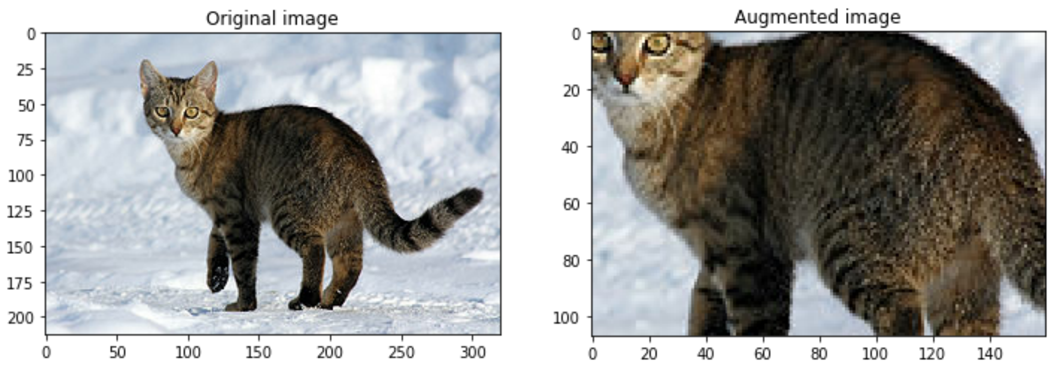
In [11]:
```
rotated = tf.image.rot90(image)
visualize(image, rotated)
```



Center crop the image

Crop the image from center upto the image part you desire.

In [12]:
```
cropped = tf.image.central_crop(image, central_fraction=0.5)
visualize(image,cropped)
```



See the `tf.image` reference for details about available augmentation options.

Augment a dataset and train a model with it

Train a model on an augmented dataset.

Note: The problem solved here is somewhat artificial. It trains a densely connected network to be shift invariant by jittering the input images. It's much more efficient to use convolutional layers instead.

In [13]:
```
dataset, info =  tfds.load('mnist', as_supervised=True, with_info=True)
train_dataset, test_dataset = dataset['train'], dataset['test']

num_train_examples= info.splits['train'].num_examples
```

Downloading and preparing dataset mnist/3.0.1 (download: 11.06 MiB, generated: 21.00 MiB, total: 32.06 MiB) to C:\Users\gcont\tensorflow_datasets\mnist\3.0.1...

```
WARNING:absl:Dataset mnist is hosted on GCS. It will automatically be downloaded to your
local data directory. If you'd instead prefer to read directly from our public
GCS bucket (recommended if you're running on GCP), you can instead pass
`try_gcs=True` to `tfds.load` or set `data_dir=gs://tfds-data/datasets`.
```

Dataset mnist downloaded and prepared to C:\Users\gcont\tensorflow_datasets\mnist\3.0.1. Subsequent calls will reuse this data.

Write a function to augment the images. Map it over the the dataset. This returns a dataset that augments the data on the fly.

In [14]:
```
def convert(image, label):
  image = tf.image.convert_image_dtype(image, tf.float32) # Cast and normalize the image to [0,1]
  return image, label

def augment(image,label):
  image,label = convert(image, label)
  image = tf.image.convert_image_dtype(image, tf.float32) # Cast and normalize the image to [0,1]
  image = tf.image.resize_with_crop_or_pad(image, 34, 34) # Add 6 pixels of padding
  image = tf.image.random_crop(image, size=[28, 28, 1]) # Random crop back to 28x28
  image = tf.image.random_brightness(image, max_delta=0.5) # Random brightness

  return image,label
```

In [15]:
```
BATCH_SIZE = 64
# Only use a subset of the data so it's easier to overfit, for this tutorial
NUM_EXAMPLES = 2048
```

Create the augmented dataset.

In [16]:
```
augmented_train_batches = (
    train_dataset
    # Only train on a subset, so you can quickly see the effect.
    .take(NUM_EXAMPLES)
    .cache()
    .shuffle(num_train_examples//4)
    # The augmentation is added here.
    .map(augment, num_parallel_calls=AUTOTUNE)
    .batch(BATCH_SIZE)
    .prefetch(AUTOTUNE)
)
```

And a non-augmented one for comparison.

In [17]:
```
non_augmented_train_batches = (
    train_dataset
    # Only train on a subset, so you can quickly see the effect.
    .take(NUM_EXAMPLES)
    .cache()
    .shuffle(num_train_examples//4)
    # No augmentation.
    .map(convert, num_parallel_calls=AUTOTUNE)
    .batch(BATCH_SIZE)
    .prefetch(AUTOTUNE)
)
```

Setup the validation dataset. This doesn't change whether or not you're using the augmentation.

In [18]:
```
validation_batches = (
    test_dataset
    .map(convert, num_parallel_calls=AUTOTUNE)
    .batch(2*BATCH_SIZE)
)
```

Create and compile the model. The model is a two layered, fully-connected neural network without convolution.

In [19]:
```
def make_model():
  model = tf.keras.Sequential([
      layers.Flatten(input_shape=(28, 28, 1)),
      layers.Dense(4096, activation='relu'),
      layers.Dense(4096, activation='relu'),
      layers.Dense(10)
  ])
  model.compile(optimizer = 'adam',
                loss=tf.losses.SparseCategoricalCrossentropy(from_logits=Tru
e),
                metrics=['accuracy'])
  return model
```

Train the model, **without** augmentation:

In [20]:
```
model_without_aug = make_model()

no_aug_history = model_without_aug.fit(non_augmented_train_batches, epochs=5
0, validation_data=validation_batches)
```

```
Epoch 1/50
32/32 [==============================] - 10s 300ms/step - loss: 0.9558 - accu
racy: 0.7285 - val_loss: 0.4432 - val_accuracy: 0.8653
Epoch 2/50
32/32 [==============================] - 11s 335ms/step - loss: 0.2270 - accu
racy: 0.9282 - val_loss: 0.3182 - val_accuracy: 0.9058
Epoch 3/50
32/32 [==============================] - 6s 202ms/step - loss: 0.0797 - accur
acy: 0.9756 - val_loss: 0.2658 - val_accuracy: 0.9237
Epoch 4/50
32/32 [==============================] - 6s 200ms/step - loss: 0.0364 - accur
acy: 0.9893 - val_loss: 0.2954 - val_accuracy: 0.9280
Epoch 5/50
32/32 [==============================] - 7s 211ms/step - loss: 0.0228 - accur
acy: 0.9932 - val_loss: 0.3575 - val_accuracy: 0.9222
Epoch 6/50
32/32 [==============================] - 7s 216ms/step - loss: 0.0249 - accur
acy: 0.9893 - val_loss: 0.4029 - val_accuracy: 0.9183
Epoch 7/50
32/32 [==============================] - 7s 216ms/step - loss: 0.0265 - accur
acy: 0.9922 - val_loss: 0.3789 - val_accuracy: 0.9238
Epoch 8/50
32/32 [==============================] - 7s 216ms/step - loss: 0.0367 - accur
acy: 0.9868 - val_loss: 0.4761 - val_accuracy: 0.9010
Epoch 9/50
32/32 [==============================] - 7s 218ms/step - loss: 0.0650 - accur
acy: 0.9810 - val_loss: 0.3904 - val_accuracy: 0.9238
Epoch 10/50
32/32 [==============================] - 7s 213ms/step - loss: 0.0362 - accur
acy: 0.9893 - val_loss: 0.3626 - val_accuracy: 0.9272
Epoch 11/50
32/32 [==============================] - 7s 220ms/step - loss: 0.0268 - accur
acy: 0.9907 - val_loss: 0.4220 - val_accuracy: 0.9200
Epoch 12/50
32/32 [==============================] - 7s 223ms/step - loss: 0.0484 - accur
acy: 0.9829 - val_loss: 0.4235 - val_accuracy: 0.9141
Epoch 13/50
32/32 [==============================] - 7s 231ms/step - loss: 0.0402 - accur
acy: 0.9902 - val_loss: 0.4773 - val_accuracy: 0.9118
Epoch 14/50
32/32 [==============================] - 8s 237ms/step - loss: 0.0196 - accur
acy: 0.9951 - val_loss: 0.4108 - val_accuracy: 0.9215
Epoch 15/50
32/32 [==============================] - 8s 250ms/step - loss: 0.0080 - accur
acy: 0.9980 - val_loss: 0.3638 - val_accuracy: 0.9299
Epoch 16/50
32/32 [==============================] - 9s 285ms/step - loss: 0.0077 - accur
acy: 0.9971 - val_loss: 0.3838 - val_accuracy: 0.9290
Epoch 17/50
32/32 [==============================] - 11s 338ms/step - loss: 0.0117 - accu
racy: 0.9966 - val_loss: 0.3424 - val_accuracy: 0.9380
Epoch 18/50
32/32 [==============================] - 10s 310ms/step - loss: 0.0058 - accu
racy: 0.9980 - val_loss: 0.5034 - val_accuracy: 0.9218
Epoch 19/50
32/32 [==============================] - 9s 269ms/step - loss: 0.0332 - accur
acy: 0.9922 - val_loss: 0.4584 - val_accuracy: 0.9254
Epoch 20/50
32/32 [==============================] - 9s 284ms/step - loss: 0.0271 - accur
acy: 0.9917 - val_loss: 0.5029 - val_accuracy: 0.9201
Epoch 21/50
32/32 [==============================] - 9s 290ms/step - loss: 0.0163 - accur
acy: 0.9937 - val_loss: 0.4485 - val_accuracy: 0.9294
Epoch 22/50
32/32 [==============================] - 11s 335ms/step - loss: 0.0205 - accu
racy: 0.9946 - val_loss: 0.5501 - val_accuracy: 0.9161
Epoch 23/50
32/32 [==============================] - 10s 315ms/step - loss: 0.0354 - accu
```

```
racy: 0.9912 - val_loss: 0.4424 - val_accuracy: 0.9284
Epoch 24/50
32/32 [==============================] - 14s 424ms/step - loss: 0.0380 - accu
racy: 0.9893 - val_loss: 0.5404 - val_accuracy: 0.9182
Epoch 25/50
32/32 [==============================] - 12s 388ms/step - loss: 0.0409 - accu
racy: 0.9873 - val_loss: 0.4819 - val_accuracy: 0.9208
Epoch 26/50
32/32 [==============================] - 11s 340ms/step - loss: 0.0162 - accu
racy: 0.9946 - val_loss: 0.4960 - val_accuracy: 0.9247
Epoch 27/50
32/32 [==============================] - 11s 336ms/step - loss: 0.0303 - accu
racy: 0.9897 - val_loss: 0.6399 - val_accuracy: 0.9092
Epoch 28/50
32/32 [==============================] - 10s 313ms/step - loss: 0.0293 - accu
racy: 0.9941 - val_loss: 0.5493 - val_accuracy: 0.9231
Epoch 29/50
32/32 [==============================] - 11s 342ms/step - loss: 0.0518 - accu
racy: 0.9893 - val_loss: 0.5894 - val_accuracy: 0.9119
Epoch 30/50
32/32 [==============================] - 9s 272ms/step - loss: 0.0171 - accur
acy: 0.9951 - val_loss: 0.4361 - val_accuracy: 0.9327
Epoch 31/50
32/32 [==============================] - 9s 290ms/step - loss: 0.0071 - accur
acy: 0.9980 - val_loss: 0.4741 - val_accuracy: 0.9277
Epoch 32/50
32/32 [==============================] - 9s 277ms/step - loss: 0.0049 - accur
acy: 0.9990 - val_loss: 0.4174 - val_accuracy: 0.9367
Epoch 33/50
32/32 [==============================] - 9s 280ms/step - loss: 6.7054e-04 - a
ccuracy: 0.9995 - val_loss: 0.4276 - val_accuracy: 0.9356
Epoch 34/50
32/32 [==============================] - 9s 278ms/step - loss: 3.4375e-04 - a
ccuracy: 1.0000 - val_loss: 0.4298 - val_accuracy: 0.9352
Epoch 35/50
32/32 [==============================] - 9s 276ms/step - loss: 9.0659e-05 - a
ccuracy: 1.0000 - val_loss: 0.4292 - val_accuracy: 0.9370
Epoch 36/50
32/32 [==============================] - 9s 275ms/step - loss: 5.5513e-05 - a
ccuracy: 1.0000 - val_loss: 0.4296 - val_accuracy: 0.9373
Epoch 37/50
32/32 [==============================] - 10s 303ms/step - loss: 4.7191e-05 -
accuracy: 1.0000 - val_loss: 0.4304 - val_accuracy: 0.9373
Epoch 38/50
32/32 [==============================] - 12s 371ms/step - loss: 4.1376e-05 -
accuracy: 1.0000 - val_loss: 0.4313 - val_accuracy: 0.9372
Epoch 39/50
32/32 [==============================] - 11s 329ms/step - loss: 3.7425e-05 -
accuracy: 1.0000 - val_loss: 0.4322 - val_accuracy: 0.9371
Epoch 40/50
32/32 [==============================] - 9s 296ms/step - loss: 3.4229e-05 - a
ccuracy: 1.0000 - val_loss: 0.4330 - val_accuracy: 0.9365
Epoch 41/50
32/32 [==============================] - 13s 394ms/step - loss: 3.1477e-05 -
accuracy: 1.0000 - val_loss: 0.4340 - val_accuracy: 0.9365
Epoch 42/50
32/32 [==============================] - 11s 348ms/step - loss: 2.9094e-05 -
accuracy: 1.0000 - val_loss: 0.4349 - val_accuracy: 0.9365
Epoch 43/50
32/32 [==============================] - 10s 307ms/step - loss: 2.7075e-05 -
accuracy: 1.0000 - val_loss: 0.4359 - val_accuracy: 0.9366
Epoch 44/50
32/32 [==============================] - 10s 309ms/step - loss: 2.5183e-05 -
accuracy: 1.0000 - val_loss: 0.4369 - val_accuracy: 0.9367
Epoch 45/50
32/32 [==============================] - 10s 303ms/step - loss: 2.3403e-05 -
accuracy: 1.0000 - val_loss: 0.4379 - val_accuracy: 0.9366
Epoch 46/50
```

```
32/32 [==============================] - 9s 289ms/step - loss: 2.1840e-05 - a
ccuracy: 1.0000 - val_loss: 0.4393 - val_accuracy: 0.9366
Epoch 47/50
32/32 [==============================] - 10s 319ms/step - loss: 2.0270e-05 -
accuracy: 1.0000 - val_loss: 0.4404 - val_accuracy: 0.9365
Epoch 48/50
32/32 [==============================] - 9s 288ms/step - loss: 1.8933e-05 - a
ccuracy: 1.0000 - val_loss: 0.4418 - val_accuracy: 0.9365
Epoch 49/50
32/32 [==============================] - 9s 286ms/step - loss: 1.7515e-05 - a
ccuracy: 1.0000 - val_loss: 0.4433 - val_accuracy: 0.9364
Epoch 50/50
32/32 [==============================] - 12s 382ms/step - loss: 1.6231e-05 -
accuracy: 1.0000 - val_loss: 0.4449 - val_accuracy: 0.9364
```

