Lecture 26 Part 1 - Best Practices for Training Artificial Neural Networks

Challenges in Training ANNs

We have introduced standard MLPs, which are generally shallow networks, with just a few layers. What if you need to tackle a very complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper architecture, perhaps with 10 layers or much more, each containing hundreds of neurons, connected by hundreds of thousands of connections. This would not be an easy task.

1. **Vanishing gradient problem**. You would be faced with the tricky vanishing gradients problem (or the related exploding gradients problem) that affects deep neural networks and makes lower layers very hard to train.

2. **Not enough training data**. You might not have enough training data for such a large network, or it might be too costly to label.

3. **Training is too slow**. Training may be extremely slow.

4. **Model has millions of parameters causing a severe risk of overfitting**. A model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances, or they are too noisy.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('bmh')
```

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

In [17]:
```
fashion_mnist = keras.datasets.fashion_mnist

(X_train_full, t_train_full), (X_test, t_test) = fashion_mnist.load_data()

X_train_full.shape, t_train_full.shape, X_test.shape, t_test.shape
```

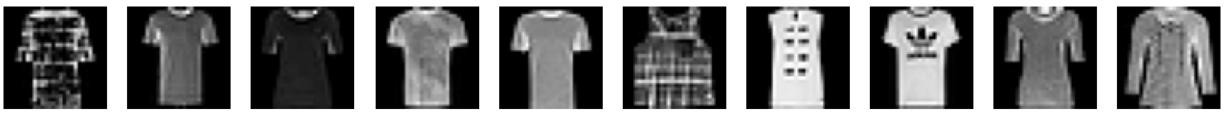
Out[17]:
```
((60000, 28, 28), (60000,), (10000, 28, 28), (10000,))
```

In [18]:
```
# Training and Validation sets
# First 5,000 samples as validation and the remaining ones as training samples
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:15000] / 255.0
t_valid, t_train = t_train_full[:5000], t_train_full[5000:15000]

X_test = X_test / 255.0
```

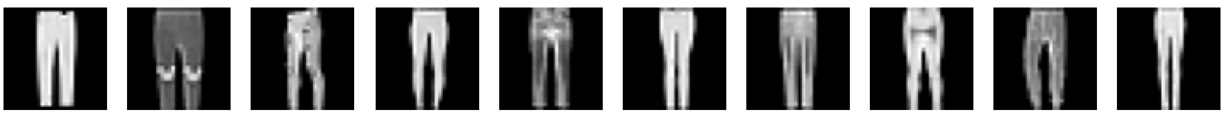
In [19]:
```python
class_names = np.array(["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
                        "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"])
```

In [6]:
```python
for i in range(10):
    labels = np.where(t_train==i)[0]
    idx = np.random.permutation(range(len(labels)))
    print(class_names[i])
    plt.figure(figsize=(15,3))
    for j in range(1,11):
        plt.subplot(1,10,j)
        plt.imshow(X_train[labels[j],:,:], cmap='gray')
        plt.axis('off')
    plt.pause(0.01);
```

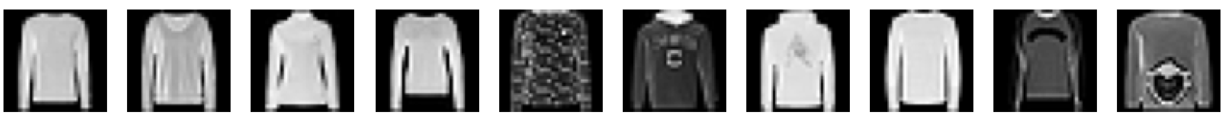
T-shirt/top



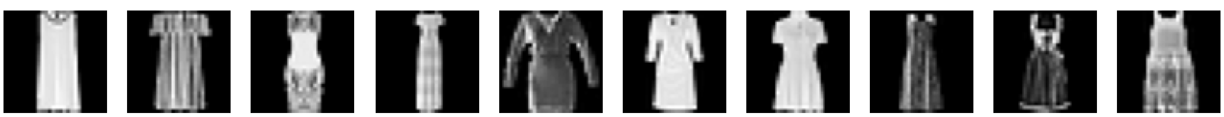
Trouser



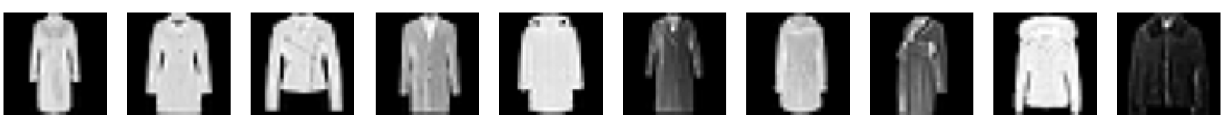
Pullover



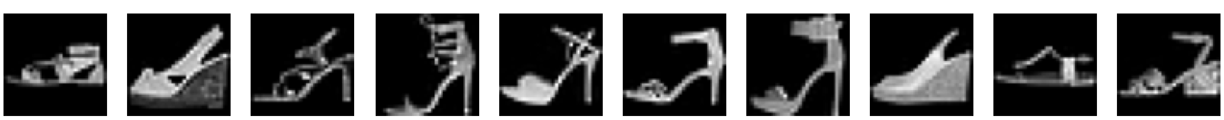
Dress



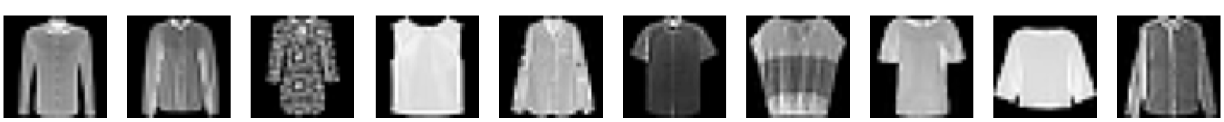
Coat



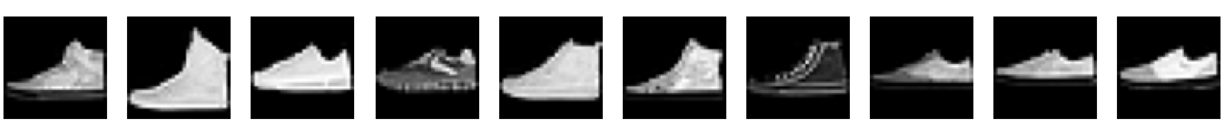
Sandal



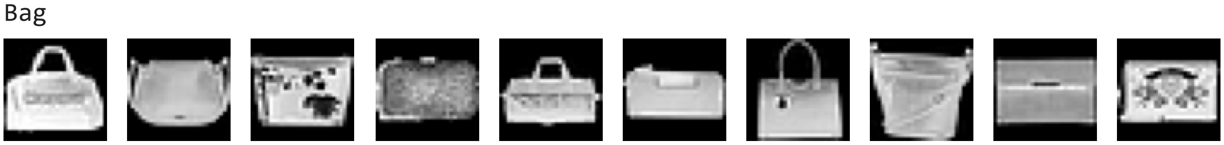
Shirt



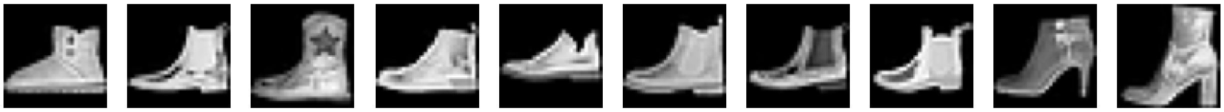
Sneaker



Bag



Ankle boot



Creating the model using the Sequential API

Now let's build the neural network! Here is a classification MLP with two hidden layers:

In [14]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])

model.output_shape
```

