Advances in CNN - Part A

Instructor: Santosh Chapaneri

RK University, Rajkot

June 2021

Recap - Neural Networks

In [1]:
```
import sklearn
import numpy as np

import tensorflow as tf
from tensorflow import keras

# to make this notebook's output stable across runs
np.random.seed(45)

%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)
```

In [2]:
```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx
1-ubyte.gz
32768/29515 [================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx
3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1
-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3
-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

In [3]:
```
X_train_full.shape
```

Out[3]:
```
(60000, 28, 28)
```

- Let's split the full training set into a validation set and a (smaller) training set. We also scale the pixel intensities down to the 0-1 range and convert them to floats, by dividing by 255.

In [4]:
```
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0

y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

X_test = X_test / 255.0
```

In [5]:
```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

In [6]:
```
print(class_names[y_train[0]])
plt.imshow(X_train[0], cmap="binary")
plt.axis('off')
plt.show()
```

```
Coat
```

- The validation set contains 5,000 images, and the test set contains 10,000 images

In [7]:
```
print(X_valid.shape)
print(X_test.shape)
```

```
(5000, 28, 28)
(10000, 28, 28)
```

- Let's take a look at a sample of the images in the dataset

In [8]:
```
n_rows = 4
n_cols = 10
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))
for row in range(n_rows):
    for col in range(n_cols):
        index = n_cols * row + col
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(X_train[index], cmap="binary", interpolation="nearest")
        plt.axis('off')
        plt.title(class_names[y_train[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```

In [9]:
```
# Model
model = keras.models.Sequential()

model.add(keras.layers.Flatten(input_shape=[28, 28]))

model.add(keras.layers.Dense(300, activation="relu"))

model.add(keras.layers.Dense(100, activation="relu"))

model.add(keras.layers.Dense(10, activation="softmax"))
```

In [ ]:
```
# Alternatively:
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")
])
```

In [10]:
```
# Layers
model.layers
```

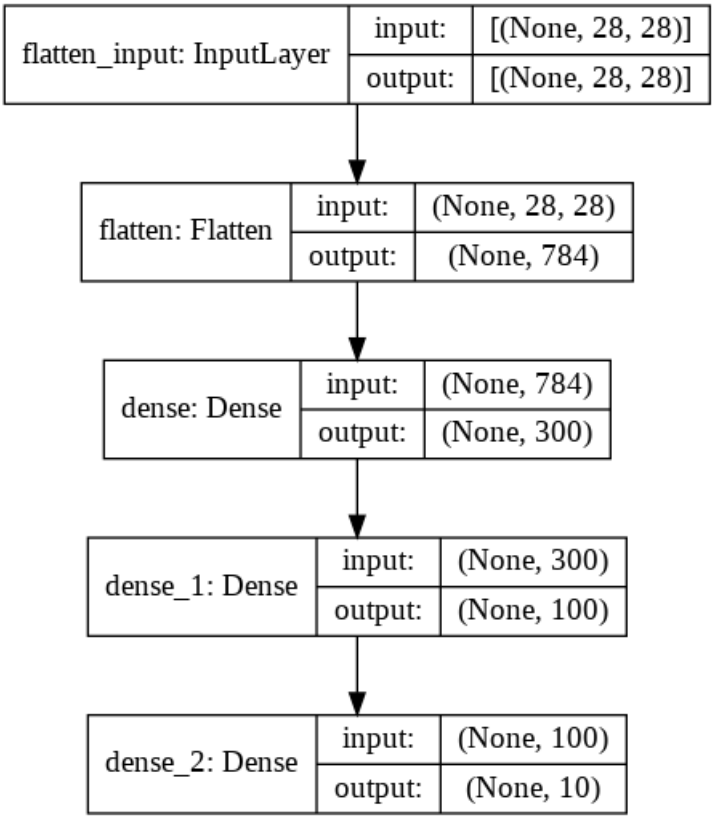
Out[10]:
```
[<tensorflow.python.keras.layers.core.Flatten at 0x7efbf2405350>,
 <tensorflow.python.keras.layers.core.Dense at 0x7efbf2405110>,
 <tensorflow.python.keras.layers.core.Dense at 0x7efbe0216c10>,
 <tensorflow.python.keras.layers.core.Dense at 0x7efbe01c0190>]
```

In [11]:
```
# Summary
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
_________________________________________________________________
```

In [12]:
```
# The network
keras.utils.plot_model(model, "my_fashion_mnist_model.png", show_shapes=True)
```

Out[12]:



In [13]:
```
hidden1 = model.layers[1]
weights, biases = hidden1.get_weights()
weights
```

Out[13]:
```
array([[ 0.00502659,  0.05429789, -0.02767852, ...,  0.02860302,
        -0.01545772, -0.00709982],
       [-0.01978104,  0.0437853 ,  0.03746057, ...,  0.00015489,
        -0.02995867,  0.01817646],
       [ 0.00109173, -0.01929785,  0.00167192, ..., -0.06355966,
        -0.07389013, -0.05587272],
       ...,
       [-0.06193846, -0.05760755, -0.03359643, ...,  0.00054065,
         0.03157959, -0.044904  ],
       [-0.03522344,  0.06875853,  0.00085665, ...,  0.04128237,
        -0.0575419 , -0.06490077],
       [ 0.0536994 , -0.02526898, -0.02960093, ..., -0.06151951,
        -0.02575229,  0.01322784]], dtype=float32)
```

In [14]:
```
weights.shape
```

Out[14]:
```
(784, 300)
```

In [15]:
```
# Compile
model.compile(loss      = "sparse_categorical_crossentropy",
              optimizer= "sgd",
              metrics   = ["accuracy"])
```

This is equivalent to:

```
model.compile(loss=keras.losses.sparse_categorical_crossentropy,
              optimizer=keras.optimizers.SGD(),
              metrics=[keras.metrics.sparse_categorical_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 we were doing binary classification (with one or more binary labels), 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.

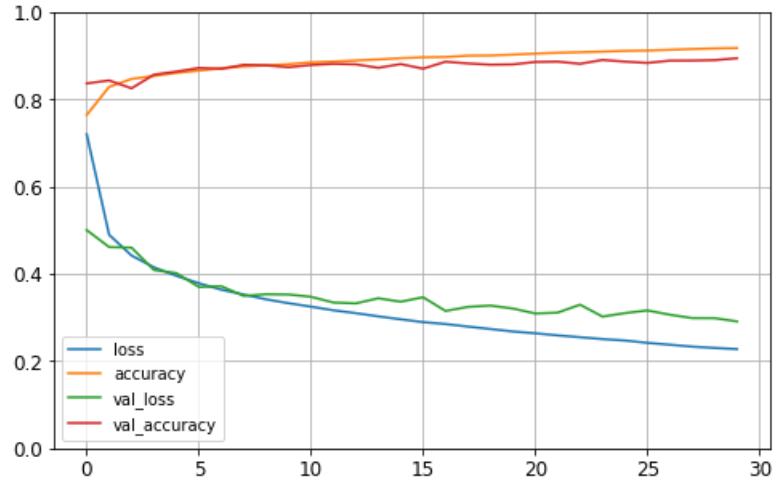
- If you want to convert sparse labels (i.e., class indices) to one-hot vector labels, use the `keras.utils.to_categorical()` function.
- To go the other way round, use the `np.argmax()` function with axis=1.