Train it again with augmentation:

In [21]:
```
model_with_aug = make_model()

aug_history = model_with_aug.fit(augmented_train_batches, epochs=50, validat
ion_data=validation_batches)
```

```
Epoch 1/50
32/32 [==============================] - 12s 379ms/step - loss: 2.3040 - accu
racy: 0.3076 - val_loss: 1.1106 - val_accuracy: 0.7139
Epoch 2/50
32/32 [==============================] - 10s 311ms/step - loss: 1.3123 - accu
racy: 0.5674 - val_loss: 0.7567 - val_accuracy: 0.7586
Epoch 3/50
32/32 [==============================] - 10s 326ms/step - loss: 0.9369 - accu
racy: 0.6807 - val_loss: 0.5183 - val_accuracy: 0.8437
Epoch 4/50
32/32 [==============================] - 10s 322ms/step - loss: 0.7522 - accu
racy: 0.7412 - val_loss: 0.3666 - val_accuracy: 0.8905
Epoch 5/50
32/32 [==============================] - 10s 324ms/step - loss: 0.6499 - accu
racy: 0.7778 - val_loss: 0.3120 - val_accuracy: 0.9111
Epoch 6/50
32/32 [==============================] - 9s 296ms/step - loss: 0.5878 - accur
acy: 0.7979 - val_loss: 0.3062 - val_accuracy: 0.9061
Epoch 7/50
32/32 [==============================] - 13s 402ms/step - loss: 0.5146 - accu
racy: 0.8413 - val_loss: 0.2507 - val_accuracy: 0.9240
Epoch 8/50
32/32 [==============================] - 11s 351ms/step - loss: 0.5211 - accu
racy: 0.8306 - val_loss: 0.3510 - val_accuracy: 0.8806
Epoch 9/50
32/32 [==============================] - 12s 379ms/step - loss: 0.5186 - accu
racy: 0.8218 - val_loss: 0.2766 - val_accuracy: 0.9113
Epoch 10/50
32/32 [==============================] - 10s 323ms/step - loss: 0.4263 - accu
racy: 0.8652 - val_loss: 0.2462 - val_accuracy: 0.9235
Epoch 11/50
32/32 [==============================] - 14s 425ms/step - loss: 0.4073 - accu
racy: 0.8672 - val_loss: 0.2096 - val_accuracy: 0.9346
Epoch 12/50
32/32 [==============================] - 11s 350ms/step - loss: 0.3593 - accu
racy: 0.8857 - val_loss: 0.2102 - val_accuracy: 0.9331
Epoch 13/50
32/32 [==============================] - 9s 288ms/step - loss: 0.3796 - accur
acy: 0.8667 - val_loss: 0.2231 - val_accuracy: 0.9314
Epoch 14/50
32/32 [==============================] - 9s 280ms/step - loss: 0.3449 - accur
acy: 0.8921 - val_loss: 0.2314 - val_accuracy: 0.9263
Epoch 15/50
32/32 [==============================] - 9s 283ms/step - loss: 0.3128 - accur
acy: 0.8931 - val_loss: 0.2220 - val_accuracy: 0.9257
Epoch 16/50
32/32 [==============================] - 9s 280ms/step - loss: 0.3405 - accur
acy: 0.8838 - val_loss: 0.2003 - val_accuracy: 0.9357
Epoch 17/50
32/32 [==============================] - 9s 291ms/step - loss: 0.2647 - accur
acy: 0.9136 - val_loss: 0.1985 - val_accuracy: 0.9406
Epoch 18/50
32/32 [==============================] - 13s 411ms/step - loss: 0.3147 - accu
racy: 0.8896 - val_loss: 0.2109 - val_accuracy: 0.9332
Epoch 19/50
32/32 [==============================] - 9s 281ms/step - loss: 0.2909 - accur
acy: 0.9038 - val_loss: 0.1985 - val_accuracy: 0.9380
Epoch 20/50
32/32 [==============================] - 9s 287ms/step - loss: 0.2791 - accur
acy: 0.9048 - val_loss: 0.1897 - val_accuracy: 0.9414
Epoch 21/50
32/32 [==============================] - 9s 285ms/step - loss: 0.2887 - accur
acy: 0.8994 - val_loss: 0.1900 - val_accuracy: 0.9405
Epoch 22/50
32/32 [==============================] - 11s 339ms/step - loss: 0.2550 - accu
racy: 0.9209 - val_loss: 0.1785 - val_accuracy: 0.9459
Epoch 23/50
32/32 [==============================] - 12s 373ms/step - loss: 0.2853 - accu
```

```
racy: 0.9077 - val_loss: 0.1728 - val_accuracy: 0.9477
Epoch 24/50
32/32 [==============================] - 11s 346ms/step - loss: 0.2869 - accu
racy: 0.9072 - val_loss: 0.1935 - val_accuracy: 0.9333
Epoch 25/50
32/32 [==============================] - 12s 373ms/step - loss: 0.2534 - accu
racy: 0.9141 - val_loss: 0.1925 - val_accuracy: 0.9367
Epoch 26/50
32/32 [==============================] - 11s 340ms/step - loss: 0.2565 - accu
racy: 0.9219 - val_loss: 0.1866 - val_accuracy: 0.9422
Epoch 27/50
32/32 [==============================] - 9s 293ms/step - loss: 0.2349 - accur
acy: 0.9194 - val_loss: 0.1566 - val_accuracy: 0.9509
Epoch 28/50
32/32 [==============================] - 11s 348ms/step - loss: 0.2156 - accu
racy: 0.9268 - val_loss: 0.1638 - val_accuracy: 0.9495
Epoch 29/50
32/32 [==============================] - 10s 302ms/step - loss: 0.2387 - accu
racy: 0.9219 - val_loss: 0.1692 - val_accuracy: 0.9482
Epoch 30/50
32/32 [==============================] - 11s 334ms/step - loss: 0.2155 - accu
racy: 0.9336 - val_loss: 0.1677 - val_accuracy: 0.9520
Epoch 31/50
32/32 [==============================] - 9s 296ms/step - loss: 0.1932 - accur
acy: 0.9321 - val_loss: 0.1833 - val_accuracy: 0.9463
Epoch 32/50
32/32 [==============================] - 10s 301ms/step - loss: 0.1790 - accu
racy: 0.9409 - val_loss: 0.1658 - val_accuracy: 0.9507
Epoch 33/50
32/32 [==============================] - 11s 347ms/step - loss: 0.2296 - accu
racy: 0.9282 - val_loss: 0.1741 - val_accuracy: 0.9459
Epoch 34/50
32/32 [==============================] - 11s 352ms/step - loss: 0.1880 - accu
racy: 0.9370 - val_loss: 0.1764 - val_accuracy: 0.9473
Epoch 35/50
32/32 [==============================] - 9s 278ms/step - loss: 0.2127 - accur
acy: 0.9341 - val_loss: 0.1917 - val_accuracy: 0.9435
Epoch 36/50
32/32 [==============================] - 11s 342ms/step - loss: 0.1955 - accu
racy: 0.9370 - val_loss: 0.1766 - val_accuracy: 0.9482
Epoch 37/50
32/32 [==============================] - 10s 318ms/step - loss: 0.1772 - accu
racy: 0.9429 - val_loss: 0.1575 - val_accuracy: 0.9526
Epoch 38/50
32/32 [==============================] - 9s 276ms/step - loss: 0.1931 - accur
acy: 0.9390 - val_loss: 0.1689 - val_accuracy: 0.9510
Epoch 39/50
32/32 [==============================] - 9s 272ms/step - loss: 0.2634 - accur
acy: 0.9219 - val_loss: 0.1764 - val_accuracy: 0.9464
Epoch 40/50
32/32 [==============================] - 9s 278ms/step - loss: 0.2062 - accur
acy: 0.9321 - val_loss: 0.1577 - val_accuracy: 0.9514
Epoch 41/50
32/32 [==============================] - 9s 270ms/step - loss: 0.1666 - accur
acy: 0.9419 - val_loss: 0.1740 - val_accuracy: 0.9489
Epoch 42/50
32/32 [==============================] - 9s 287ms/step - loss: 0.1860 - accur
acy: 0.9399 - val_loss: 0.1524 - val_accuracy: 0.9536
Epoch 43/50
32/32 [==============================] - 9s 269ms/step - loss: 0.1613 - accur
acy: 0.9497 - val_loss: 0.1756 - val_accuracy: 0.9496
Epoch 44/50
32/32 [==============================] - 9s 278ms/step - loss: 0.1695 - accur
acy: 0.9434 - val_loss: 0.1503 - val_accuracy: 0.9540
Epoch 45/50
32/32 [==============================] - 9s 270ms/step - loss: 0.1501 - accur
acy: 0.9517 - val_loss: 0.1672 - val_accuracy: 0.9491
Epoch 46/50
```

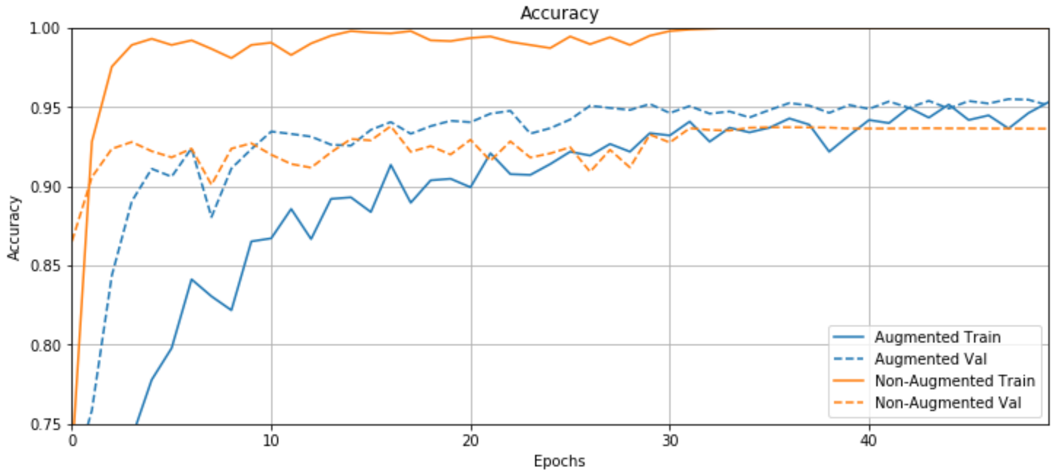
```
32/32 [==============================] - 9s 277ms/step - loss: 0.1519 - accur
acy: 0.9419 - val_loss: 0.1701 - val_accuracy: 0.9539
Epoch 47/50
32/32 [==============================] - 9s 270ms/step - loss: 0.1752 - accur
acy: 0.9448 - val_loss: 0.1597 - val_accuracy: 0.9523
Epoch 48/50
32/32 [==============================] - 9s 276ms/step - loss: 0.1835 - accur
acy: 0.9365 - val_loss: 0.1495 - val_accuracy: 0.9551
Epoch 49/50
32/32 [==============================] - 9s 280ms/step - loss: 0.1559 - accur
acy: 0.9463 - val_loss: 0.1629 - val_accuracy: 0.9547
Epoch 50/50
32/32 [==============================] - 9s 273ms/step - loss: 0.1431 - accur
acy: 0.9531 - val_loss: 0.1663 - val_accuracy: 0.9511
```

Conclusion:

In this example the augmented model converges to an accuracy ~95% on validation set. This is slightly higher (+1%) than the model trained without data augmentation.