Out[14]:
```
(None, 10)
```

The model's `summary()` method displays all the model's layers, including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (None means the batch size can be anything), and its number of parameters. The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters:

In [8]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 300)               235500

 dense_1 (Dense)             (None, 100)               30100

 dense_2 (Dense)             (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Note that Dense layers often have a lot of parameters. For example, the first hidden layer has 784×300 connection weights, plus 300 bias terms, which adds up to 235,500 parameters! This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the **risk of overfitting**, especially when you do not have a lot of training data.

Compiling the Model

After a model is created, you must call its `compile()` method to specify the loss function and the optimizer to use. Optionally, you can specify a list of extra metrics to compute during training and evaluation:

In [9]:
```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

We use the `sparse_categorical_crossentropy` loss because we have sparse labels (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are exclusive.

If instead we had one target probability per class for each instance (such as one-hot vectors, e.g. $[0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]$ to represent class 3), then we would need to use the `categorical_crossentropy` loss instead.

- If you want to convert sparse labels (i.e., class indices) to one-hot vector labels, use the `keras.utils.to_categorical()` function.

If we were doing binary classification or multilabel binary classification, then we would use the `sigmoid` (i.e., logistic) activation function in the output layer instead of the `softmax` activation function, and we would use the `binary_crossentropy` loss.

In [10]:
```
optimizer=keras.optimizers.SGD(learning_rate=0.01)
optimizer
```

Out[10]:
```
<keras.optimizers.optimizer_v2.gradient_descent.SGD at 0x2155e3e67f0>
```

In [11]:
```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

Training and evaluating the model

Now the model is ready to be trained. For this we simply need to call its `fit()` method:

In [16]:
```
10000/32
```

Out[16]:
```
312.5
```

In [12]:
```
history = model.fit(X_train, t_train, epochs=10, batch_size=32,
                    validation_data=(X_valid, t_valid))
```

```
Epoch 1/10
313/313 [==============================] - 1s 2ms/step - loss: 1.1846 - accuracy: 0.6
249 - val_loss: 0.7780 - val_accuracy: 0.7558
Epoch 2/10
313/313 [==============================] - 1s 2ms/step - loss: 0.7117 - accuracy: 0.7
611 - val_loss: 0.6642 - val_accuracy: 0.7692
Epoch 3/10
313/313 [==============================] - 1s 2ms/step - loss: 0.6155 - accuracy: 0.7
899 - val_loss: 0.6241 - val_accuracy: 0.7864
Epoch 4/10
313/313 [==============================] - 1s 2ms/step - loss: 0.5632 - accuracy: 0.8
078 - val_loss: 0.5427 - val_accuracy: 0.8162
Epoch 5/10
313/313 [==============================] - 1s 2ms/step - loss: 0.5256 - accuracy: 0.8
199 - val_loss: 0.5487 - val_accuracy: 0.8150
Epoch 6/10
313/313 [==============================] - 1s 2ms/step - loss: 0.5038 - accuracy: 0.8
242 - val_loss: 0.5196 - val_accuracy: 0.8132
Epoch 7/10
313/313 [==============================] - 1s 2ms/step - loss: 0.4853 - accuracy: 0.8
291 - val_loss: 0.4851 - val_accuracy: 0.8388
Epoch 8/10
313/313 [==============================] - 1s 2ms/step - loss: 0.4656 - accuracy: 0.8
342 - val_loss: 0.4812 - val_accuracy: 0.8340
Epoch 9/10
313/313 [==============================] - 1s 2ms/step - loss: 0.4536 - accuracy: 0.8
369 - val_loss: 0.4972 - val_accuracy: 0.8236
Epoch 10/10
313/313 [==============================] - 1s 2ms/step - loss: 0.4426 - accuracy: 0.8
430 - val_loss: 0.4817 - val_accuracy: 0.8330
```

In [13]:

```
history.history
```

Out[13]:
```
{'loss': [1.184567928314209,
  0.7116761207580566,
  0.6154585480690002,
  0.5631874799728394,
  0.5255810618400574,
  0.5037969350814819,
  0.48534125089645386,
  0.4655675292015757,
  0.45364463329315186,
  0.44259002804756165],
 'accuracy': [0.6248999834060669,
  0.7610999941825867,
  0.789900004863739,
  0.8077999949455261,
  0.8198999762535095,
  0.8241999745368958,
  0.8291000127792358,
  0.8342000246047974,
  0.836899995803833,
  0.8429999947547913],
 'val_loss': [0.778011798858642 6,
  0.6641666889190674,
  0.6241042017936707,
  0.5427234768867493,
  0.5486789345741272,
  0.5195819735527039,
  0.48513105511665344,
  0.48115137219429016,
  0.49720728397369385,
  0.4816984236240387],
 'val_accuracy': [0.7558000087738037,
  0.7692000269889832,
  0.7864000201225281,
  0.8162000179290771,
  0.8149999976158142,
  0.8131999969482422,
  0.8388000130653381,
  0.8339999914169312,
  0.8235999941825867,
  0.8330000042915344]}
```