In [16]:
```
history = model.fit(X_train, y_train, epochs = 30,
                    validation_data = (X_valid, y_valid))
```

```
Epoch 1/30
1719/1719 [==============================] - 7s 2ms/step - loss: 0.7202 - accuracy: 0.7645 - val_l
oss: 0.5004 - val_accuracy: 0.8368
Epoch 2/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.4897 - accuracy: 0.8289 - val_l
oss: 0.4616 - val_accuracy: 0.8436
Epoch 3/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.4421 - accuracy: 0.8471 - val_l
oss: 0.4603 - val_accuracy: 0.8256
Epoch 4/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.4150 - accuracy: 0.8539 - val_l
oss: 0.4089 - val_accuracy: 0.8570
Epoch 5/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3955 - accuracy: 0.8609 - val_l
oss: 0.4018 - val_accuracy: 0.8640
Epoch 6/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3786 - accuracy: 0.8661 - val_l
oss: 0.3701 - val_accuracy: 0.8722
Epoch 7/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3638 - accuracy: 0.8714 - val_l
oss: 0.3717 - val_accuracy: 0.8702
Epoch 8/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3528 - accuracy: 0.8759 - val_l
oss: 0.3495 - val_accuracy: 0.8794
Epoch 9/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3417 - accuracy: 0.8781 - val_l
oss: 0.3534 - val_accuracy: 0.8782
Epoch 10/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3328 - accuracy: 0.8809 - val_l
oss: 0.3528 - val_accuracy: 0.8740
Epoch 11/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3249 - accuracy: 0.8851 - val_l
oss: 0.3475 - val_accuracy: 0.8790
Epoch 12/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3166 - accuracy: 0.8870 - val_l
oss: 0.3344 - val_accuracy: 0.8818
Epoch 13/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3100 - accuracy: 0.8893 - val_l
oss: 0.3324 - val_accuracy: 0.8806
Epoch 14/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3027 - accuracy: 0.8917 - val_l
oss: 0.3441 - val_accuracy: 0.8726
Epoch 15/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2959 - accuracy: 0.8948 - val_l
oss: 0.3362 - val_accuracy: 0.8812
Epoch 16/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2894 - accuracy: 0.8966 - val_l
oss: 0.3465 - val_accuracy: 0.8706
Epoch 17/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2851 - accuracy: 0.8974 - val_l
oss: 0.3150 - val_accuracy: 0.8866
Epoch 18/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2792 - accuracy: 0.9005 - val_l
oss: 0.3244 - val_accuracy: 0.8826
Epoch 19/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2737 - accuracy: 0.9009 - val_l
oss: 0.3274 - val_accuracy: 0.8798
Epoch 20/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2680 - accuracy: 0.9028 - val_l
oss: 0.3207 - val_accuracy: 0.8804
Epoch 21/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2639 - accuracy: 0.9052 - val_l
oss: 0.3093 - val_accuracy: 0.8860
Epoch 22/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2590 - accuracy: 0.9071 - val_l
oss: 0.3112 - val_accuracy: 0.8868
Epoch 23/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2548 - accuracy: 0.9083 - val_l
oss: 0.3294 - val_accuracy: 0.8818
Epoch 24/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2506 - accuracy: 0.9097 - val_l
oss: 0.3021 - val_accuracy: 0.8906
Epoch 25/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2473 - accuracy: 0.9115 - val_l
oss: 0.3100 - val_accuracy: 0.8868
Epoch 26/30
```

```
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2419 - accuracy: 0.9123 - val_l
oss: 0.3163 - val_accuracy: 0.8838
Epoch 27/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2376 - accuracy: 0.9139 - val_l
oss: 0.3065 - val_accuracy: 0.8890
Epoch 28/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2332 - accuracy: 0.9157 - val_l
oss: 0.2987 - val_accuracy: 0.8892
Epoch 29/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2303 - accuracy: 0.9171 - val_l
oss: 0.2986 - val_accuracy: 0.8902
Epoch 30/30
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2277 - accuracy: 0.9179 - val_l
oss: 0.2910 - val_accuracy: 0.8946
```

In [17]:
```
# Plots
import pandas as pd
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



In [18]:
```
# Test the model
model.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3205 - accuracy: 0.8869
```

Out[18]:
```
[0.32046693563461304, 0.886900007724762]
```

- Using the trained model to make predictions

In [19]:
```
X_new = X_test[:3]

y_proba = model.predict(X_new)
y_proba.round(2)
```

Out[19]:
```
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.  , 0.  , 0.99],
       [0.  , 0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

- output is probability per class

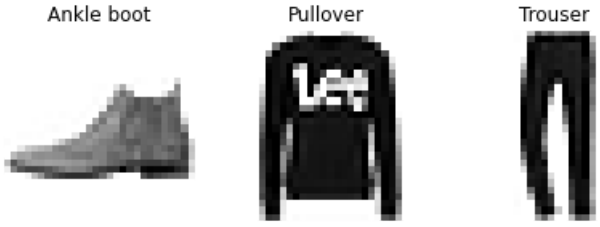
In [21]:
```
y_pred = np.argmax(model.predict(X_new), axis=-1)
print(y_pred)
print(np.array(class_names)[y_pred])
```

```
[9 2 1]
['Ankle boot' 'Pullover' 'Trouser']
```