In [22]:
```
plotter = tfdocs.plots.HistoryPlotter()
plotter.plot({"Augmented": aug_history, "Non-Augmented": no_aug_history}, me
tric = "accuracy")
plt.title("Accuracy")
plt.ylim([0.75,1])
```

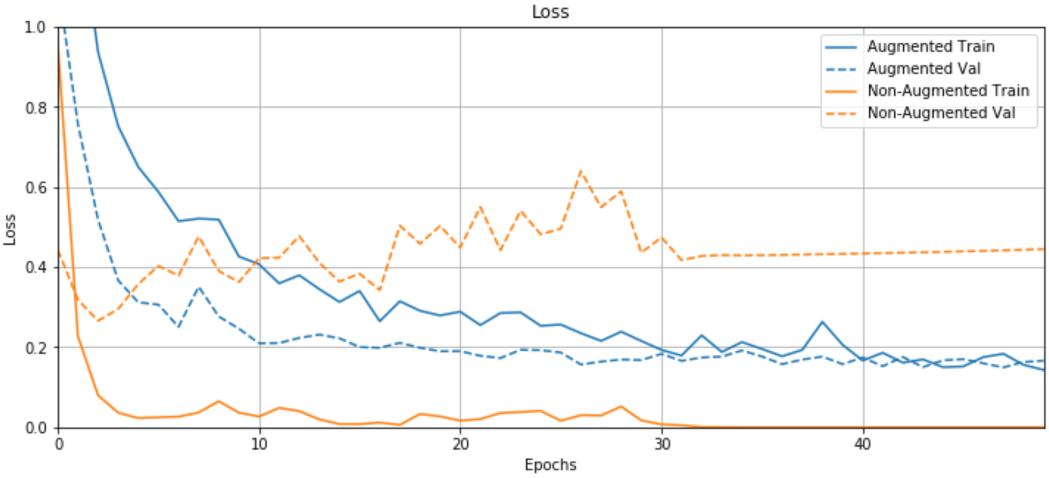
Out[22]:
```
(0.75, 1)
```



In terms of loss, the non-augmented model is obviously in the overfitting regime. The augmented model, while a few epoch slower, is still training correctly and clearly not overfitting.

In [23]:
```
plotter = tfdocs.plots.HistoryPlotter()
plotter.plot({"Augmented": aug_history, "Non-Augmented": no_aug_history}, me
tric = "loss")
plt.title("Loss")
plt.ylim([0,1])
```

Out[23]:
```
(0, 1)
```



In [1]:
```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Text generation with an RNN

[View on TensorFlow.org (https://www.tensorflow.org/tutorials/text/text_generation)]

[Run in Google Colab (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/text/text_generation.ipynb)]

[View source on GitHub (https://github.com/tensorflow/docs/blob/master/site/en/tutorials/text/text_generation.ipynb)]

[Download notebook (https://storage.googleapis.com/tensorflow_docs/docs/site/en/tutorials/text/text_generation.ipynb)]

This tutorial demonstrates how to generate text using a character-based RNN. We will work with a dataset of Shakespeare's writing from Andrej Karpathy's [The Unreasonable Effectiveness of Recurrent Neural Networks (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)](http://karpathy.github.io/2015/05/21/rnn-effectiveness/). Given a sequence of characters from this data ("Shakespear"), train a model to predict the next character in the sequence ("e"). Longer sequences of text can be generated by calling the model repeatedly.

Note: Enable GPU acceleration to execute this notebook faster. In Colab: *Runtime > Change runtime type > Hardware acclerator > GPU*. If running locally make sure TensorFlow version >= 1.11.

This tutorial includes runnable code implemented using [tf.keras (https://www.tensorflow.org/programmers_guide/keras)](https://www.tensorflow.org/programmers_guide/keras) and [eager execution (https://www.tensorflow.org/programmers_guide/eager)](https://www.tensorflow.org/programmers_guide/eager). The following is sample output when the model in this tutorial trained for 30 epochs, and started with the string "Q":

```
QUEENE:
I had thought thou hadst a Roman; for the oracle,
Thus by All bids the man against the word,
Which are so weak of care, by old care done;
Your children were in your holy love,
And the precipitation through the bleeding throne.

BISHOP OF ELY:
Marry, and will, my lord, to weep in such a one were prettiest;
Yet now I was adopted heir
Of the world's lamentable day,
To watch the next way with his father with his face?

ESCALUS:
The cause why then we are all resolved more sons.

VOLUMNIA:
O, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no,
it is no sin it should be dead,
And love and pale as any will to that word.

QUEEN ELIZABETH:
But how long have I heard the soul for this world,
And show his hands of life be proved to stand.

PETRUCHIO:
I say he look'd on, if I must be content
To stay him from the fatal of our country's bliss.
His lordship pluck'd from this sentence then for prey,
And then let us twain, being the moon,
were she such a case as fills m
```

While some of the sentences are grammatical, most do not make sense. The model has not learned the meaning of words, but consider:

- The model is character-based. When training started, the model did not know how to spell an English word, or that words were even a unit of text.
- The structure of the output resembles a play—blocks of text generally begin with a speaker name, in all capital letters similar to the dataset.
- As demonstrated below, the model is trained on small batches of text (100 characters each), and is still able to generate a longer sequence of text with coherent structure.

Setup

Import TensorFlow and other libraries

In [2]:
```
import tensorflow as tf

import numpy as np
import os
import time
```

Download the Shakespeare dataset

Change the following line to run this code on your own data.

In [3]:
```
path_to_file = tf.keras.utils.get_file('shakespeare.txt', 'https://storage.g
oogleapis.com/download.tensorflow.org/data/shakespeare.txt')
```

Read the data

First, look in the text:

In [4]:
```
# Read, then decode for py2 compat.
text = open(path_to_file, 'rb').read().decode(encoding='utf-8')
# length of text is the number of characters in it
print ('Length of text: {} characters'.format(len(text)))
```

```
Length of text: 1115394 characters
```

In [5]:
```
# Take a look at the first 250 characters in text
print(text[:250])
```

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.
```

In [6]:
```
# The unique characters in the file
vocab = sorted(set(text))
print ('{} unique characters'.format(len(vocab)))
```

```
65 unique characters
```

Process the text

Vectorize the text

Before training, we need to map strings to a numerical representation. Create two lookup tables: one mapping characters to numbers, and another for numbers to characters.

In [7]:
```
# Creating a mapping from unique characters to indices
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)

text_as_int = np.array([char2idx[c] for c in text])
```

Now we have an integer representation for each character. Notice that we mapped the character as indexes from 0 to `len(unique)` .

In [8]:
```
print('{')
for char,_ in zip(char2idx, range(20)):
    print('  {:4s}: {:3d},'.format(repr(char), char2idx[char]))
print('  ...\n}')
```

```
{
  '\n':   0,
  ' ' :   1,
  '!' :   2,
  '$' :   3,
  '&' :   4,
  "'" :   5,
  ',' :   6,
  '-' :   7,
  '.' :   8,
  '3' :   9,
  ':' :  10,
  ';' :  11,
  '?' :  12,
  'A' :  13,
  'B' :  14,
  'C' :  15,
  'D' :  16,
  'E' :  17,
  'F' :  18,
  'G' :  19,
  ...
}
```

In [9]:
```
# Show how the first 13 characters from the text are mapped to integers
print ('{} ---- characters mapped to int ---- > {}'.format(repr(text[:13]),
text_as_int[:13]))
```

```
'First Citizen' ---- characters mapped to int ---- > [18 47 56 57 58  1 15 47
58 47 64 43 52]
```

The prediction task

Given a character, or a sequence of characters, what is the most probable next character? This is the task we're training the model to perform. The input to the model will be a sequence of characters, and we train the model to predict the output—the following character at each time step.

Since RNNs maintain an internal state that depends on the previously seen elements, given all the characters computed until this moment, what is the next character?

Create training examples and targets

Next divide the text into example sequences. Each input sequence will contain `seq_length` characters from the text.

For each input sequence, the corresponding targets contain the same length of text, except shifted one character to the right.

So break the text into chunks of `seq_length+1`. For example, say `seq_length` is 4 and our text is "Hello". The input sequence would be "Hell", and the target sequence "ello".

To do this first use the `tf.data.Dataset.from_tensor_slices` function to convert the text vector into a stream of character indices.

In [10]:
```
# The maximum length sentence we want for a single input in characters
seq_length = 100
examples_per_epoch = len(text)//(seq_length+1)

# Create training examples / targets
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)

for i in char_dataset.take(5):
  print(idx2char[i.numpy()])
```

```
F
i
r
s
t
```

The `batch` method lets us easily convert these individual characters to sequences of the desired size.

In [11]:
```
sequences = char_dataset.batch(seq_length+1, drop_remainder=True)

for item in sequences.take(5):
  print(repr(''.join(idx2char[item.numpy()])))
```

```
'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpea
k, speak.\n\nFirst Citizen:\nYou '
'are all resolved rather to die than to famish?\n\nAll:\nResolved. resolved.\
n\nFirst Citizen:\nFirst, you k'
"now Caius Marcius is chief enemy to the people.\n\nAll:\nWe know't, we know'
t.\n\nFirst Citizen:\nLet us ki"
"ll him, and we'll have corn at our own price.\nIs't a verdict?\n\nAll:\nNo m
ore talking on't; let it be d"
'one: away, away!\n\nSecond Citizen:\nOne word, good citizens.\n\nFirst Citiz
en:\nWe are accounted poor citi'
```

For each sequence, duplicate and shift it to form the input and target text by using the `map` method to apply a simple function to each batch:

In [12]:
```
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text

dataset = sequences.map(split_input_target)
```

Print the first examples input and target values:

In [13]:
```
for input_example, target_example in  dataset.take(1):
  print ('Input data: ', repr(''.join(idx2char[input_example.numpy()])))
  print ('Target data:', repr(''.join(idx2char[target_example.numpy()])))
```

```
Input data:  'First Citizen:\nBefore we proceed any further, hear me speak.\
n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou'
Target data: 'irst Citizen:\nBefore we proceed any further, hear me speak.\n\
nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou '
```

Each index of these vectors are processed as one time step. For the input at time step 0, the model receives the index for "F" and trys to predict the index for "i" as the next character. At the next timestep, it does the same thing but the `RNN` considers the previous step context in addition to the current input character.

In [14]:
```
for i, (input_idx, target_idx) in enumerate(zip(input_example[:5], target_ex
ample[:5])):
    print("Step {:4d}".format(i))
    print("  input: {} ({:s})".format(input_idx, repr(idx2char[input_idx])))
    print("  expected output: {} ({:s})".format(target_idx, repr(idx2char[ta
rget_idx])))
```

```
Step    0
  input: 18 ('F')
  expected output: 47 ('i')
Step    1
  input: 47 ('i')
  expected output: 56 ('r')
Step    2
  input: 56 ('r')
  expected output: 57 ('s')
Step    3
  input: 57 ('s')
  expected output: 58 ('t')
Step    4
  input: 58 ('t')
  expected output: 1 (' ')
```

Create training batches

We used `tf.data` to split the text into manageable sequences. But before feeding this data into the model, we need to shuffle the data and pack it into batches.