In [14]:
```
history.history.keys()
```

Out[14]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

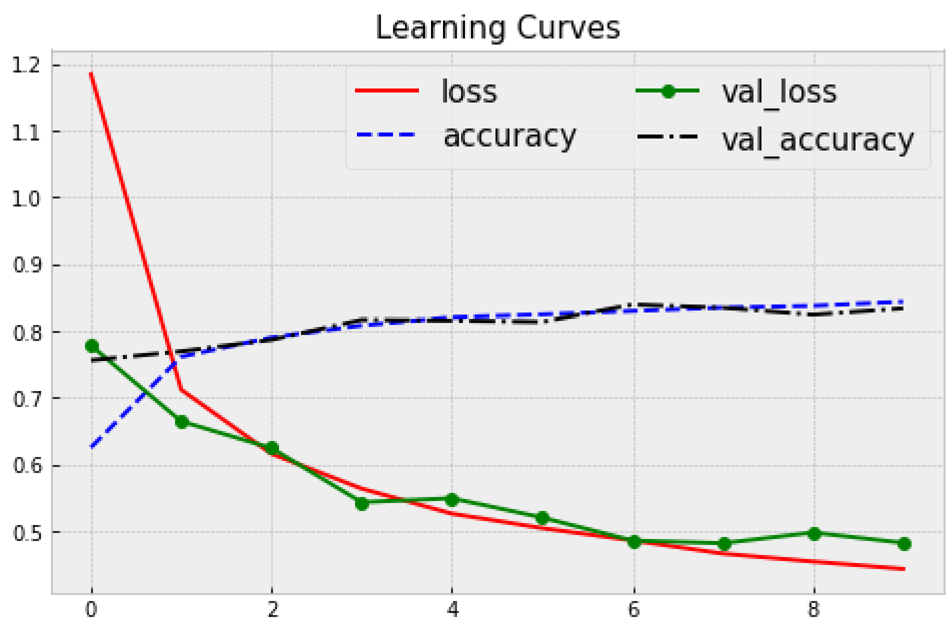
In [15]:
```
history.params
```

Out[15]:
```
{'verbose': 1, 'epochs': 10, 'steps': 313}
```

Learning Curves from the model's `history`

In [16]:
```
key_names = list(history.history.keys())
colors = ['-r','--b','-og','-.k']

plt.figure(figsize=(8,5))
for i in range(len(key_names)):
    plt.plot(history.history[key_names[i]], colors[i], label=key_names[i])
```

```python
plt.legend(fontsize=15,ncol=2)
plt.title('Learning Curves', size=15)
plt.xlabel('Epochs', size=15);
```



In [17]:
```python
model.evaluate(X_test, t_test)
```

```
313/313 [==============================] - 0s 1ms/step - loss: 0.5150 - accuracy: 0.8
199
```

Out[17]:
```
[0.5150206685066223, 0.8198999762535095]
```

Remember to resist the temptation to tweak the hyperparameters on the test set, or else your estimate of the generalization error will be too optimistic.

Using the model to make predictions

Next, we can use the model's `predict()` method to make predictions on new instances. Let's use test samples as an example:

In [18]:
```python
X_new = X_test[:3]
X_new.shape
```

Out[18]:
```
(3, 28, 28)
```

In [19]:
```python
y_proba = model.predict(X_new)
y_proba.round(2)
```

```
1/1 [==============================] - 0s 59ms/step
```

Out[19]:
```
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.04, 0.  , 0.1 , 0.  , 0.86],
       [0.  , 0.  , 0.97, 0.  , 0.01, 0.  , 0.02, 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

As you can see, for each instance the model estimates one probability per class, from class 0 to class 9.

In [20]:
```python
y_pred = np.argmax(y_proba,axis=1)
y_pred
```

Out[20]:
```
array([9, 2, 1], dtype=int64)
```

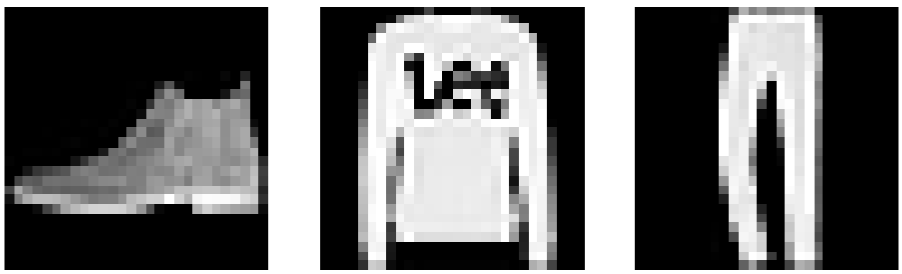
In [21]:
```python
class_names[y_pred]
```

Out[21]:
```
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

In [22]:
```python
t_test[:3]
```

Out[22]:
```
array([9, 2, 1], dtype=uint8)
```

In [23]:
```python
plt.figure(figsize=(8,5))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(X_new[i],cmap='gray')
    plt.axis('off');
```



---

1. The Vanishing/Exploding Gradient Problem

- Backpropagation works by backpropagating the error gradients
- Gradients becomes smaller and smaller as it progresses down to lower layers, this is known as the **vanishing gradient problem**

Solutions:

1. Make variance of the output of each layer equal to the variances of its inputs
2. Utilize non-saturating activation functions
3. Batch normalization

1.1. Glorot and He Initialization

In their paper, Xavier Glorot and Yoshua Bengio propose a way to significantly alleviate the vanishing gradient problem. For the signal to flow properly, the authors argue that we need **the variance of the outputs of each layer to be equal to the variance of its inputs**, and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction.

- It is actually not possible to guarantee both unless the layer has an equal number of inputs and neurons (these numbers are called the fan-in and fan-out of the layer).

But they proposed a good compromise that has proven to work very well in practice: the connection weights of each layer must be initialized randomly as a Gaussian distribution with mean 0 and variance equal to 1 over the average fanning. This is known as the **Xavier (or Glorot) Initialization**.

In this table, you can see two other initializations that have also been shown to work empirically:

Initialization	Activation Functions	σ^2 (Normal)
Glorot (2000)	None, Tanh, Sigmoid, Softmax	$1/\text{fan}_{\text{avg}}$
He (2015)	ReLU and its variants	$2/\text{fan}_{\text{in}}$
LeCunn (1999)	SELU	$1/\text{fan}_{\text{in}}$

By default, Keras uses **Glorot initialization with a uniform distribution**.

When creating a layer, you can change this to He initialization by setting `kernel_initializer="he_uniform"` or `kernel_initializer="he_normal"` like this:

In [24]:
```
keras.layers.Dense(10, activation='relu', kernel_initializer='he_normal')
```

Out[24]:
```
<keras.layers.core.dense.Dense at 0x2156ef68af0>
```

If you want He initialization with a uniform distribution but based on fan_{avg} than fan_{in}, you can use the `VarianceScaling` initializer like this:

In [25]:
```
he_avg_init = keras.initializers.VarianceScaling(scale=2.,
                                                 mode='fan_avg',
                                                 distribution='uniform')

keras.layers.Dense(10, activation='sigmoid', kernel_initializer=he_avg_init)
```

Out[25]:
```
<keras.layers.core.dense.Dense at 0x2156ef75af0>
```

1.2. Non-Saturing Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing gradients problems were in part due to a poor choice of activation function.

- In deep architectures, sigmoid activation functions tend to saturate.

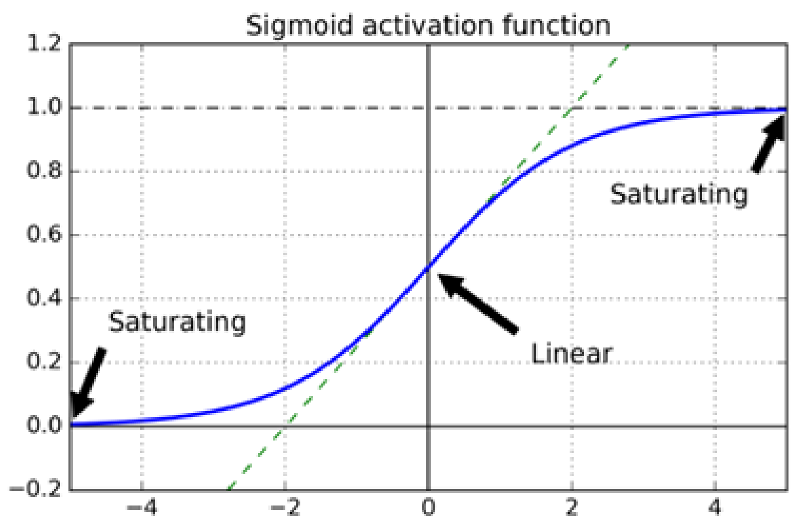
Other activation functions behave much better, in particular the **ReLU activation function**, mostly because it does not saturate for positive values and also because it is quite fast to compute.

- Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the dying neurons: during training, some neurons effectively die, meaning they stop outputting anything other than 0. When this happens, it just keeps outputting 0s, and gradient descent does not affect it anymore since the gradient of the ReLU function is 0 when its input is negative.

To solve this problem, you may want to use a variant of the ReLU function, such as the **leaky ReLU**.

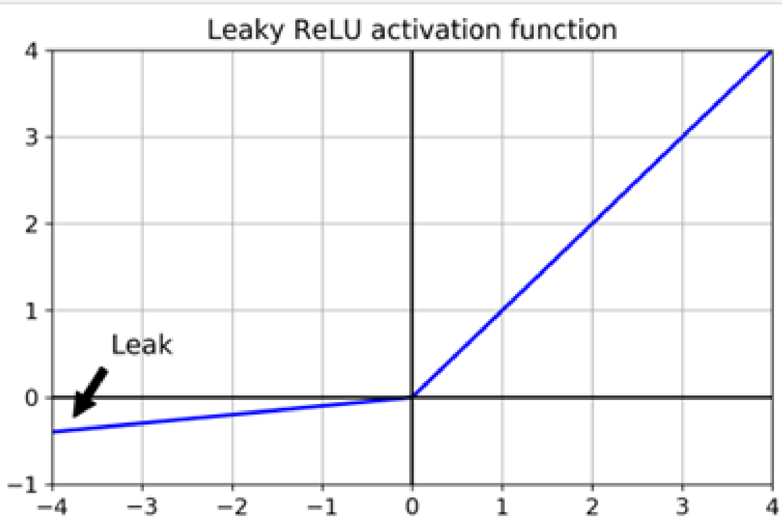
In [26]:
```
from IPython.display import Image
Image('figures/sigmoid.png',width=400)
# Source: "Hands-on machine learning with
#scikit-learn and tensorflow" by Geron Aurelien
```

Out[26]:



In [27]:
```
Image('figures/leaky_relu.png',width=400)
# Source: "Hands-on machine learning with
#scikit-learn and tensorflow" by Geron Aurelien
```

Out[27]:



- Current research has investigated different approaches to selecting the right slope for negative values. Namely, considering it a learnable parameter during training. This is known as the parametric leaky ReLU.

An alternative is to use the **Scaled Exponential Linear Unit, or SELU**. The output of each layer will tend to preserve mean 0 and standard deviation 1 during training, which solves the vanishing gradient problem.

The **Exponential Linear Unit (ELU)** is defined as:

$$\phi(x) = \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & x \leq 0 \end{cases}$$

typical values for α are $0.1 \leq \alpha \leq 0.3$.

The **Scaled Exponential Linear Unit (SELU)** is defined as:

$$\phi(x) = \lambda \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & x \leq 0 \end{cases}$$

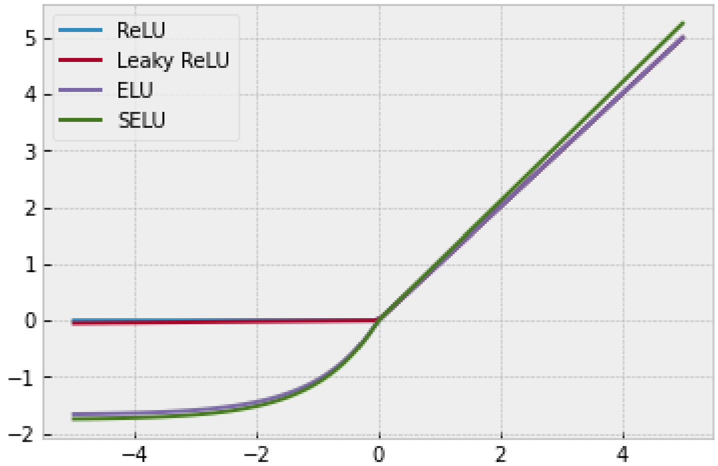
The authors proposed to consider $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$. When using this activation function in practice, one must use *Lecun Normal* for weight initialization, and if dropout wants to be applied, one should use *AlphaDropout*. More on this later in the code section.

In general we have the following relationship between the different activation function options:

> ### SELU > ELU > Leaky ReLU (and its variants) > ReLU > tanh > sigmoid

In [13]:
```
def activation_functions(x, alpha=1.67326, lam=1.0507):

    linear   = x
    sigmoid  = 1/(1+np.exp(-x))
    tanh     = (np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
    softplus = np.log(1+np.exp(x))

    relu = x.copy()
    relu[x<=0] = 0

    leaky_relu = x.copy()
    leaky_relu[x<=0]=0.01*x[x<=0]

    elu = x.copy()
    elu[x<=0] = alpha*(np.exp(x[x<=0])-1)

    selu = lam*x.copy()
    selu[x<=0] = lam*alpha*(np.exp(x[x<=0])-1)

    functions = np.vstack((linear, sigmoid, tanh, softplus, relu, leaky_relu, elu, sel
    names = ['Linear', 'Sigmoid','Tanh','Softplus','ReLU','Leaky ReLU','ELU','SELU']

    return functions, names

x = np.linspace(-5,5,100)
functions, names = activation_functions(x)

for i in [4,5,6,7]:
```

```
    plt.plot(x, functions[:,i], label=names[i])
plt.legend();
```



To use the leaky ReLU activation function, create a model just after the layer you want to apply it to:

In [28]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, kernel_initializer='he_normal'),
    keras.layers.LeakyReLU(alpha=0.2),
    keras.layers.Dense(100, activation='relu', kernel_initializer='he_normal'),
    keras.layers.Dense(10, activation='softmax')
])
```

For SELU activation, set `activation="selu"` and `kernel_initializer="lecun_normal"` when creating a layer:

In [29]:
```
layer = keras.layers.Dense(10, activation='selu',
                           kernel_initializer='lecun_normal')
```

1.3. Batch Normalization

The other popular approach for alleviating the vanishing gradient problem is to use **batch normalization**, first published in 2015.

- The technique consists of adding an operation in the model just before or after the activation function of each hidden layer, simply **zero-centering and normalizing each input then scaling and shifting the result** using two new learnable parameter vectors per layer: one for scaling, the other for shifting.

In other words, this operation lets the model learn the optimal scale and mean of each of the layer's inputs.

In many cases, if you add a batch normalization layer as the very first layer of your neural network, you do not need to standardize your training set: the batch normalization layer will do it for you. Well, approximately, since it only looks at one batch at a time, and it can also rescale and shift each input feature.

Just add a `BatchNormalization` layer before or after each hidden layer's activation function, and optionally add a batch normalization layer as well as the first layer in your model:

In [30]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation='elu', kernel_initializer='he_normal'),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation='elu', kernel_initializer='he_normal'),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation='softmax')
])
```