In [22]:
```
# Verify
y_new = y_test[:3]
y_new
```

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

In [23]:
```
plt.figure(figsize=(7.2, 2.4))
for index, image in enumerate(X_new):
    plt.subplot(1, 3, index + 1)
    plt.imshow(image, cmap="binary", interpolation="nearest")
    plt.axis('off')
    plt.title(class_names[y_test[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```



Exercise:

Suppose you have an MLP composed of one input layer with 10 neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.

- What is the shape of the input matrix X?
- What are the shapes of the hidden layer's weight vector Wh and its bias vector bh?
- What are the shapes of the output layer's weight vector Wo and its bias vector bo?
- What is the shape of the network's output matrix Y?
- Write the equation that computes the network's output matrix Y as a function of X, Wh, bh, Wo, and bo.

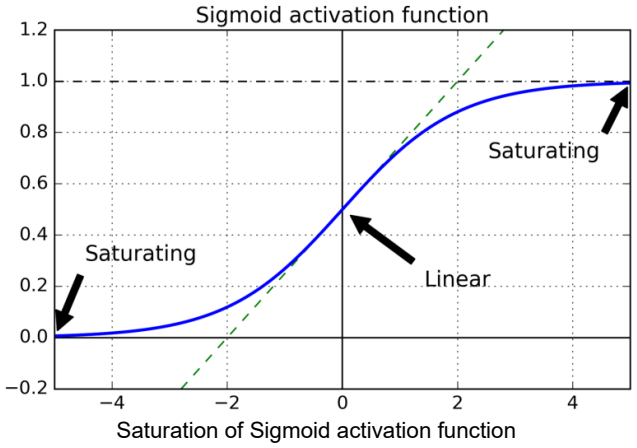
- **Solution**:
- X is m x 10, m = batch size
- Wh is 10 x 50, len(bh) is 50
- Wo is 50 x 3, len(bo) is 3
- Y is m x 3
- $Y = RELU(RELU(X \times Wh + bh) \times Wo + bo)$

Training Deep Neural Networks

- **Potential problems**:
- You may be faced with the tricky vanishing gradients problem or the related exploding gradients problem. This is when the gradients grow smaller and smaller, or larger and larger, when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.
- You might not have enough training data for such a large network, or it might be too costly to label.
- Training may be extremely slow.
- A model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances or if they are too noisy.

Vanishing/Exploding Gradients Problems

- The Gradient Descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution. We call this the vanishing gradients problem.
- In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is the exploding gradients problem, which surfaces in recurrent neural networks.



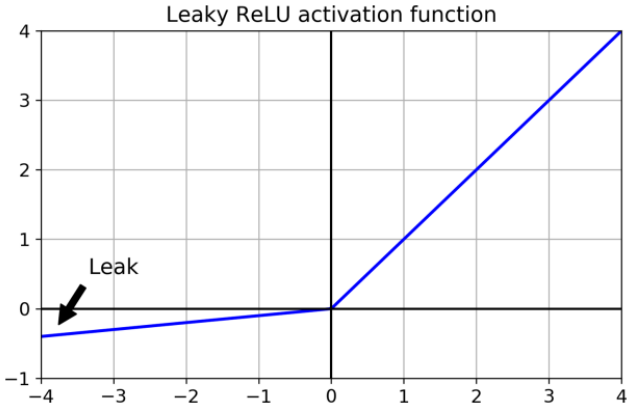
Saturation of Sigmoid activation function

- **Leaky ReLU**

$$LR(x) = \max(\alpha x, x)$$

- The hyperparameter α defines how much the function "leaks": it is the slope of the function for $x < 0$ and is typically set to 0.01.

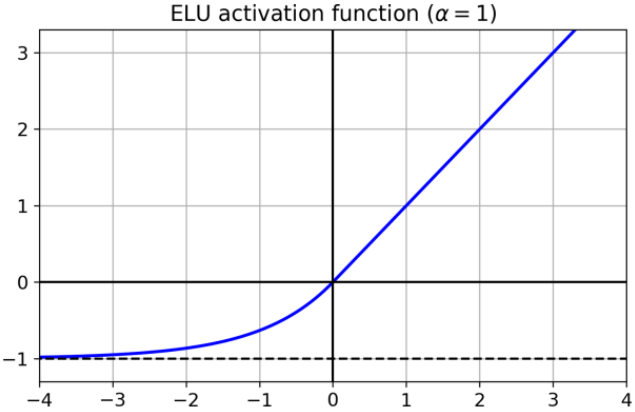
In [ ]:
```
def leaky_relu(z, alpha=0.01):
    return np.maximum(alpha*z, z)
```



Leaky ReLU activation function

- **Exponential linear unit (ELU)**

$$ELU(x) = \alpha(\exp(x) - 1), \text{if } x < 0, \text{ else } x$$



ELU activation function

Batch Normalization

- It consists of adding an operation in the model just before or after the activation function of each hidden layer.
- This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting.
- In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs.
- In order to zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation.
- It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name "Batch Normalization").

$$1.\quad \boldsymbol{\mu}_B = \frac{1}{m_B}\sum_{i=1}^{m_B}\mathbf{x}^{(i)}$$

$$2.\quad \boldsymbol{\sigma}_B^{\,2} = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$$

$$3.\quad \widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^{\,2} + \varepsilon}}$$

$$4.\quad \mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \widehat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

Batch Normalization

- $\boldsymbol{\mu}_B$ is the vector of input means, evaluated over the whole mini-batch B (it contains one mean per input).
- $\boldsymbol{\sigma}_B$ is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- m_B is the number of instances in the mini-batch.
- $\widehat{\mathbf{x}}^{(i)}$ is the vector of zero-centered and normalized inputs for instance i.

- $\boldsymbol{\gamma}$ is the output scale parameter vector for the layer (it contains one scale parameter per input).
- $\otimes$ represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- $\boldsymbol{\beta}$ is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
- ε is a tiny number that avoids division by zero (typically 10^{-5}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation. It is a rescaled and shifted version of the inputs.

In [24]:
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

In [25]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 784)               3136
_________________________________________________________________
dense_3 (Dense)              (None, 300)               235500
_________________________________________________________________
batch_normalization_1 (Batch (None, 300)               1200
_________________________________________________________________
dense_4 (Dense)              (None, 100)               30100
_________________________________________________________________
batch_normalization_2 (Batch (None, 100)               400
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
_________________________________________________________________
```