In [15]:
```
# Batch size
BATCH_SIZE = 64

# Buffer size to shuffle the dataset
# (TF data is designed to work with possibly infinite sequences,
# so it doesn't attempt to shuffle the entire sequence in memory. Instead,
# it maintains a buffer in which it shuffles elements).
BUFFER_SIZE = 10000

dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=Tru
e)

dataset
```

Out[15]:
```
<BatchDataset shapes: ((64, 100), (64, 100)), types: (tf.int32, tf.int32)>
```

Build The Model

Use `tf.keras.Sequential` to define the model. For this simple example three layers are used to define our model:

- `tf.keras.layers.Embedding` : The input layer. A trainable lookup table that will map the numbers of each character to a vector with `embedding_dim` dimensions;
- `tf.keras.layers.GRU` : A type of RNN with size `units=rnn_units` (You can also use a LSTM layer here.)
- `tf.keras.layers.Dense` : The output layer, with `vocab_size` outputs.

In [16]:
```
# Length of the vocabulary in chars
vocab_size = len(vocab)

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024
```

In [17]:
```
def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
  model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              batch_input_shape=[batch_size, None]),
    tf.keras.layers.GRU(rnn_units,
                        return_sequences=True,
                        stateful=True,
                        recurrent_initializer='glorot_uniform'),
    tf.keras.layers.Dense(vocab_size)
  ])
  return model
```

In [18]:
```
model = build_model(
  vocab_size = len(vocab),
  embedding_dim=embedding_dim,
  rnn_units=rnn_units,
  batch_size=BATCH_SIZE)
```

For each character the model looks up the embedding, runs the GRU one timestep with the embedding as input, and applies the dense layer to generate logits predicting the log-likelihood of the next character:

A drawing of the data passing through the model

Please note that we choose to Keras sequential model here since all the layers in the model only have single input and produce single output. In case you want to retrieve and reuse the states from stateful RNN layer, you might want to build your model with Keras functional API or model subclassing. Please check [Keras RNN guide (https://www.tensorflow.org/guide/keras/rnn#rnn_state_reuse)](https://www.tensorflow.org/guide/keras/rnn#rnn_state_reuse) for more details.

Try the model

Now run the model to see that it behaves as expected.

First check the shape of the output:

In [19]:
```
for input_example_batch, target_example_batch in dataset.take(1):
  example_batch_predictions = model(input_example_batch)
  print(example_batch_predictions.shape, "# (batch_size, sequence_length, vocab_size)")
```

```
(64, 100, 65) # (batch_size, sequence_length, vocab_size)
```

In the above example the sequence length of the input is `100` but the model can be run on inputs of any length:

In [20]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (64, None, 256)           16640
_________________________________________________________________
gru (GRU)                    (64, None, 1024)          3938304
_________________________________________________________________
dense (Dense)                (64, None, 65)            66625
=================================================================
Total params: 4,021,569
Trainable params: 4,021,569
Non-trainable params: 0
_________________________________________________________________
```

To get actual predictions from the model we need to sample from the output distribution, to get actual character indices. This distribution is defined by the logits over the character vocabulary.

Note: It is important to *sample* from this distribution as taking the *argmax* of the distribution can easily get the model stuck in a loop.

Try it for the first example in the batch:

In [21]:
```
sampled_indices = tf.random.categorical(example_batch_predictions[0], num_samples=1)
sampled_indices = tf.squeeze(sampled_indices,axis=-1).numpy()
```

This gives us, at each timestep, a prediction of the next character index:

In [22]:
```
sampled_indices
```

Out[22]:
```
array([59, 44, 39, 62, 14, 17, 18, 58, 14,  4, 63, 35, 33, 46, 23, 63,  0,
       35, 44, 37, 42, 52, 35, 11, 64, 34, 54,  6,  5, 47, 58, 53, 12, 18,
       38, 33,  5, 45,  4, 12, 38, 17, 54, 64, 34, 52, 48, 59, 14, 51,  6,
       14, 20,  5, 54, 17, 23, 49,  1, 44, 63, 43, 15, 55, 39, 15, 61, 55,
       39, 14, 42, 49,  9, 59,  7, 58, 14, 21, 61, 62, 21,  7, 16, 38, 22,
       60, 46, 54, 18, 16, 14, 55, 30,  3, 45, 45,  6, 58, 25, 33],
      dtype=int64)
```

Decode these to see the text predicted by this untrained model:

In [23]:
```
print("Input: \n", repr("".join(idx2char[input_example_batch[0]])))
print()
print("Next Char Predictions: \n", repr("".join(idx2char[sampled_indices
])))
```

```
Input:
 'in your lips,\nLike man new made.\n\nANGELO:\nBe you content, fair maid;\nI
t is the law, not I condemn yo'

Next Char Predictions:
 "ufaxBEFtB&yWUhKy\nWfYdnW;zVp,'ito?FZU'g&?ZEpzVnjuBm,BH'pEKk fyeCqaCwqaBdk3u
-tBIwxI-DZJvhpFDBqR$gg,tMU"
```

Train the model

At this point the problem can be treated as a standard classification problem. Given the previous RNN state, and the input this time step, predict the class of the next character.

Attach an optimizer, and a loss function

The standard `tf.keras.losses.sparse_categorical_crossentropy` loss function works in this case because it is applied across the last dimension of the predictions.

Because our model returns logits, we need to set the `from_logits` flag.

In [24]:
```
def loss(labels, logits):
  return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, fro
m_logits=True)

example_batch_loss  = loss(target_example_batch, example_batch_predictions)
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_siz
e, sequence_length, vocab_size)")
print("scalar_loss:      ", example_batch_loss.numpy().mean())
```

```
Prediction shape:  (64, 100, 65)  # (batch_size, sequence_length, vocab_size)
scalar_loss:       4.173717
```

Configure the training procedure using the `tf.keras.Model.compile` method. We'll use `tf.keras.optimizers.Adam` with default arguments and the loss function.

In [25]:
```
model.compile(optimizer='adam', loss=loss)
```

Configure checkpoints

Use a `tf.keras.callbacks.ModelCheckpoint` to ensure that checkpoints are saved during training:

In [26]:
```
# Directory where the checkpoints will be saved
checkpoint_dir = './training_checkpoints'
# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

checkpoint_callback=tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_prefix,
    save_weights_only=True)
```

Execute the training

To keep training time reasonable, use 10 epochs to train the model. In Colab, set the runtime to GPU for faster training.

In [27]:
```
EPOCHS=10
```

In [28]:
```
history = model.fit(dataset, epochs=EPOCHS, callbacks=[checkpoint_callback])
```

```
Train for 172 steps
Epoch 1/10
172/172 [==============================] - 483s 3s/step - loss: 2.6622
Epoch 2/10
172/172 [==============================] - 601s 3s/step - loss: 1.9442
Epoch 3/10
172/172 [==============================] - 617s 4s/step - loss: 1.6804
Epoch 4/10
172/172 [==============================] - 509s 3s/step - loss: 1.5355
Epoch 5/10
172/172 [==============================] - 526s 3s/step - loss: 1.4493
Epoch 6/10
172/172 [==============================] - 588s 3s/step - loss: 1.3900
Epoch 7/10
172/172 [==============================] - 606s 4s/step - loss: 1.3450
Epoch 8/10
172/172 [==============================] - 593s 3s/step - loss: 1.3062
Epoch 9/10
172/172 [==============================] - 576s 3s/step - loss: 1.2710
Epoch 10/10
172/172 [==============================] - 549s 3s/step - loss: 1.2380
```

Generate text

Restore the latest checkpoint

To keep this prediction step simple, use a batch size of 1.

Because of the way the RNN state is passed from timestep to timestep, the model only accepts a fixed batch size once built.

To run the model with a different `batch_size` , we need to rebuild the model and restore the weights from the checkpoint.

In [29]:
```
tf.train.latest_checkpoint(checkpoint_dir)
```
Out[29]:
```
'./training_checkpoints\\ckpt_10'
```

In [30]:
```
model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)

model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))

model.build(tf.TensorShape([1, None]))
```

In [31]:
```
model.summary()
```
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (1, None, 256)            16640     
_________________________________________________________________
gru_1 (GRU)                  (1, None, 1024)           3938304   
_________________________________________________________________
dense_1 (Dense)              (1, None, 65)             66625     
=================================================================
Total params: 4,021,569
Trainable params: 4,021,569
Non-trainable params: 0
_________________________________________________________________
```

The prediction loop

The following code block generates the text:

- It Starts by choosing a start string, initializing the RNN state and setting the number of characters to generate.
- Get the prediction distribution of the next character using the start string and the RNN state.
- Then, use a categorical distribution to calculate the index of the predicted character. Use this predicted character as our next input to the model.
- The RNN state returned by the model is fed back into the model so that it now has more context, instead than only one character. After predicting the next character, the modified RNN states are again fed back into the model, which is how it learns as it gets more context from the previously predicted characters.

To generate text the model's output is fed back to the input

Looking at the generated text, you'll see the model knows when to capitalize, make paragraphs and imitates a Shakespeare-like writing vocabulary. With the small number of training epochs, it has not yet learned to form coherent sentences.

In [32]:
```
def generate_text(model, start_string):
  # Evaluation step (generating text using the learned model)

  # Number of characters to generate
  num_generate = 1000

  # Converting our start string to numbers (vectorizing)
  input_eval = [char2idx[s] for s in start_string]
  input_eval = tf.expand_dims(input_eval, 0)

  # Empty string to store our results
  text_generated = []

  # Low temperatures results in more predictable text.
  # Higher temperatures results in more surprising text.
  # Experiment to find the best setting.
  temperature = 1.0

  # Here batch size == 1
  model.reset_states()
  for i in range(num_generate):
      predictions = model(input_eval)
      # remove the batch dimension
      predictions = tf.squeeze(predictions, 0)

      # using a categorical distribution to predict the character returned b
y the model
      predictions = predictions / temperature
      predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,
0].numpy()

      # We pass the predicted character as the next input to the model
      # along with the previous hidden state
      input_eval = tf.expand_dims([predicted_id], 0)

      text_generated.append(idx2char[predicted_id])

  return (start_string + ''.join(text_generated))
```

In [33]:
```
print(generate_text(model, start_string=u"ROMEO: "))
```

```
ROMEO: I send upon you.--
Of Jovel upon your pleasure; I pray thee, Friar, be resign'd; so can never li
neame to kills:
You have no creeble still, judge eight shappy grace; I'll bid I hear,
Upon the own mourning will set thee, and
Am all a faurt,
Thou art genet and noble in, whom they musterman to
brant ye were, thou noblest wisdom stream
As present secrecish wages,
And pity my son a-broke; but much minechere is your knowledge habedue must b
e.

MERCUTIO:
Thou dost thou, 'Awas pleaseds Edward stands of wimes and all:
Then thou wilt anvised, knock me with mine place,
I'll prove a night.
USETRASHORK:
And if I te? call the supple dependers were affection.
This is his horse and keeprots make his wime.

CATRSCHARD III:
This sunger wife's sake.

LORD ROSS:
Patience;
I servey have if you did give nothing;
Or let me so boldly.

First Murderer:
When we may never sat this earth.

KING HENRY VI:
So friar at Saint call'd my heart access.
Therefore, insoly, my lords, and did one
to all full of any court: I
```

The easiest thing you can do to improve the results it to train it for longer (try `EPOCHS=30`).

You can also experiment with a different start string, or try adding another RNN layer to improve the model's accuracy, or adjusting the temperature parameter to generate more or less random predictions.