In [31]:
```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 flatten_2 (Flatten)         (None, 784)               0         
                                                                 
 batch_normalization (BatchN  (None, 784)              3136      
 ormalization)                                                   
                                                                 
 dense_9 (Dense)             (None, 300)               235500    
                                                                 
 batch_normalization_1 (Batc  (None, 300)              1200      
 hNormalization)                                                 
                                                                 
 dense_10 (Dense)            (None, 100)               30100     
                                                                 
 batch_normalization_2 (Batc  (None, 100)              400       
 hNormalization)                                                 
                                                                 
 dense_11 (Dense)            (None, 10)                1010      
                                                                 
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
_________________________________________________________________
```

The authors of the Batch Normalization (BN) paper argued in favor of adding the BN layers before the activation functions, rather than after (as we just did). There is some debate about this, as which is preferable seems to depend on the task - you can experiment with this too to see which option works best on your dataset.

To add the BN layers before the activation functions, you must remove the activation function from the hidden layers and add them as separate layers after the BN layers.

Moreover, since a Batch Normalization layer includes one offset parameter per input, you can remove the bias term from the previous layer (just pass use_bias=False when creating it):

In [32]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer='he_normal', use_bias=True),
    keras.layers.BatchNormalization(),
    keras.layers.Activation('elu'),

    keras.layers.Dense(100, kernel_initializer='he_normal', use_bias=True),
    keras.layers.BatchNormalization(),
    keras.layers.Activation('elu'),

    keras.layers.Dense(10, activation='softmax')
])
```

In [33]:
```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_3 (Flatten)         (None, 784)               0

 batch_normalization_3 (Batc  (None, 784)              3136
 hNormalization)

 dense_12 (Dense)            (None, 300)               235200

 batch_normalization_4 (Batc  (None, 300)              1200
 hNormalization)

 activation (Activation)     (None, 300)               0

 dense_13 (Dense)            (None, 100)               30000

 batch_normalization_5 (Batc  (None, 100)              400
 hNormalization)

 activation_1 (Activation)   (None, 100)               0

 dense_14 (Dense)            (None, 10)                1010

=================================================================
Total params: 270,946
Trainable params: 268,578
Non-trainable params: 2,368
_________________________________________________________________
```

1.4. Gradient Clipping

Another popular technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called **Gradient Clipping**.

This technique is most often used in recurrent neural networks (RNNs), as Batch Normalization is tricky to use in RNNs. For other types of networks, BN is usually sufficient. In `Keras`, implementing Gradient Clipping is just a matter of setting the `clipvalue` or `clipnorm` argument when creating an optimizer, like this:

In [34]:
```
optimizer = keras.optimizers.SGD(clipvalue=1.0)

