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)

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)
```

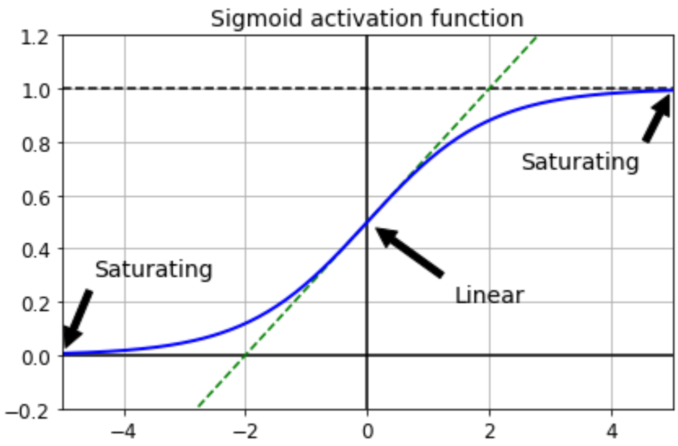
1. Vanishing/Exploding Gradients Problem

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

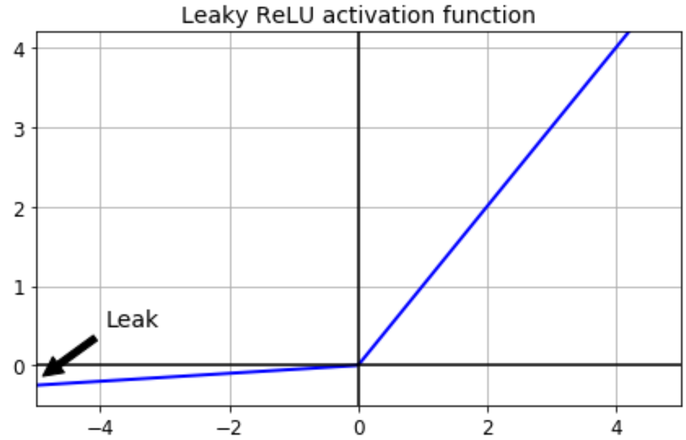
Using non-saturating activation functions helps with the vanishing/exploding gradient problem.

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))
```

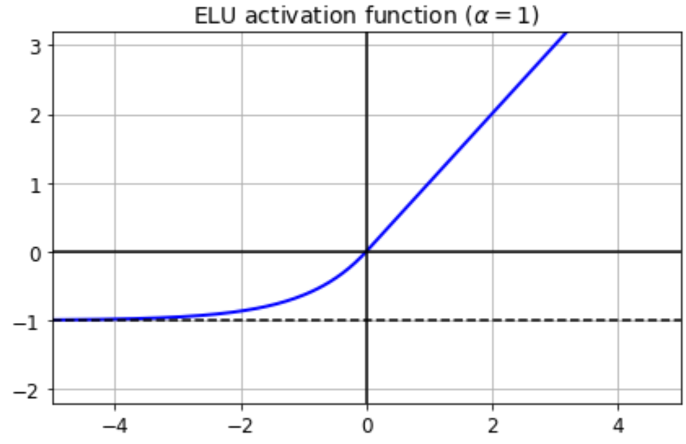
```
Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 3s 63us/sample - loss: 1.3452
- accuracy: 0.6203 - val_loss: 0.9241 - val_accuracy: 0.7170
Epoch 2/10
55000/55000 [==============================] - 3s 60us/sample - loss: 0.8196
- accuracy: 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

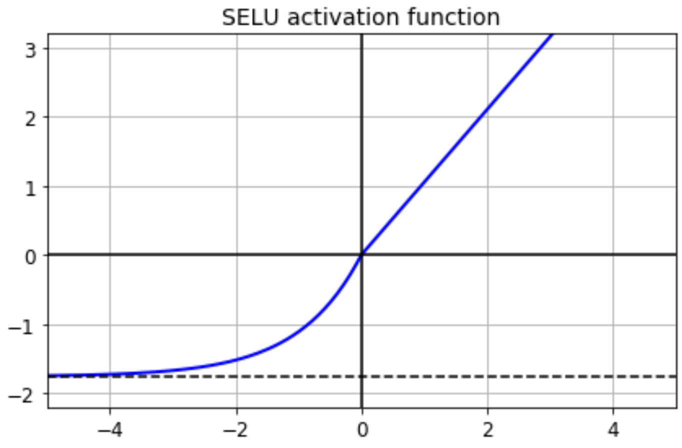
This activation function was proposed in this [great paper (https://arxiv.org/pdf/1706.02515.pdf)](https://arxiv.org/pdf/1706.02515.pdf) by Günter Klambauer, Thomas Unterthiner and Andreas Mayr, published in June 2017. During training, a neural network composed exclusively of a stack of dense layers 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_normal"),
    keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_normal"),
    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, metrics=["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_norma
l"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_norma
l"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
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 [==============================] - 11s 193us/sample - loss: 0.573
0 - 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_nor
mal",
                           kernel_constraint=keras.constraints.max_norm
(1.)),
    keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_nor
mal",
                           kernel_constraint=keras.constraints.max_norm
(1.)),
    keras.layers.Dense(10, activation="softmax")
])
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 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
```