In [ ]:
```

```

In [1]:
```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Time series forecasting

[View on TensorFlow.org (https://www.tensorflow.org/tutorials/structured_data/time_series)] [Run in Google Colab (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/structured_data/time_series.ipynb)] [View source on GitHub (https://github.com/tensorflow/docs/blob/master/site/en/tutorials/structured_data/time_series.ipynb)] [Download notebook (https://storage.googleapis.com/tensorflow_docs/docs/site/en/tutorials/structured_data/time_series.ipynb)]

This tutorial is an introduction to time series forecasting using Recurrent Neural Networks (RNNs). This is covered in two parts: first, you will forecast a univariate time series, then you will forecast a multivariate time series.

In [2]:
```
import tensorflow as tf

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd

mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['axes.grid'] = False
```

The weather dataset

This tutorial uses a [weather time series dataset (https://www.bgc-jena.mpg.de/wetter/)] recorded by the Max Planck Institute for Biogeochemistry (https://www.bgc-jena.mpg.de).

This dataset contains 14 different features such as air temperature, atmospheric pressure, and humidity. These were collected every 10 minutes, beginning in 2003. For efficiency, you will use only the data collected between 2009 and 2016. This section of the dataset was prepared by François Chollet for his book Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python).

In [3]:
```
zip_path = tf.keras.utils.get_file(
    origin='https://storage.googleapis.com/tensorflow/tf-keras-datasets/jena_climate_2009_2016.csv.zip',
    fname='jena_climate_2009_2016.csv.zip',
    extract=True)
csv_path, _ = os.path.splitext(zip_path)
```

In [4]:
```
df = pd.read_csv(csv_path)
```

Let's take a glance at the data.

In [5]:
```
df.head()
```

Out[5]:

	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)
0	01.01.2009 00:10:00	996.52	-8.02	265.40	-8.90	93.3	3.33	3.11	0.22	1.94	3.12	1307.75
1	01.01.2009 00:20:00	996.57	-8.41	265.01	-9.28	93.4	3.23	3.02	0.21	1.89	3.03	1309.80
2	01.01.2009 00:30:00	996.53	-8.51	264.91	-9.31	93.9	3.21	3.01	0.20	1.88	3.02	1310.24
3	01.01.2009 00:40:00	996.51	-8.31	265.12	-9.07	94.2	3.26	3.07	0.19	1.92	3.08	1309.19
4	01.01.2009 00:50:00	996.51	-8.27	265.15	-9.04	94.1	3.27	3.08	0.19	1.92	3.09	1309.00

As you can see above, an observation is recorded every 10 minutes. This means that, for a single hour, you will have 6 observations. Similarly, a single day will contain 144 (6x24) observations.

Given a specific time, let's say you want to predict the temperature 6 hours in the future. In order to make this prediction, you choose to use 5 days of observations. Thus, you would create a window containing the last 720(5x144) observations to train the model. Many such configurations are possible, making this dataset a good one to experiment with.

The function below returns the above described windows of time for the model to train on. The parameter `history_size` is the size of the past window of information. The `target_size` is how far in the future does the model need to learn to predict. The `target_size` is the label that needs to be predicted.

In [6]:
```
def univariate_data(dataset, start_index, end_index, history_size, target_size):
  data = []
  labels = []

  start_index = start_index + history_size
  if end_index is None:
    end_index = len(dataset) - target_size

  for i in range(start_index, end_index):
    indices = range(i-history_size, i)
    # Reshape data from (history_size,) to (history_size, 1)
    data.append(np.reshape(dataset[indices], (history_size, 1)))
    labels.append(dataset[i+target_size])
  return np.array(data), np.array(labels)
```

In both the following tutorials, the first 300,000 rows of the data will be the training dataset, and there remaining will be the validation dataset. This amounts to ~2100 days worth of training data.

In [7]:
```
TRAIN_SPLIT = 300000
```

Setting seed to ensure reproducibility.

In [8]:
```
tf.random.set_seed(13)
```

Part 1: Forecast a univariate time series

First, you will train a model using only a single feature (temperature), and use it to make predictions for that value in the future.

Let's first extract only the temperature from the dataset.

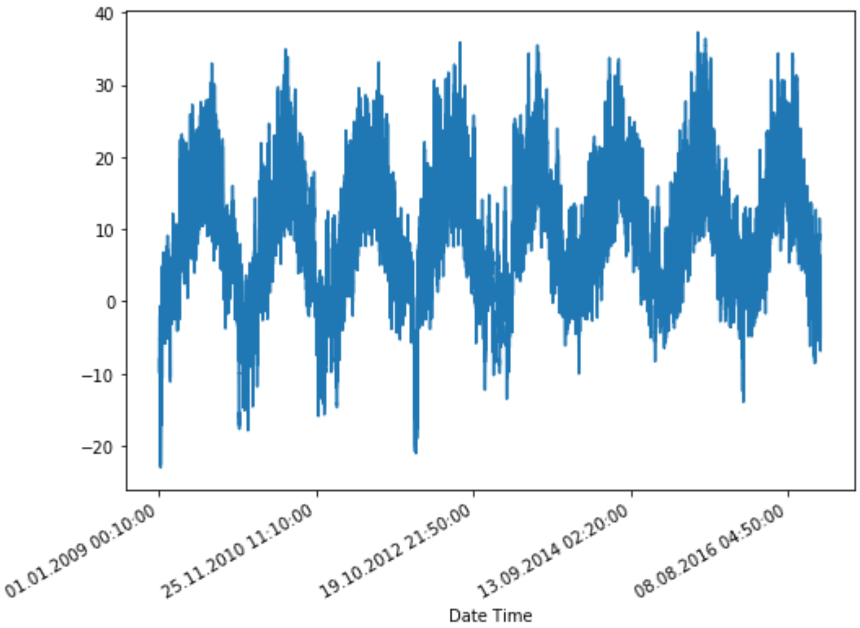
In [9]:
```
uni_data = df['T (degC)']
uni_data.index = df['Date Time']
uni_data.head()
```

Out[9]:
```
Date Time
01.01.2009 00:10:00   -8.02
01.01.2009 00:20:00   -8.41
01.01.2009 00:30:00   -8.51
01.01.2009 00:40:00   -8.31
01.01.2009 00:50:00   -8.27
Name: T (degC), dtype: float64
```

Let's observe how this data looks across time.

In [10]:
```
uni_data.plot(subplots=True)
```

Out[10]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x000001F30942BF48>],
      dtype=object)
```



In [11]:
```
uni_data = uni_data.values
```

It is important to scale features before training a neural network. Standardization is a common way of doing this scaling by subtracting the mean and dividing by the standard deviation of each feature.You could also use a `tf.keras.utils.normalize` method that rescales the values into a range of [0,1].

Note: The mean and standard deviation should only be computed using the training data.

In [12]:
```
uni_train_mean = uni_data[:TRAIN_SPLIT].mean()
uni_train_std = uni_data[:TRAIN_SPLIT].std()
```

Let's standardize the data.

In [13]:
```
uni_data = (uni_data-uni_train_mean)/uni_train_std
```

Let's now create the data for the univariate model. For part 1, the model will be given the last 20 recorded temperature observations, and needs to learn to predict the temperature at the next time step.

In [14]:
```
univariate_past_history = 20
univariate_future_target = 0

x_train_uni, y_train_uni = univariate_data(uni_data, 0, TRAIN_SPLIT,
                                           univariate_past_history,
                                           univariate_future_target)
x_val_uni, y_val_uni = univariate_data(uni_data, TRAIN_SPLIT, None,
                                       univariate_past_history,
                                       univariate_future_target)
```

This is what the `univariate_data` function returns.

In [15]:
```
print ('Single window of past history')
print (x_train_uni[0])
print ('\n Target temperature to predict')
print (y_train_uni[0])
```

```
Single window of past history
[[-1.99766294]
 [-2.04281897]
 [-2.05439744]
 [-2.0312405 ]
 [-2.02660912]
 [-2.00113649]
 [-1.95134907]
 [-1.95134907]
 [-1.98492663]
 [-2.04513467]
 [-2.08334362]
 [-2.09723778]
 [-2.09376424]
 [-2.09144854]
 [-2.07176515]
 [-2.07176515]
 [-2.07639653]
 [-2.08913285]
 [-2.09260639]
 [-2.10418486]]

 Target temperature to predict
-2.1041848598100876
```

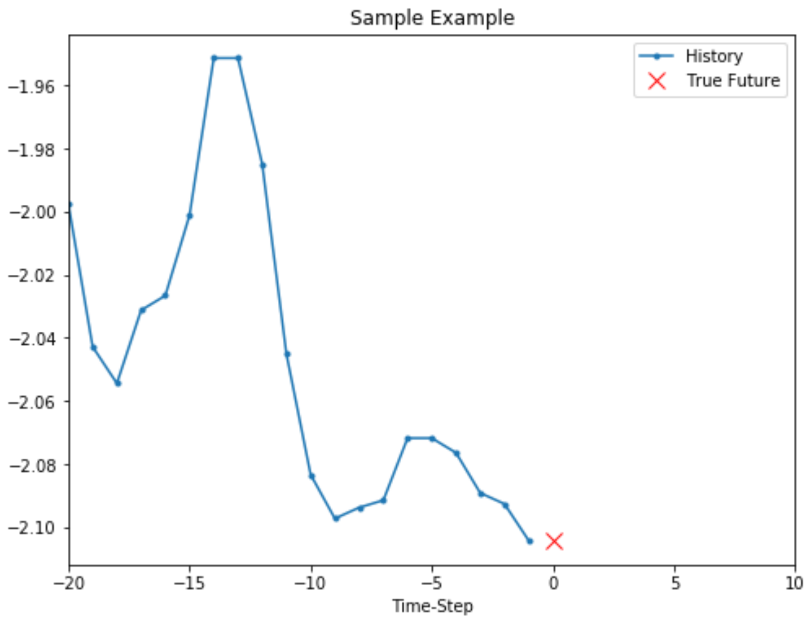
Now that the data has been created, let's take a look at a single example. The information given to the network is given in blue, and it must predict the value at the red cross.

In [16]:
```
def create_time_steps(length):
  return list(range(-length, 0))
```

In [17]:
```
def show_plot(plot_data, delta, title):
  labels = ['History', 'True Future', 'Model Prediction']
  marker = ['.-', 'rx', 'go']
  time_steps = create_time_steps(plot_data[0].shape[0])
  if delta:
    future = delta
  else:
    future = 0

  plt.title(title)
  for i, x in enumerate(plot_data):
    if i:
      plt.plot(future, plot_data[i], marker[i], markersize=10,
               label=labels[i])
    else:
      plt.plot(time_steps, plot_data[i].flatten(), marker[i], label=labels
[i])
  plt.legend()
  plt.xlim([time_steps[0], (future+5)*2])
  plt.xlabel('Time-Step')
  return plt
```

In [18]:
```
show_plot([x_train_uni[0], y_train_uni[0]], 0, 'Sample Example')
```

Out[18]:
```
<module 'matplotlib.pyplot' from 'C:\\Users\\gcont\\Anaconda3\\envs\\tf\\li
b\\site-packages\\matplotlib\\pyplot.py'>
```



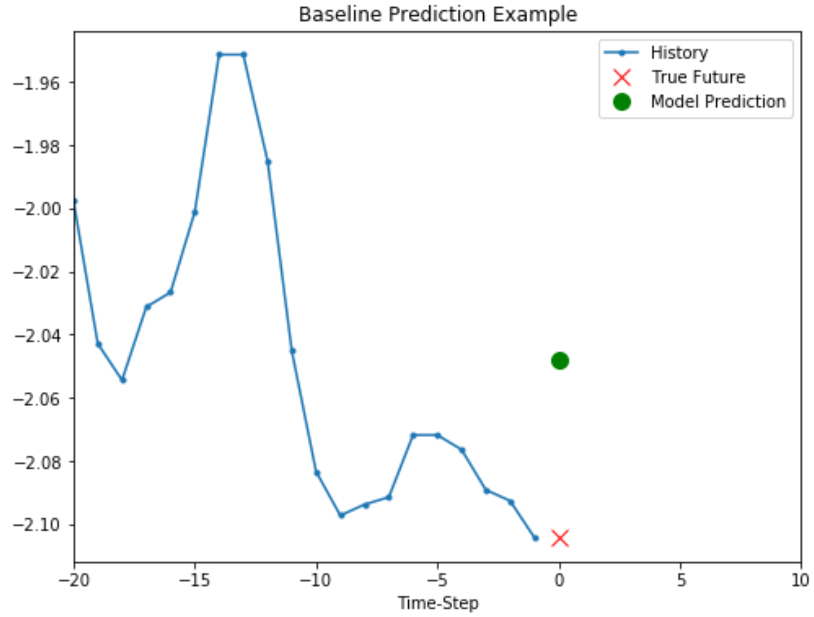
Baseline

Before proceeding to train a model, let's first set a simple baseline. Given an input point, the baseline method looks at all the history and predicts the next point to be the average of the last 20 observations.