model.compile(loss='sparse_categorical_crossentropy',
              optimizer=optimizer)
```

This optimizer will clip every component of the gradient vector to a value between -1.0 and 1.0.

Saving and Restoring a Model

When using the Sequential API or the Functional API, saving a trained Keras model is as simple as it gets:

In [35]:
```
model.save('my_keras_model.h5')
```

Keras will use the HDF5 format to save both the model's architecture (including every layer's hyperparameters) and the values of all the model parameters for every layer (e.g., connection weights and biases). It also saves the optimizer (including its hyperparameters and any state it may have).

Loading the model is just as easy:

In [36]:
```
model = keras.models.load_model('my_keras_model.h5')
```

---

2. Transfer Learning

It is generally not a good idea to train a very large deep neural network architecture from scratch. Specially if you have limited computational resources and/or a small training dataset.

Instead, you should always try to find an **existing neural network that accomplishes a similar task** to the one you are trying to tackle, then just **reuse the lower layers of this network**: this is called **transfer learning**. It will not only speed up training considerably but will also require much less training data.

- For example, suppose that you have access to a Deep Neural Network (DNN) that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects. You now want to train a DNN to classify specific types of vehicles. These tasks are very similar, even partly overlapping, so you should try to reuse parts of the first network.

The **output layer of the original model should usually be replaced since it is most likely not useful at all for the new task**, and it may not even have the right number of outputs for the new task.

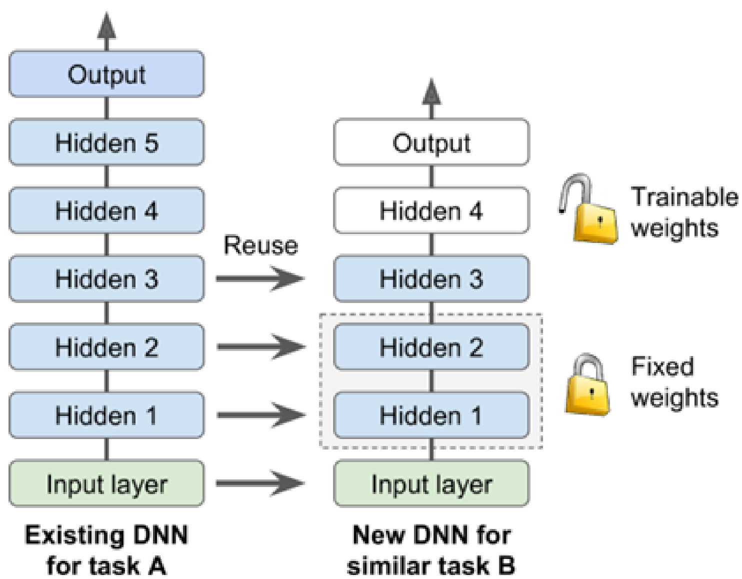
Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. Try freezing all the reused layers first, then train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves.

- The more training data you have, the more layers you can unfreeze.

It is also useful to reduce the learning rate when you unfreeze reused layers: this will avoid wrecking their fine-tuned weights.

In [37]:
```
Image('figures/transfer_learning.png',width=400)
# Source: "Hands-on machine learning with scikit-learn and tensorflow" by Geron Aureli
```

Out[37]:



---

3. Faster Optimizers

3.1. Gradient Descent with Momentum

Momentum optimization was first **introduced in 1964**. Recall that Gradient Descent simply updates the weights theta by directly subtracting the gradient of the cost function multiplied by the learning rate:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \nabla J(\theta^{(t)})$$

Momentum optimization cares a great deal about what previous gradients were. At each iteration, it subtracts the local gradient from the momentum vector m (multiplied by the learning rate), and it updates the weights by simply adding this momentum vector. In other words, the gradient is used for acceleration, not for speed.

$$m^{(t+1)} \leftarrow \beta m^{(t)} - \eta \nabla J(\theta^{(t)})$$
$$\theta^{(t+1)} \leftarrow \theta^{(t)} + m^{(t+1)}$$

To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β, simply called the **momentum**, which must be set between 0 (high friction) and 1 (no friction). A **typical momentum value is 0.9**.

- Momentum optimization will roll down the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use Batch Normalization, the upper layers will often end up having inputs with very different scales, so using Momentum optimization helps a lot. It can also help roll past local optima.

Implementing momentum optimization in `Keras` just use the optimizer `SGD` and set its momentum hyperparameter:

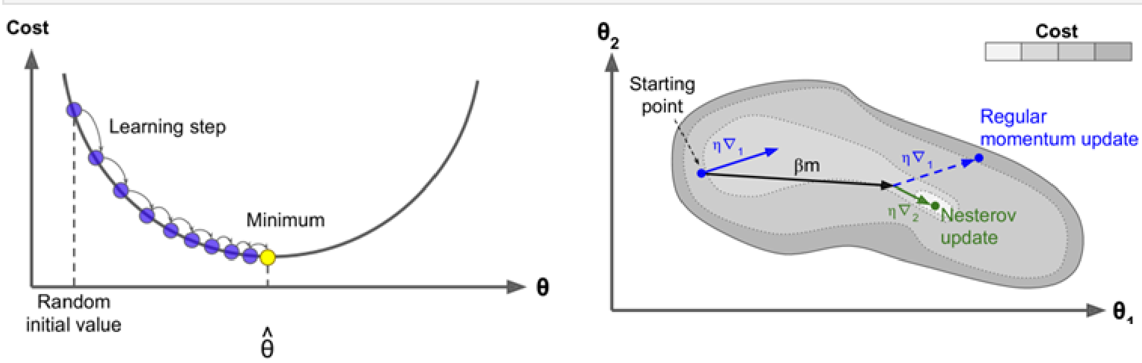
In [53]:
```
optimizer = keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
```

One small variant to Momentum optimization, called **Nesterov Accelerated Gradient or NAG**, proposed by Nesterov in 1983, is almost always faster than vanilla Momentum optimization. NAG measures the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum, as seen in this picture. Making the convergence even faster than standard momentum.

$$m^{(t)} \leftarrow \theta^{(t)} + \mu(\theta^{(t)} - \theta^{(t-1)})$$
$$\theta^{(t+1)} \leftarrow \beta m^{(t)} - \eta \nabla J(m^{(t)})$$

In [54]:
```
Image('figures/momentum.png',width=900)
# Source: "Hands-on machine learning with scikit-learn and tensorflow" by Geron Aureli
```

Out[54]:



In [55]:
```
optimizer = keras.optimizers.SGD(learning_rate=0.001, momentum=0.9, nesterov=True)
```

3.2. Gradient Descent with Adaptive Learning Rate

In addition to being able to speed up Gradient Descent, we can also **update the learning rate** to perform larger or smaller updates depending on their importance.

$$\Delta w_{ij}^{(t)} = -\eta^{(t)} \nabla J(w_{ij}^{(t)})$$

- **AdaGrad (Adaptive Gradient algorithm, 2011)**: It simply allows the learning rate to decrease based on the parameters of the network. So it makes big updates for infrequent parameters, and small updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data. Its main weakness is that its learning rate is always decreasing and decaying.

- **RMSProp (Root Mean Squared Progagation algorithm, 2012):** RMSProp is also a method in which the learning rate is adapted for each of the parameters in the network. The idea is to divide the learning rate for a parameter by a running average of the magnitudes of recent gradients for that weight.

- **Adam (Adaptive Moment Estimation, 2015):** Adam combines RMSProp and momentum learning. It is by far the most common method used today. Adam also computes adaptive learning rates for each parameter of the network. In this optimization algorithm, running averages of both the gradients and the second moments of the gradients are used.

Adam, or **Adaptive Moment Estimation**, was introduced in 2015. Adam is currently the standard optimization learning algorithm for training deep neural networks as it combines the ideas of **momentum optimization and adaptive learning rate**.

- Just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and it keeps track of an exponentially decaying average of past squared gradients.

Since Adam is an adaptive learning rate algorithm, it requires less tuning of the learning rate hyperparameter. You can often use the default value of eta equal to $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

Nesterov Adam or Nadam optimization, introduced in 2016, is simply Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam.

In [56]:
```
optimizer = keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
```

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

3.3. Learning Rate Scheduler

One approach is to start with a large learning rate and divide it by 3 until the training algorithm stops diverging. You will not be too far from the optimal learning rate, which will learn quickly and converge to good solution.

However, you can do better than a constant learning rate: if you start with a high learning rate and then reduce it once it stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training. These strategies are called learning schedules. Some examples include:

1. Power scheduling, where you set the learning rate to a function of the iteration number t,

$$\eta(t) = \eta_0/(1 + t/s)^c$$

where c is typically set to $c = 1$, t is the iteration number and the steps s are hyperparameters.

1. exponential scheduling, and
2. piecewise constant scheduling.

Implementing power scheduling with `Keras`, set the `decay` hyperparameter when creating an optimizer:

In [58]:
```
optimizer = keras.optimizers.SGD(learning_rate=0.01, decay=1e-4)
```

The `decay` is the inverse of s (the number of steps it takes to divide the learning rate) and the value for c is set to default $c = 1$.

---

4. Avoid Overfitting Through Regularization

4.1. L1 and L2 Regularization

Deep neural networks typically have tens of thousands of parameters, sometimes even millions. With so many parameters, the network has an incredible amount of freedom and can fit a huge variety of complex datasets. But this great flexibility also means that it is prone to overfitting the training set.

- Regularization constraints model parameters from becoming too large

Ridge regularizer or **L2-Norm** adds the term $\lambda \sum_{i=1}^{M} w_i^2$ to the cost function during training, where λ controls tradeoff between minimizing error term and penalty term in cost function.

Lasso regularizer or **L1-Norm** adds the term $\lambda \sum_{i=1}^{M} |w_i|$. The Lasso regularizer promotes **sparsity** of the weight vector.

Elastic Net adds the term $\beta\lambda \sum_{i=1}^{M} |w_i| + \frac{1-\beta}{2}\lambda \sum_{i=1}^{M} w_i^2$.

In [59]:
```
layer = keras.layers.Dense(100,
                           activation='elu',
```