- Each BN layer adds four parameters per input: $\gamma, \beta, \mu, \sigma$ (for example, the first BN layer adds 3,136 parameters, which is 4 × 784).
- The last two parameters, μ and σ, are the moving averages; they are not affected by backpropagation, so Keras calls them “non-trainable”
- total number of BN parameters = 3,136 + 1,200 + 400; divide by 2, to get 2,368, which is the total number of non-trainable parameters in this model.

In [26]:
```
bn1 = model.layers[1]
[(var.name, var.trainable) for var in bn1.variables]
```

Out[26]:
```
[('batch_normalization/gamma:0', True),
 ('batch_normalization/beta:0', True),
 ('batch_normalization/moving_mean:0', False),
 ('batch_normalization/moving_variance:0', False)]
```

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.

- All Keras optimizers accept clipnorm or clipvalue arguments:

In [27]:
```
optimizer = keras.optimizers.SGD(clipvalue = 1.0)
# or
optimizer = keras.optimizers.SGD(clipnorm = 1.0)
```

- This optimizer will clip every component of the gradient vector to a value between –1.0 and 1.0.
- This means that all the partial derivatives of the loss (with regard to each and every trainable parameter) will be clipped between –1.0 and 1.0.

Faster Optimizers

Momentum Optimization

- Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the very simple idea behind momentum optimization.
- In contrast, regular Gradient Descent will simply take small, regular steps down the slope, so the algorithm will take much more time to reach the bottom.

- **GD**:

$$\theta \leftarrow \theta - \eta \nabla J(\theta)$$

- **Momentum**:

$$m \leftarrow \beta m - \eta \nabla J(\theta);$$
$$\theta \leftarrow \theta + m$$

- To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β, called the momentum, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.
- Gradient Descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, momentum optimization will roll down the valley faster and faster until it reaches the bottom (the optimum).

In [28]:
```
optimizer = keras.optimizers.SGD(learning_rate = 0.001, momentum = 0.9)
```

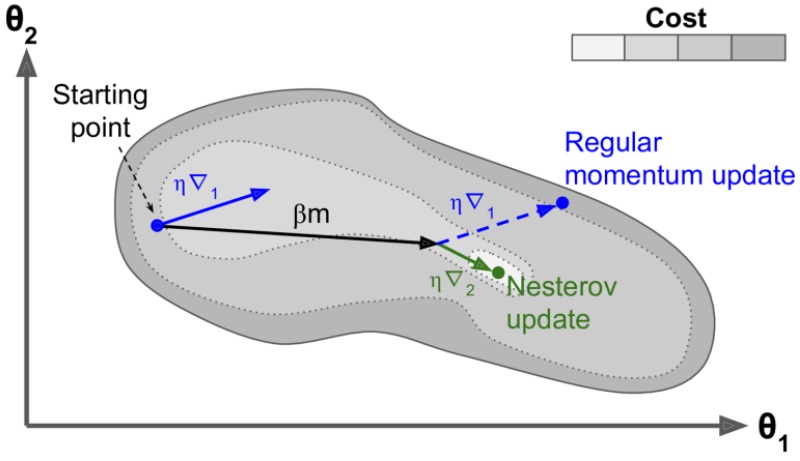
- The one drawback of momentum optimization is that it adds yet another hyperparameter to tune.