In [19]:
```
def baseline(history):
  return np.mean(history)
```

In [20]:
```
show_plot([x_train_uni[0], y_train_uni[0], baseline(x_train_uni[0])], 0,
           'Baseline Prediction Example')
```

Out[20]:
```
<module 'matplotlib.pyplot' from 'C:\\Users\\gcont\\Anaconda3\\envs\\tf\\li
b\\site-packages\\matplotlib\\pyplot.py'>
```



Let's see if you can beat this baseline using a recurrent neural network.

Recurrent neural network

A Recurrent Neural Network (RNN) is a type of neural network well-suited to time series data. RNNs process a time series step-by-step, maintaining an internal state summarizing the information they've seen so far. For more details, read the [RNN tutorial (https://www.tensorflow.org/tutorials/sequences/recurrent)](https://www.tensorflow.org/tutorials/sequences/recurrent). In this tutorial, you will use a specialized RNN layer called Long Short Term Memory ([LSTM (https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/LSTM)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/LSTM))

Let's now use `tf.data` to shuffle, batch, and cache the dataset.

In [21]:
```
BATCH_SIZE = 256
BUFFER_SIZE = 10000

train_univariate = tf.data.Dataset.from_tensor_slices((x_train_uni, y_train_
uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_SIZE).batch(BATCH
_SIZE).repeat()

val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni, y_val_uni))
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

The following visualisation should help you understand how the data is represented after batching.

Time Series

You will see the LSTM requires the input shape of the data it is being given.

In [22]:
```
simple_lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(8, input_shape=x_train_uni.shape[-2:]),
    tf.keras.layers.Dense(1)
])

simple_lstm_model.compile(optimizer='adam', loss='mae')
```

Let's make a sample prediction, to check the output of the model.

In [23]:
```
for x, y in val_univariate.take(1):
    print(simple_lstm_model.predict(x).shape)
```

```
(256, 1)
```

Let's train the model now. Due to the large size of the dataset, in the interest of saving time, each epoch will only run for 200 steps, instead of the complete training data as normally done.

In [24]:
```
EVALUATION_INTERVAL = 200
EPOCHS = 10

simple_lstm_model.fit(train_univariate, epochs=EPOCHS,
                      steps_per_epoch=EVALUATION_INTERVAL,
                      validation_data=val_univariate, validation_steps=50)
```

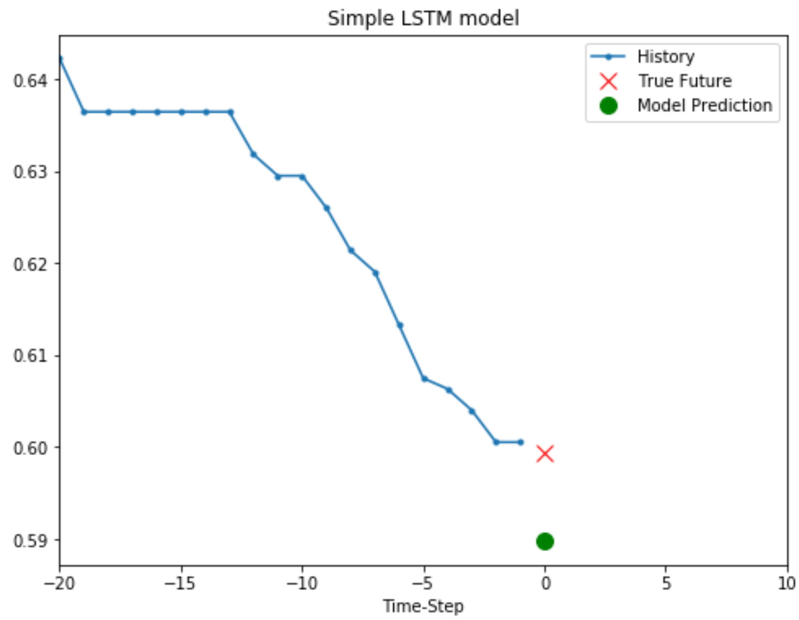
```
Train for 200 steps, validate for 50 steps
Epoch 1/10
200/200 [==============================] - 4s 20ms/step - loss: 0.4075 - val_
loss: 0.1351
Epoch 2/10
200/200 [==============================] - 2s 9ms/step - loss: 0.1118 - val_l
oss: 0.0359
Epoch 3/10
200/200 [==============================] - 2s 9ms/step - loss: 0.0489 - val_l
oss: 0.0290
Epoch 4/10
200/200 [==============================] - 2s 9ms/step - loss: 0.0443 - val_l
oss: 0.0258
Epoch 5/10
200/200 [==============================] - 2s 9ms/step - loss: 0.0299 - val_l
oss: 0.0235
Epoch 6/10
200/200 [==============================] - 2s 9ms/step - loss: 0.0317 - val_l
oss: 0.0224
Epoch 7/10
200/200 [==============================] - 2s 9ms/step - loss: 0.0286 - val_l
oss: 0.0206
Epoch 8/10
200/200 [==============================] - 2s 10ms/step - loss: 0.0263 - val_
loss: 0.0197
Epoch 9/10
200/200 [==============================] - 2s 10ms/step - loss: 0.0253 - val_
loss: 0.0182
Epoch 10/10
200/200 [==============================] - 2s 10ms/step - loss: 0.0227 - val_
loss: 0.0174
```

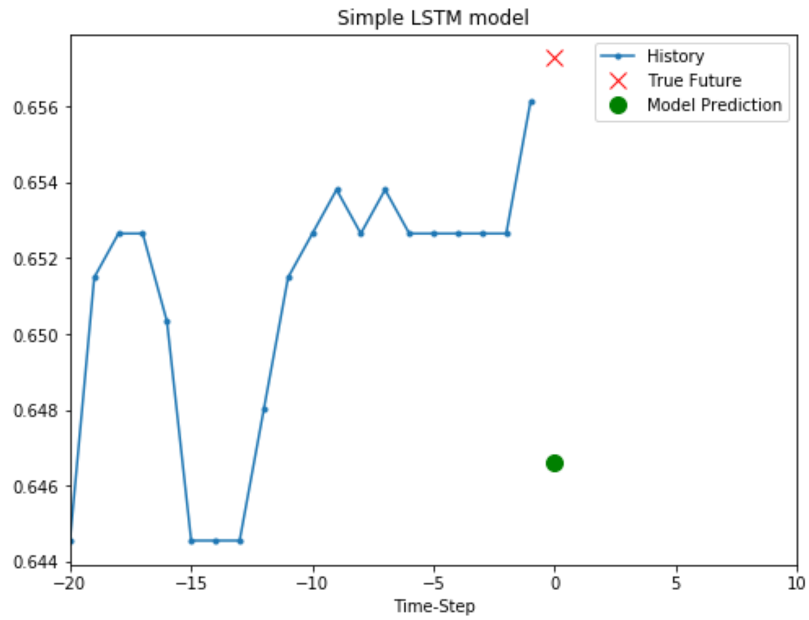
Out[24]:
```
<tensorflow.python.keras.callbacks.History at 0x1f30b08c788>
```

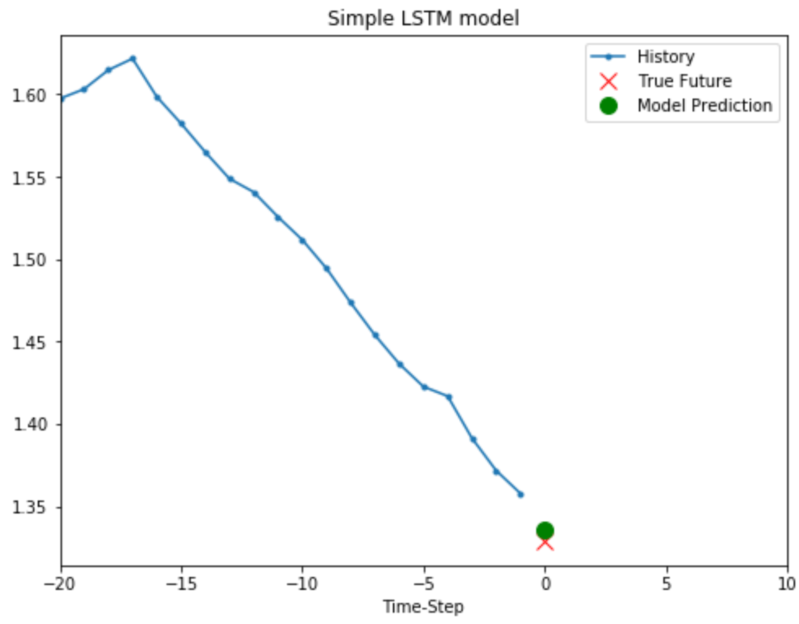
Predict using the simple LSTM model

Now that you have trained your simple LSTM, let's try and make a few predictions.

In [25]:
```
for x, y in val_univariate.take(3):
  plot = show_plot([x[0].numpy(), y[0].numpy(),
                    simple_lstm_model.predict(x)[0]], 0, 'Simple LSTM model
')
  plot.show()
```


Simple LSTM model
History
True Future
Model Prediction
0.64
0.63
0.62
0.61
0.60
0.59
−20
−15
−10
−5
0
5
10
Time-Step


Simple LSTM model
History
True Future
Model Prediction
0.656
0.654
0.652
0.650
0.648
0.646
0.644
−20
−15
−10
−5
0
5
10
Time-Step



This looks better than the baseline. Now that you have seen the basics, let's move on to part two, where you will work with a multivariate time series.

Part 2: Forecast a multivariate time series

The original dataset contains fourteen features. For simplicity, this section considers only three of the original fourteen. The features used are air temperature, atmospheric pressure, and air density.

To use more features, add their names to this list.

In [26]:
```
features_considered = ['p (mbar)', 'T (degC)', 'rho (g/m**3)']
```

In [27]:
```
features = df[features_considered]
features.index = df['Date Time']
features.head()
```

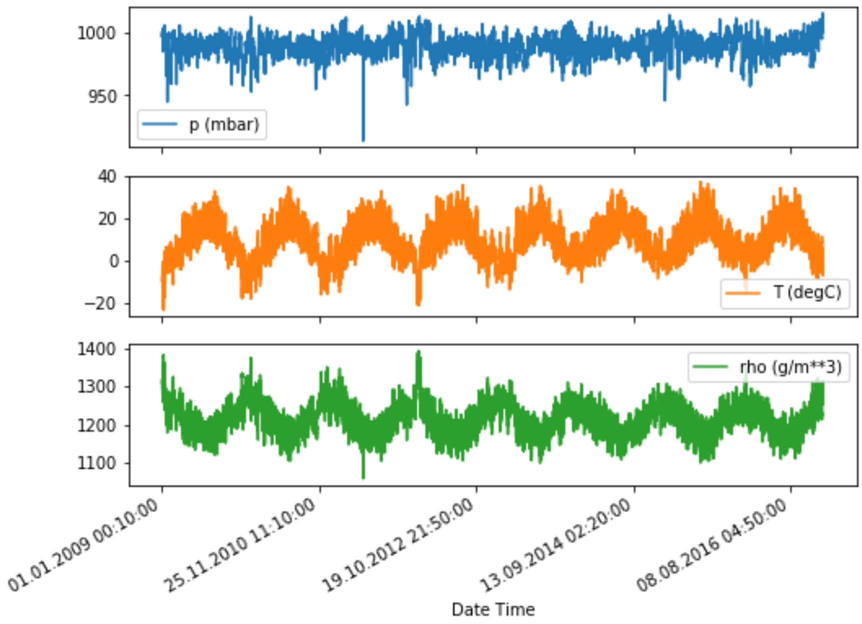
Out[27]:

Date Time	p (mbar)	T (degC)	rho (g/m**3)
01.01.2009 00:10:00	996.52	-8.02	1307.75
01.01.2009 00:20:00	996.57	-8.41	1309.80
01.01.2009 00:30:00	996.53	-8.51	1310.24
01.01.2009 00:40:00	996.51	-8.31	1309.19
01.01.2009 00:50:00	996.51	-8.27	1309.00

Let's have a look at how each of these features vary across time.

In [28]:
```
features.plot(subplots=True)
```

Out[28]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x000001F313B7B508>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x000001F313B5C5C8>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x000001F313B7B408>],
      dtype=object)
```



As mentioned, the first step will be to standardize the dataset using the mean and standard deviation of the training data.