```
                                   kernel_initializer='he_normal',
                                   kernel_regularizer=keras.regularizers.l2(0.01))
```

In [60]:
```
layer = keras.layers.Dense(100,
                                   activation='elu',
                                   kernel_initializer='he_normal',
                                   kernel_regularizer=keras.regularizers.l1(0.01))
```

In [61]:
```
layer = keras.layers.Dense(100,
                                   activation='elu',
                                   kernel_initializer='he_normal',
                                   kernel_regularizer=keras.regularizers.l1_l2(0.01))
```

4.2. Dropout

There are several strategies for network pruning which include:

- **Network Growing**: Start with a small MLP and add to it when unable to meet design specifications
- **Network Pruning**: Start with a large MLP and prune it by eliminating weights (driving them to zero)
- **Complexity Regularization**: Need an appropriate trade-off between reliability of training data and goodness of the model/NN architecture.

Dropout is one of the most popular regularization techniques for deep neural net works. It was proposed by Geoffrey Hinton in 2012.

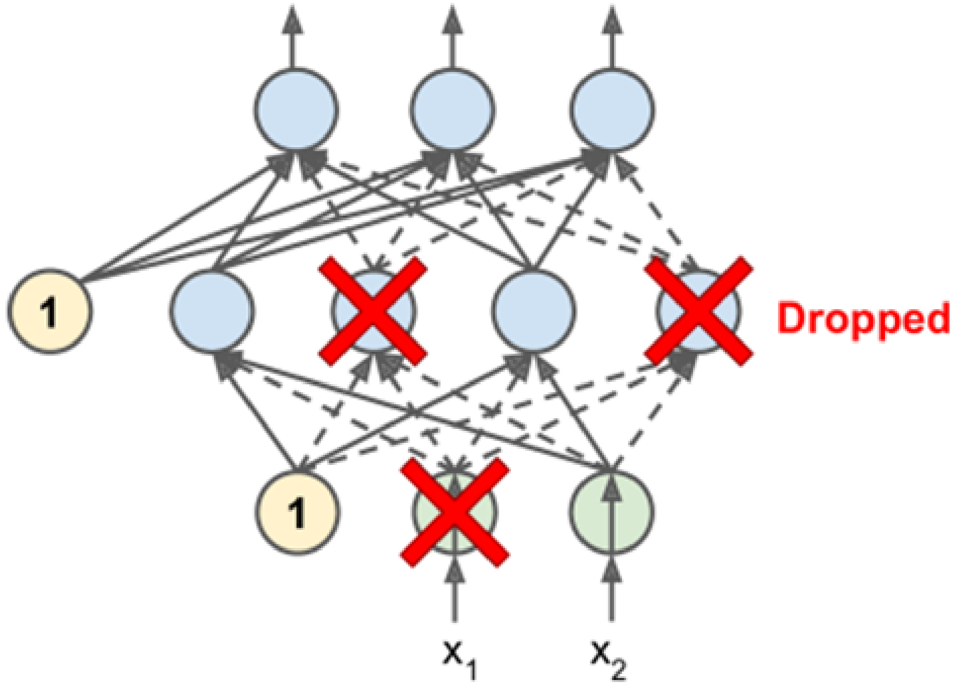
It is a fairly simple algorithm: at every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability p of being temporarily "dropped out," meaning it will be entirely ignored during this training step, but it may be active during the next step. The hyperparameter p is called the dropout rate, and it is typically set to 50%.

After training, neurons don't get dropped anymore. That's it!

Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end you get a more robust network that generalizes better.

In [62]:
```
Image('figures/dropout.png',width=500)
# Source: "Hands-on machine learning with scikit-learn and tensorflow" by Geron Aureli
```

Out[62]:



In [20]:
```
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')
])
```

- If you observe that the model is overfitting, you can increase the dropout rate.
- Conversely, you should try decreasing the dropout rate if the model underfits the training set.
- It can also help to increase the dropout rate for large layers, and reduce it for small ones.
- Moreover, many state-of-the-art architectures only use dropout after the last hidden layer, so you may want to try this if full dropout is too strong.

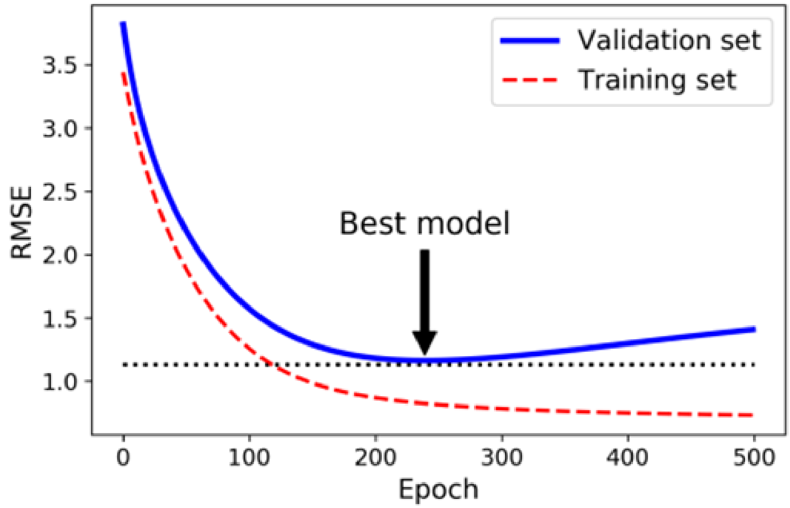
4.3. Early Stopping Criteria

A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called early stopping. As you can see in this figure, as the epochs go by, the algorithm learns and its prediction error (RMSE) on the training set naturally goes down, and so does its prediction error on the validation set. However, after a while the validation error stops decreasing and starts to go back up. This indicates that the model has started to overfit the training data.

```
Image('figures/early_stopping.png',width=400)
```

In [64]:
```
# Source: "Hands-on machine learning with scikit-learn and tensorflow" by Geron Aureli
```

Out[64]:



This approach may result in a premature termination of learning.

an alternative is to consider the backpropagation algorithm to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small threshold. However, learning may take a long time and it requires computation the norm of the gradient vector.

In [22]:
```
callback = keras.callbacks.EarlyStopping(monitor='val_loss',
                                         patience=3,
                                         restore_best_weights=True)
# This callback will stop the training when there is no improvement in
# the loss for three consecutive epochs.
```

In [23]:
```
optimizer = keras.optimizers.Adam(learning_rate=0.01, beta_1=0.9, beta_2=0.999)

model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
             metrics=['accuracy'])

history = model.fit(X_train, t_train, epochs=10,
                    validation_data=(X_valid, t_valid),
                    callbacks=[callback])
```