Nesterov Accelerated Gradient

- The Nesterov Accelerated Gradient (NAG) method, also known as Nesterov momentum optimization, measures the gradient of the cost function not at the local position θ but slightly ahead in the direction of the momentum, at $\theta + \beta m$

$$m \leftarrow \beta m - \eta \nabla J(\theta + \beta m);$$
$$\theta \leftarrow \theta + m$$

- This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than the gradient at the original position.



Regular versus Nesterov momentum optimization

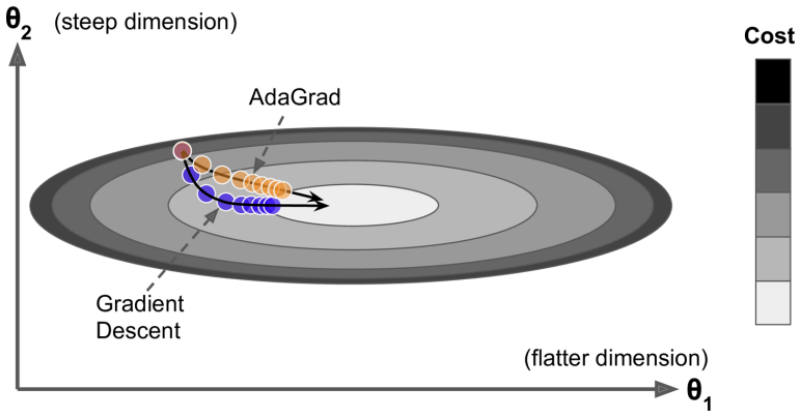
In [29]:
```
optimizer = keras.optimizers.SGD(learning_rate = 0.001,
                                 momentum = 0.9,
                                 nesterov = True)
```

AdaGrad

- Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley.
- It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum.
- The AdaGrad algorithm achieves this correction by scaling down the gradient vector along the steepest dimensions.

$$s \leftarrow s + \nabla J(\theta) \otimes \nabla J(\theta);$$
$$\theta \leftarrow \theta - \eta \nabla J(\theta) \oslash \sqrt{s + \epsilon}$$

- The first step accumulates the square of the gradients into the vector s (recall that the $\otimes$ symbol represents the element-wise multiplication).
- The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{s + \epsilon}$ (the $\oslash$ symbol represents the element-wise division, and ε is a smoothing term to avoid division by zero, typically set to 10^{-10}).
- This algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an adaptive learning rate. It helps point the resulting updates more directly toward the global optimum.



Adagrad

In [30]:
```
optimizer = keras.optimizers.Adagrad(learning_rate = 0.001)
```

RMSProp

- AdaGrad runs the risk of slowing down a bit too fast and never converging to the global optimum.
- The RMSProp algorithm fixes this by accumulating only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training). It does so by using exponential decay in the first step:

$$s \leftarrow \beta s + (1 - \beta) \nabla J(\theta) \otimes \nabla J(\theta)\ \theta \leftarrow \theta - \eta \nabla J(\theta) \oslash \sqrt{s + \epsilon}$$

- The decay rate β is typically set to 0.9.

In [31]:
```
optimizer = keras.optimizers.RMSprop(learning_rate = 0.001, rho = 0.9)
```

Adam Optimization

- Adam, which stands for adaptive moment estimation, combines the ideas of momentum optimization and RMSProp: just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients:

$$
\begin{aligned}
m &\leftarrow \beta_1 m - (1 - \beta_1)\nabla J(\theta) \\
s &\leftarrow \beta_2 s + (1 - \beta_2)\nabla J(\theta) \otimes \nabla J(\theta) \\
\hat{m} &\leftarrow \frac{m}{1-\beta_1^t} \\
\hat{s} &\leftarrow \frac{s}{1-\beta_2^t} \\
\theta &\leftarrow \theta + \eta \hat{m} \oslash \sqrt{\hat{s} + \epsilon}
\end{aligned}
$$

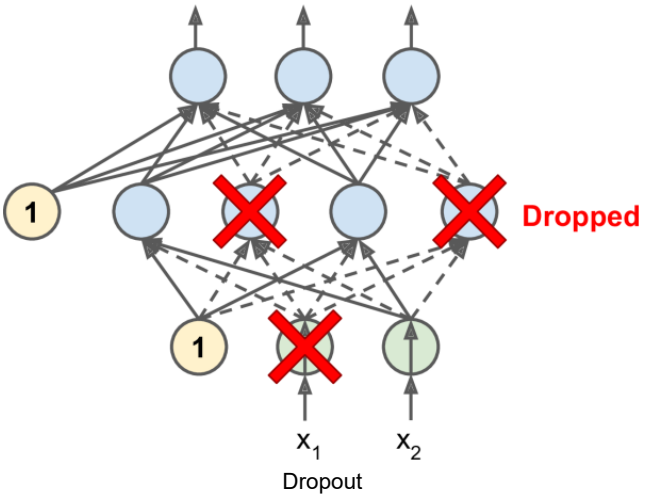
- t represents the iteration number (starting at 1)
- Step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $(1 - \beta_1)$ times the decaying sum).
- In steps 3 and 4, since m and s are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost m and s at the beginning of training.
- The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999.

In [32]:
```
optimizer = keras.optimizers.Adam(learning_rate = 0.001,
                                  beta_1 = 0.9,
                                  beta_2 = 0.999)
```

- Since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter η.
- We can often use the default value η = 0.001, making Adam even easier to use than Gradient Descent.

DropOut

- Dropout is one of the most popular regularization techniques for deep neural networks.
- 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 between 10% and 50%.



Dropout

In [42]:
```
# Without Dropout
model_nd = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation = "elu", kernel_initializer = "he_normal"),
    keras.layers.Dense(100, activation = "elu", kernel_initializer = "he_normal"),
    keras.layers.Dense(10, activation  = "softmax")
])
model_nd.compile(loss    = "sparse_categorical_crossentropy",
              optimizer = "adam",
              metrics   = ["accuracy"])
n_epochs = 20
history_nd = model_nd.fit(X_train, y_train, epochs=n_epochs,
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.4810 - accuracy: 0.8246 - val_l
oss: 0.4095 - val_accuracy: 0.8492
Epoch 2/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3694 - accuracy: 0.8643 - val_l
oss: 0.3568 - val_accuracy: 0.8694
Epoch 3/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3297 - accuracy: 0.8771 - val_l
oss: 0.3285 - val_accuracy: 0.8846
Epoch 4/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3059 - accuracy: 0.8859 - val_l
oss: 0.3258 - val_accuracy: 0.8842
Epoch 5/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2887 - accuracy: 0.8911 - val_l
oss: 0.3282 - val_accuracy: 0.8826
Epoch 6/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2713 - accuracy: 0.8973 - val_l
oss: 0.3060 - val_accuracy: 0.8890
Epoch 7/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2595 - accuracy: 0.9025 - val_l
oss: 0.3181 - val_accuracy: 0.8854
Epoch 8/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2434 - accuracy: 0.9070 - val_l
oss: 0.3293 - val_accuracy: 0.8918
Epoch 9/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2353 - accuracy: 0.9105 - val_l
oss: 0.3228 - val_accuracy: 0.8942
Epoch 10/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2232 - accuracy: 0.9138 - val_l
oss: 0.3191 - val_accuracy: 0.8882
Epoch 11/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2124 - accuracy: 0.9187 - val_l
oss: 0.3417 - val_accuracy: 0.8850
Epoch 12/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2045 - accuracy: 0.9212 - val_l
oss: 0.3356 - val_accuracy: 0.8898
Epoch 13/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1970 - accuracy: 0.9250 - val_l
oss: 0.3193 - val_accuracy: 0.8992
Epoch 14/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1883 - accuracy: 0.9279 - val_l
oss: 0.3238 - val_accuracy: 0.8934
Epoch 15/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1817 - accuracy: 0.9302 - val_l
oss: 0.3343 - val_accuracy: 0.9010
Epoch 16/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1723 - accuracy: 0.9347 - val_l
oss: 0.3386 - val_accuracy: 0.8944
Epoch 17/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1691 - accuracy: 0.9335 - val_l
oss: 0.3519 - val_accuracy: 0.8914
Epoch 18/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1609 - accuracy: 0.9381 - val_l
oss: 0.3513 - val_accuracy: 0.8952
Epoch 19/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1571 - accuracy: 0.9401 - val_l
oss: 0.3694 - val_accuracy: 0.8940
Epoch 20/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.1521 - accuracy: 0.9417 - val_l
oss: 0.3946 - val_accuracy: 0.8952
```

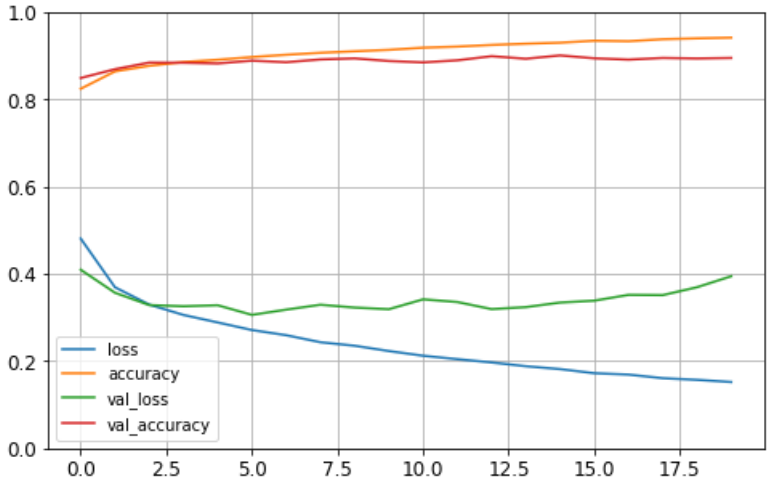
In [43]:
```
# Test the model
model_nd.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.4237 - accuracy: 0.8913
```

Out[43]:
```
[0.42374420166015625, 0.8913000226020813]
```

In [44]:
```
# Plots
import pandas as pd

pd.DataFrame(history_nd.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



In [45]:
```
# With Dropout and Batch Normalization
model_do = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(rate = 0.2),
    keras.layers.Dense(300, activation = "elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(rate = 0.2),
    keras.layers.Dense(100, activation = "elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(rate = 0.2),
    keras.layers.Dense(10, activation  = "softmax")
])
model_do.compile(loss      = "sparse_categorical_crossentropy",
                 optimizer = "adam",
                 metrics   = ["accuracy"])
n_epochs = 20
history_do = model_do.fit(X_train, y_train, epochs=n_epochs,
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/20
1719/1719 [==============================] - 7s 4ms/step - loss: 0.5571 - accuracy: 0.8006 - val_l
oss: 0.3871 - val_accuracy: 0.8594
Epoch 2/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.4391 - accuracy: 0.8399 - val_l
oss: 0.3529 - val_accuracy: 0.8732
Epoch 3/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.4041 - accuracy: 0.8529 - val_l
oss: 0.3185 - val_accuracy: 0.8804
Epoch 4/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.3826 - accuracy: 0.8604 - val_l
oss: 0.3137 - val_accuracy: 0.8788
Epoch 5/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.3646 - accuracy: 0.8670 - val_l
oss: 0.3042 - val_accuracy: 0.8830
Epoch 6/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.3505 - accuracy: 0.8712 - val_l
oss: 0.2910 - val_accuracy: 0.8898
Epoch 7/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.3408 - accuracy: 0.8726 - val_l
oss: 0.2820 - val_accuracy: 0.8936
Epoch 8/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.3331 - accuracy: 0.8750 - val_l
oss: 0.2825 - val_accuracy: 0.8950
Epoch 9/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.3221 - accuracy: 0.8795 - val_l
oss: 0.2802 - val_accuracy: 0.8968
Epoch 10/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.3153 - accuracy: 0.8824 - val_l
oss: 0.2736 - val_accuracy: 0.8956
Epoch 11/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.3113 - accuracy: 0.8848 - val_l
oss: 0.2803 - val_accuracy: 0.8902
Epoch 12/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.3052 - accuracy: 0.8866 - val_l
oss: 0.2706 - val_accuracy: 0.9004
Epoch 13/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.2963 - accuracy: 0.8888 - val_l
oss: 0.2722 - val_accuracy: 0.8962
Epoch 14/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.2909 - accuracy: 0.8918 - val_l
oss: 0.2654 - val_accuracy: 0.8978
Epoch 15/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.2904 - accuracy: 0.8922 - val_l
oss: 0.2614 - val_accuracy: 0.8974
Epoch 16/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.2847 - accuracy: 0.8929 - val_l
oss: 0.2667 - val_accuracy: 0.8980
Epoch 17/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.2791 - accuracy: 0.8959 - val_l
oss: 0.2669 - val_accuracy: 0.9000
Epoch 18/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.2794 - accuracy: 0.8944 - val_l
oss: 0.2608 - val_accuracy: 0.9002
Epoch 19/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.2750 - accuracy: 0.8978 - val_l
oss: 0.2650 - val_accuracy: 0.9020
Epoch 20/20
1719/1719 [==============================] - 6s 4ms/step - loss: 0.2722 - accuracy: 0.8973 - val_l
oss: 0.2656 - val_accuracy: 0.8986
```

In [46]:
```
# Test the model
model_do.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.2969 - accuracy: 0.8955
```

Out[46]:
```
[0.2969399094581604, 0.8955000042915344]
```

In [47]:
```
# Plots
import pandas as pd

pd.DataFrame(history_do.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```