In [29]:
```
dataset = features.values
data_mean = dataset[:TRAIN_SPLIT].mean(axis=0)
data_std = dataset[:TRAIN_SPLIT].std(axis=0)
```

In [30]:
```
dataset = (dataset-data_mean)/data_std
```

Single step model

In a single step setup, the model learns to predict a single point in the future based on some history provided.

The below function performs the same windowing task as above, however, here it samples the past observation based on the step size given.

In [31]:
```
def multivariate_data(dataset, target, start_index, end_index, history_size,
                      target_size, step, single_step=False):
  data = []
  labels = []

  start_index = start_index + history_size
  if end_index is None:
    end_index = len(dataset) - target_size

  for i in range(start_index, end_index):
    indices = range(i-history_size, i, step)
    data.append(dataset[indices])

    if single_step:
      labels.append(target[i+target_size])
    else:
      labels.append(target[i:i+target_size])

  return np.array(data), np.array(labels)
```

In this tutorial, the network is shown data from the last five (5) days, i.e. 720 observations that are sampled every hour. The sampling is done every one hour since a drastic change is not expected within 60 minutes. Thus, 120 observation represent history of the last five days. For the single step prediction model, the label for a datapoint is the temperature 12 hours into the future. In order to create a label for this, the temperature after 72(12*6) observations is used.

In [32]:
```
past_history = 720
future_target = 72
STEP = 6

x_train_single, y_train_single = multivariate_data(dataset, dataset[:, 1],
0,
                                                   TRAIN_SPLIT, past_histor
y,
                                                   future_target, STEP,
                                                   single_step=True)
x_val_single, y_val_single = multivariate_data(dataset, dataset[:, 1],
                                               TRAIN_SPLIT, None, past_histo
ry,
                                               future_target, STEP,
                                               single_step=True)
```

Let's look at a single data-point.

In [33]:
```
print ('Single window of past history : {}'.format(x_train_single[0].shape))
```

```
Single window of past history : (120, 3)
```

In [34]:
```
train_data_single = tf.data.Dataset.from_tensor_slices((x_train_single, y_tr
ain_single))
train_data_single = train_data_single.cache().shuffle(BUFFER_SIZE).batch(BAT
CH_SIZE).repeat()

val_data_single = tf.data.Dataset.from_tensor_slices((x_val_single, y_val_si
ngle))
val_data_single = val_data_single.batch(BATCH_SIZE).repeat()
```

In [35]:
```
single_step_model = tf.keras.models.Sequential()
single_step_model.add(tf.keras.layers.LSTM(32,
                                           input_shape=x_train_single.shape
[-2:]))
single_step_model.add(tf.keras.layers.Dense(1))

single_step_model.compile(optimizer=tf.keras.optimizers.RMSprop(), loss='mae
')
```

Let's check out a sample prediction.

In [36]:
```
for x, y in val_data_single.take(1):
  print(single_step_model.predict(x).shape)
```

```
(256, 1)
```

In [37]:
```
single_step_history = single_step_model.fit(train_data_single, epochs=EPOCH
S,
                                            steps_per_epoch=EVALUATION_INTER
VAL,
                                            validation_data=val_data_single,
                                            validation_steps=50)
```

```
Train for 200 steps, validate for 50 steps
Epoch 1/10
200/200 [==============================] - 34s 169ms/step - loss: 0.3090 - va
l_loss: 0.2647
Epoch 2/10
200/200 [==============================] - 38s 192ms/step - loss: 0.2623 - va
l_loss: 0.2429
Epoch 3/10
200/200 [==============================] - 48s 242ms/step - loss: 0.2614 - va
l_loss: 0.2474
Epoch 4/10
200/200 [==============================] - 1031s 5s/step - loss: 0.2569 - val
_loss: 0.2448
Epoch 5/10
200/200 [==============================] - 797s 4s/step - loss: 0.2267 - val_
loss: 0.2344
Epoch 6/10
200/200 [==============================] - 48s 239ms/step - loss: 0.2415 - va
l_loss: 0.2668
Epoch 7/10
200/200 [==============================] - 48s 239ms/step - loss: 0.2416 - va
l_loss: 0.2566
Epoch 8/10
200/200 [==============================] - 48s 238ms/step - loss: 0.2410 - va
l_loss: 0.2387
Epoch 9/10
200/200 [==============================] - 47s 236ms/step - loss: 0.2452 - va
l_loss: 0.2478
Epoch 10/10
200/200 [==============================] - 54s 272ms/step - loss: 0.2387 - va
l_loss: 0.2425
```

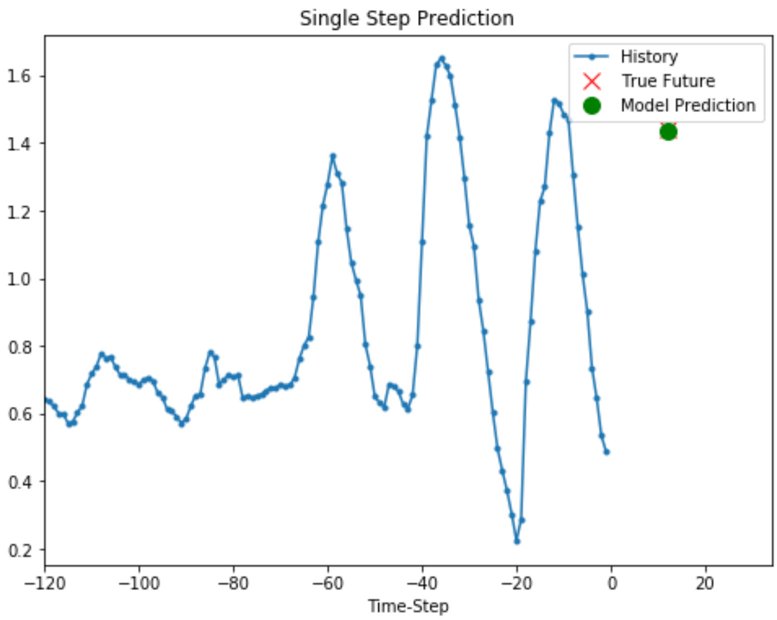
In [38]:
```
def plot_train_history(history, title):
  loss = history.history['loss']
  val_loss = history.history['val_loss']

  epochs = range(len(loss))

  plt.figure()

  plt.plot(epochs, loss, 'b', label='Training loss')
  plt.plot(epochs, val_loss, 'r', label='Validation loss')
  plt.title(title)
  plt.legend()

  plt.show()
```

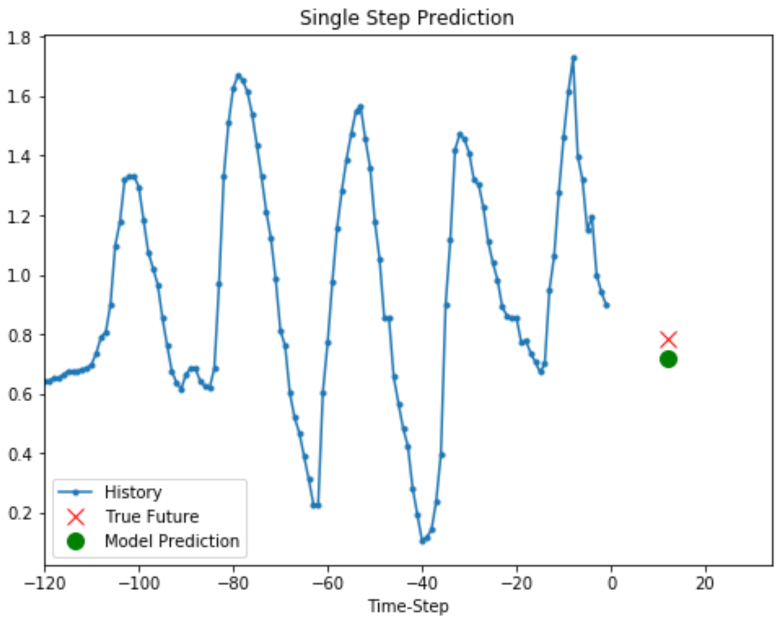
In [39]:
```
plot_train_history(single_step_history,
                   'Single Step Training and validation loss')
```

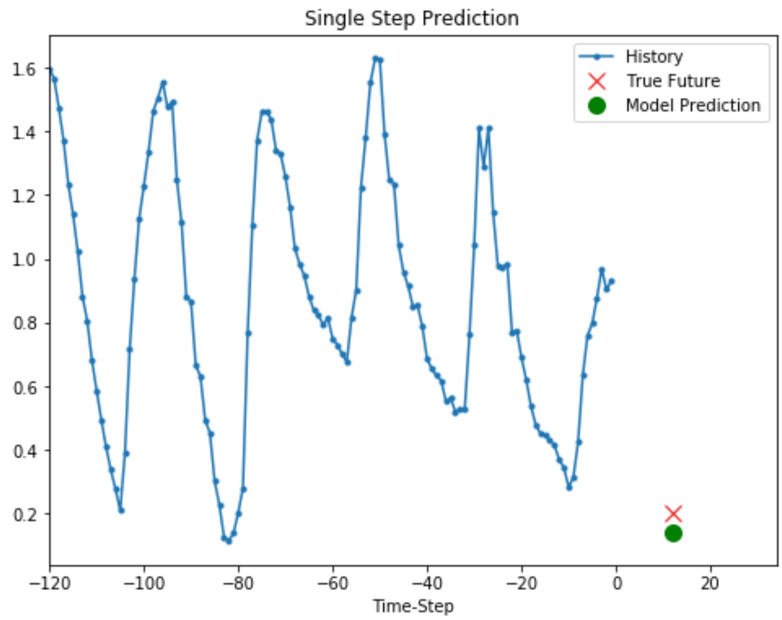
Predict a single step future

Now that the model is trained, let's make a few sample predictions. The model is given the history of three features over the past five days sampled every hour (120 data-points), since the goal is to predict the temperature, the plot only displays the past temperature. The prediction is made one day into the future (hence the gap between the history and prediction).

In [40]:
```
for x, y in val_data_single.take(3):
  plot = show_plot([x[0][:, 1].numpy(), y[0].numpy(),
                    single_step_model.predict(x)[0]], 12,
                   'Single Step Prediction')
  plot.show()
```


Single Step Prediction
History
True Future
Model Prediction
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
−120
−100
−80
−60
−40
−20
0
20
Time-Step


Single Step Prediction
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
History
True Future
Model Prediction
−120
−100
−80
−60
−40
−20
0
20
Time-Step



Multi-Step model

In a multi-step prediction model, given a past history, the model needs to learn to predict a range of future values. Thus, unlike a single step model, where only a single future point is predicted, a multi-step model predict a sequence of the future.

For the multi-step model, the training data again consists of recordings over the past five days sampled every hour. However, here, the model needs to learn to predict the temperature for the next 12 hours. Since an obversation is taken every 10 minutes, the output is 72 predictions. For this task, the dataset needs to be prepared accordingly, thus the first step is just to create it again, but with a different target window.

In [41]:
```
future_target = 72
x_train_multi, y_train_multi = multivariate_data(dataset, dataset[:, 1], 0,
                                                 TRAIN_SPLIT, past_history,
                                                 future_target, STEP)
x_val_multi, y_val_multi = multivariate_data(dataset, dataset[:, 1],
                                             TRAIN_SPLIT, None, past_histor
y,
                                             future_target, STEP)
```

Let's check out a sample data-point.