```
Epoch 1/10
313/313 [==============================] - 2s 4ms/step - loss: 1.0834 - accuracy: 0.6
523 - val_loss: 0.6456 - val_accuracy: 0.7786
Epoch 2/10
313/313 [==============================] - 1s 3ms/step - loss: 0.8085 - accuracy: 0.7
141 - val_loss: 0.6738 - val_accuracy: 0.7624
Epoch 3/10
313/313 [==============================] - 1s 3ms/step - loss: 0.7859 - accuracy: 0.7
195 - val_loss: 0.6239 - val_accuracy: 0.7948
Epoch 4/10
313/313 [==============================] - 1s 3ms/step - loss: 0.7365 - accuracy: 0.7
446 - val_loss: 0.6736 - val_accuracy: 0.7788
Epoch 5/10
313/313 [==============================] - 1s 3ms/step - loss: 0.7627 - accuracy: 0.7
368 - val_loss: 0.5324 - val_accuracy: 0.8198
Epoch 6/10
313/313 [==============================] - 1s 3ms/step - loss: 0.7488 - accuracy: 0.7
373 - val_loss: 0.6026 - val_accuracy: 0.7964
Epoch 7/10
313/313 [==============================] - 1s 3ms/step - loss: 0.7649 - accuracy: 0.7
362 - val_loss: 0.5940 - val_accuracy: 0.7782
Epoch 8/10
313/313 [==============================] - 1s 3ms/step - loss: 0.7665 - accuracy: 0.7
441 - val_loss: 0.7949 - val_accuracy: 0.6988
```

---

5. Online/Stochastic, Batch and Mini-Batch Learning

The number of samples used to perform a single training iteration (forward pass + backward pass) will impact the results.

- **Online Learning** (or stochastic/sequential learning): uses one sample to update the parameters of the network.
- **Batch Learning:** uses the entire training set to update the parameters of the network.

Another way to successfully train a deep neural network is to use **mini-bath learning**.

- As a contrast, batch learning uses the entire training data to make changes on the model parameters by using the average gradient value. The **convergence of batch learning is smooth**, as you can see in the figure below, **but it is very slow**.

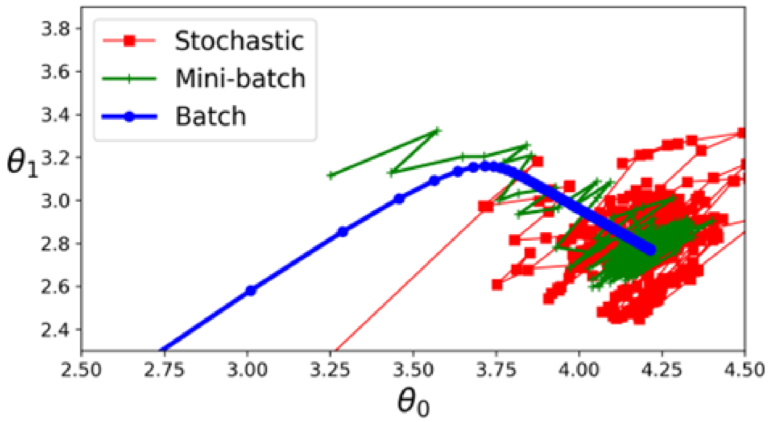
- In Stochastic or **Online learning**, we use a single training instance at a time to compute the gradient and use gradient descent to make changes on the model's parameters. The **convergence for online learning is random**, by nature, so it will be **erratic** as it becomes sensitive to small differences between training instances.

Mini-batch learning is a good compromise between the two other learning strategies. At each step, instead of computing the gradients based on the full training set (as in Batch) or based on just one instance (as in Stochastic), Minibatch computes the gradients on small random sets of instances called minibatches. The main advantage of Mini-batch over Stochastic is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

Common mini-batch sizes are 32, 64 and 128.

In [67]:
```
Image('figures/mini-batch.png',width=400)
# Source: "Hands-on machine learning with scikit-learn and tensorflow" by Geron Aureli
```

Out[67]:



In [70]:
```
history = model.fit(X_train, t_train, epochs=10, batch_size=1,
                    validation_data=(X_valid, t_valid))
```

```
Epoch 1/10
10000/10000 [==============================] - 52s 5ms/step - loss: 3.4150 - accurac
y: 0.0964 - val_loss: 3.4052 - val_accuracy: 0.1024
Epoch 2/10
10000/10000 [==============================] - 35s 4ms/step - loss: 3.2839 - accurac
y: 0.1050 - val_loss: 3.2528 - val_accuracy: 0.0986
Epoch 3/10
10000/10000 [==============================] - 36s 4ms/step - loss: 3.2817 - accurac
y: 0.0971 - val_loss: 2.7665 - val_accuracy: 0.0976
Epoch 4/10
10000/10000 [==============================] - 48s 5ms/step - loss: 3.2909 - accurac
y: 0.1000 - val_loss: 2.4967 - val_accuracy: 0.0986
Epoch 5/10
10000/10000 [==============================] - 49s 5ms/step - loss: 3.2874 - accurac
y: 0.0995 - val_loss: 3.3801 - val_accuracy: 0.0980
Epoch 6/10
10000/10000 [==============================] - 22s 2ms/step - loss: 3.2818 - accurac
y: 0.1002 - val_loss: 3.4568 - val_accuracy: 0.1112
Epoch 7/10
10000/10000 [==============================] - 15s 1ms/step - loss: 3.2801 - accurac
y: 0.0983 - val_loss: 2.9839 - val_accuracy: 0.0986
Epoch 8/10
10000/10000 [==============================] - 16s 2ms/step - loss: 3.2864 - accurac
y: 0.0987 - val_loss: 2.7750 - val_accuracy: 0.1024
Epoch 9/10
10000/10000 [==============================] - 16s 2ms/step - loss: 3.2550 - accurac
y: 0.0984 - val_loss: 3.0428 - val_accuracy: 0.1024
Epoch 10/10
10000/10000 [==============================] - 15s 1ms/step - loss: 3.2777 - accurac
y: 0.0992 - val_loss: 3.5041 - val_accuracy: 0.0980
```

7. Determining Whether to Gather More Data

After the first end-to-end system is established, it is time to measure the performance of the algorithm and determine how to improve it. It is often much better to gather more data than to improve the learning algorithm (or first model of choice).

How does one decide whether to gather more data?

1. Determine whether the performance on the training set is acceptable. If performance on the training set is poor, the learning algorithm is not using the training data that is already available, so there is no reason to gather more data.

   - Instead, try adding complexity to the model by adding more layers or adding more hidden units to each layer.
   - Also, try improving the optimization algorithm, for example by tuning the learning rate.
   - If more complex models and carefully tuned optimization algorithms do not work well, then the problem might be the *quality* of the training data. The data may be too noisy or may not include the right inputs needed to predict the desired outputs. This suggests starting over, collecting cleaner data, or collecting a richer set of features.
2. If the performance on the training set is acceptable, then measure the performance on a test set. If the performance on the test set is also acceptable, then there is nothing left to be done. If test set performance is much worse than training set performance, then gathering more data is one of the most effective solutions. In some applications, gathering more data is simply infeasible or impossible.

   - A simple alternative to gathering more data is to reduce the size of the model or improve regularization, by adjusting hyperparameters such as weight decay coefficients,or by adding regularization strategies such as dropout.
   - If you find that the gap between train and test performance is still unacceptable even after tuning the regularization hyperparameters, then gathering more data is advisable.
3. When deciding whether to gather more data, it is also necessary to decide how much to gather. It is helpful to plot curves showing the relationship between training set size and generalization error.

   - You can experiment with training set sizes on a logarithmic scale, for example, doubling the number of examples between consecutive experiments.

Summary

We covered a wide range of techniques for training deep neural networks. The configuration provided in this table will work fine in most cases, without requiring much hyperparameter tuning.

Hyperparameter	Default Value
Kernel Initializer	LeCunn Initialization
Activation Function	SELU
Regularization	Early stopping
Optimizer	Adam
Learning rate schedule	Performance scheduling
Learning configuration	Mini-batch
Dropout	50%

Decision Maps in MLPs

A Neural Network Playground is a great tool to provide visual interpretation of MLPs performance. You can choose and interpret the effect of the use of different features, architecture size, learning rate, among others.