In [42]:
```
print ('Single window of past history : {}'.format(x_train_multi[0].shape))
print ('\n Target temperature to predict : {}'.format(y_train_multi[0].shap
e))
```

```
Single window of past history : (120, 3)

 Target temperature to predict : (72,)
```

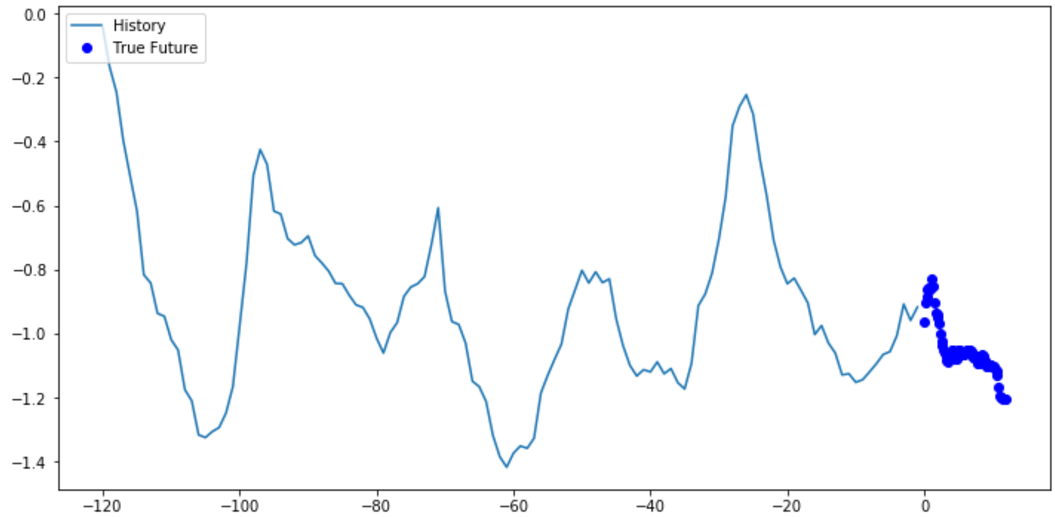
In [43]:
```
train_data_multi = tf.data.Dataset.from_tensor_slices((x_train_multi, y_trai
n_multi))
train_data_multi = train_data_multi.cache().shuffle(BUFFER_SIZE).batch(BATCH
_SIZE).repeat()

val_data_multi = tf.data.Dataset.from_tensor_slices((x_val_multi, y_val_mult
i))
val_data_multi = val_data_multi.batch(BATCH_SIZE).repeat()
```

Plotting a sample data-point.

In [44]:
```
def multi_step_plot(history, true_future, prediction):
  plt.figure(figsize=(12, 6))
  num_in = create_time_steps(len(history))
  num_out = len(true_future)

  plt.plot(num_in, np.array(history[:, 1]), label='History')
  plt.plot(np.arange(num_out)/STEP, np.array(true_future), 'bo',
           label='True Future')
  if prediction.any():
    plt.plot(np.arange(num_out)/STEP, np.array(prediction), 'ro',
             label='Predicted Future')
  plt.legend(loc='upper left')
  plt.show()
```

In this plot and subsequent similar plots, the history and the future data are sampled every hour.

In [45]:
```
for x, y in train_data_multi.take(1):
  multi_step_plot(x[0], y[0], np.array([0]))
```



Since the task here is a bit more complicated than the previous task, the model now consists of two LSTM layers. Finally, since 72 predictions are made, the dense layer outputs 72 predictions.

In [46]:
```
multi_step_model = tf.keras.models.Sequential()
multi_step_model.add(tf.keras.layers.LSTM(32,
                                          return_sequences=True,
                                          input_shape=x_train_multi.shape[-
2:]))
multi_step_model.add(tf.keras.layers.LSTM(16, activation='relu'))
multi_step_model.add(tf.keras.layers.Dense(72))

multi_step_model.compile(optimizer=tf.keras.optimizers.RMSprop(clipvalue=1.
0), loss='mae')
```

Let's see how the model predicts before it trains.

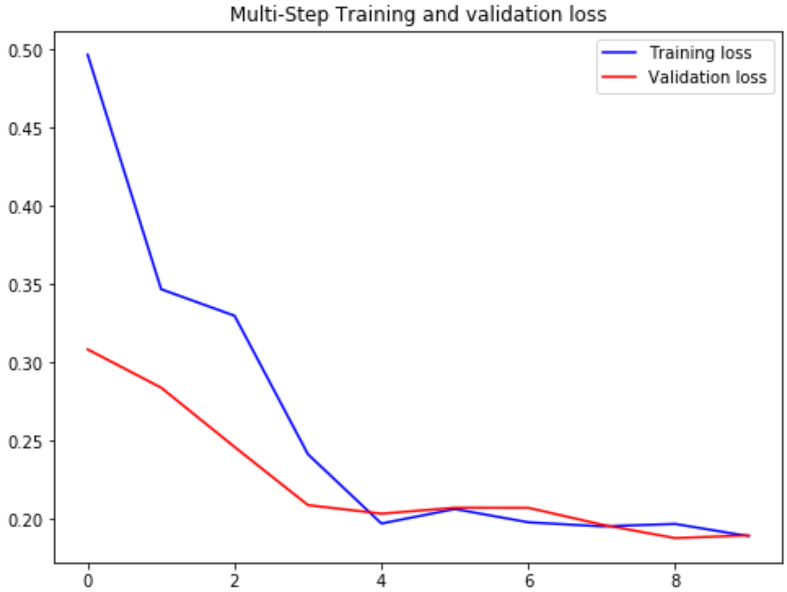
In [47]:
```
for x, y in val_data_multi.take(1):
  print (multi_step_model.predict(x).shape)
```

```
(256, 72)
```

In [48]:
```
multi_step_history = multi_step_model.fit(train_data_multi, epochs=EPOCHS,
                                          steps_per_epoch=EVALUATION_INTERVA
L,
                                          validation_data=val_data_multi,
                                          validation_steps=50)
```

```
Train for 200 steps, validate for 50 steps
Epoch 1/10
200/200 [==============================] - 87s 437ms/step - loss: 0.4969 - va
l_loss: 0.3084
Epoch 2/10
200/200 [==============================] - 108s 538ms/step - loss: 0.3469 - v
al_loss: 0.2840
Epoch 3/10
200/200 [==============================] - 130s 648ms/step - loss: 0.3299 - v
al_loss: 0.2460
Epoch 4/10
200/200 [==============================] - 122s 608ms/step - loss: 0.2413 - v
al_loss: 0.2088
Epoch 5/10
200/200 [==============================] - 133s 665ms/step - loss: 0.1971 - v
al_loss: 0.2034
Epoch 6/10
200/200 [==============================] - 159s 795ms/step - loss: 0.2061 - v
al_loss: 0.2072
Epoch 7/10
200/200 [==============================] - 174s 871ms/step - loss: 0.1978 - v
al_loss: 0.2071
Epoch 8/10
200/200 [==============================] - 177s 887ms/step - loss: 0.1953 - v
al_loss: 0.1964
Epoch 9/10
200/200 [==============================] - 171s 855ms/step - loss: 0.1968 - v
al_loss: 0.1877
Epoch 10/10
200/200 [==============================] - 164s 819ms/step - loss: 0.1890 - v
al_loss: 0.1895
```

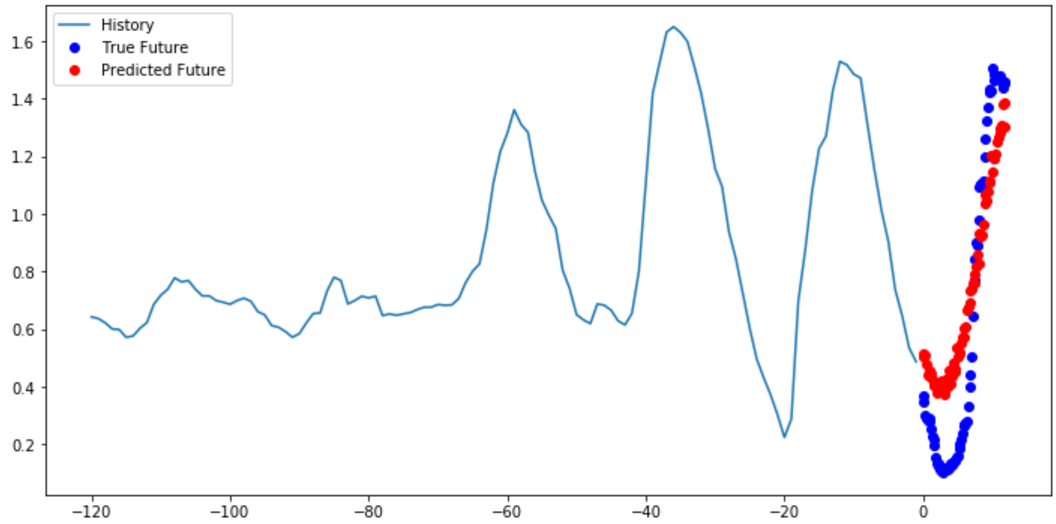
In [49]:
```
plot_train_history(multi_step_history, 'Multi-Step Training and validation loss')
```

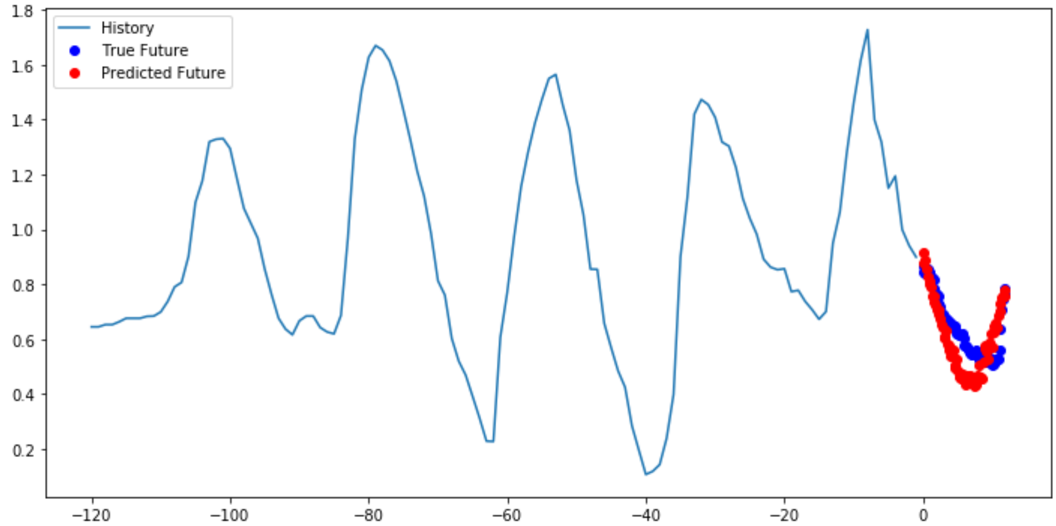


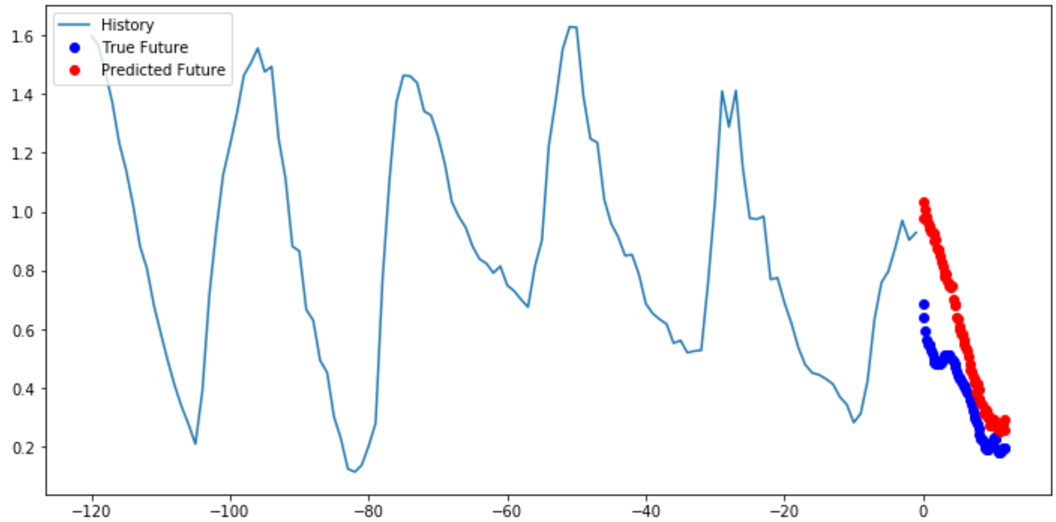
Predict a multi-step future

Let's now have a look at how well your network has learnt to predict the future.

In [50]:
```
for x, y in val_data_multi.take(3):
  multi_step_plot(x[0], y[0], multi_step_model.predict(x)[0])
```







Next steps

This tutorial was a quick introduction to time series forecasting using an RNN. You may now try to predict the stock market and become a billionaire.

In addition, you may also write a generator to yield data (instead of the uni/multivariate_data function), which would be more memory efficient. You may also check out this [time series windowing (https://www.tensorflow.org/guide/data#time_series_windowing)](https://www.tensorflow.org/guide/data#time_series_windowing) guide and use it in this tutorial.

For further understanding, you may read Chapter 15 of [Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow (https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/)](https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/), 2nd Edition and Chapter 6 of [Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python)](https://www.manning.com/books/deep-learning-with-python).

In [ ]: