3.
Implementing MLPs with Keras

Keras

- Tensorflow is a low-level software library created by Google to implement machine learning models.
- Keras is TensorFlow's high-level deep learning API.
  - It allows you to build, train, evaluate, and execute all sorts of neural networks.
- The original Keras library was developed by François Chollet as part of a research project.
- There are other popular deep learning libraries including PyTorch by Facebook and JAX by Google.

Building an Image Classifier

- We use Fashion MNIST, which has the exact same format as MNIST (70,000 grayscale images of 28×28 pixels each, with 10 classes), but the images represent fashion items rather than handwritten digits.
  - Each class is more diverse, and the problem turns out to be significantly more challenging than MNIST.
- We can use Keras to fetch and load common datasets:

```
import tensorflow as tf

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

Fashion MNIST Dataset

- When loading MNIST or Fashion MNIST using Keras rather than Scikit-Learn, every image is represented as a 28 × 28 array rather than a 1D array of size 784:

```
X_train.shape
```

```
(55000, 28, 28)
```

- The pixel intensities are represented as integers (from 0 to 255) rather than floats (from 0.0 to 255.0):

```
X_train.dtype
```

```
dtype('uint8')
```

- We scale the pixel intensities down to the 0–1 range by dividing them by 255.0 (this also converts them to floats):

```
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.
```

Fashion MNIST Dataset

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

```
class_names[y_train[0]]
```

```
'Ankle boot'
```

Creating the Model using the Sequential API

- A classification MLP with two hidden layers:
- `tf.keras.Sequential` creates a `Sequential` model.
  - This is the simplest kind of Keras model for neural networks that are composed of a single stack of layers connected sequentially.

```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(input_shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Creating the Model using the Sequential API

- We build the first layer (an `Input` layer) and add it to the model.
- We specify the input shape, which doesn't include the batch size, only the shape of the instances.
- Keras needs to know the shape of the inputs so it can determine the shape of the connection weight matrix of the first hidden layer.

```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(input_shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Creating the Model using the Sequential API

- Then we add a `Flatten` layer to convert each input image into a 1D array: for example, if it receives a batch of shape [32, 28, 28], it will reshape it to [32, 784].
  - i.e. if it receives input data `X`, it computes `X.reshape(-1,784)`.
  - This layer doesn't have any parameters; it's just there to do some simple preprocessing.

```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(input_shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Creating the Model using the Sequential API

- Next we add a `Dense` hidden layer with 300 neurons.
  - It will use the ReLU activation function.
  - Each Dense layer manages its own weight matrix $\mathbf{W}$, containing all the connection weights between the neurons and their inputs.
  - It also manages a vector of bias terms $\mathbf{b}$ (one per neuron).
  - When it receives input data $\mathbf{X}$, it computes $h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$

```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(input_shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Creating the Model using the Sequential API

- Finally, a `Dense` output layer with 10 neurons (one per class), using the softmax activation function because the classes are exclusive.

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

- Instead of adding the layers one by one, it's often more convenient to pass a list of layers when creating the Sequential model.
  - You can also drop the Input layer and instead specify the `input_shape` in the first layer.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

Model 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
_________________________________________________________________
```

Layer Names

- Each layer in a model must have a unique name (e.g., "`dense_2`").
- You can set the layer names explicitly using the constructor's `name` argument, but it's simpler to let Keras name the layers automatically.
  - Keras takes the layer's class name and converts it to snake case.
- Keras also ensures that the name is globally unique, even across models, by appending an index if needed, as in "`dense_2`".
- Why making the names unique across models is important?
  - This makes it possible to merge models without getting name conflicts.
- All global state managed by Keras is stored in a *Keras session*, which you can clear using `tf.keras.backend.clear_session()`.
  - In particular, this resets the name counters.

Model Layers

- You can get a model's list of layers using the `layers` attribute, or use the `get_layer()` method to access a layer by name:

```
model.layers
```

```
[<keras.layers.core.flatten.Flatten at 0x7fb220f4d430>,
 <keras.layers.core.dense.Dense at 0x7fb282285af0>,
 <keras.layers.core.dense.Dense at 0x7fb282365b50>,
 <keras.layers.core.dense.Dense at 0x7fb282365fa0>]
```

```
hidden1 = model.layers[1]
hidden1.name
```

```
'dense'
```

```
model.get_layer('dense') is hidden1
```

```
True
```

Model Parameters

- All the parameters of a layer can be accessed using its `get_weights`() and `set_weights`() methods.
- For a `Dense` layer, this includes both the connection weights and the bias terms:

```
weights, biases = hidden1.get_weights()
weights
```

```
array([[ 0.01932564,  0.02057646,  0.07405128, ...,  0.0649996 ,
         0.05198881,  0.05619334],
       [ 0.06790733,  0.03334583,  0.02839814, ...,  0.02439063,
         0.06042613,  0.05941309],
       ...,
       [ 0.07028989, -0.06171814, -0.03369412, ...,  0.02316521,
         0.06790586, -0.01861121]], dtype=float32)
```

```
biases
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
      ..., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

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:

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

- Using the SGD optimizer, it is important to tune the learning rate.
  - Use `optimizer=tf.keras.optimizers.SGD(learning_rate=__???__)` to set the learning rate, rather than `optimizer="sgd"`, which defaults to a learning rate of 0.01.

```
model.compile(loss=tf.keras.losses.sparse_categorical_crossentropy,
              optimizer=tf.keras.optimizers.SGD(),
              metrics=[tf.keras.metrics.sparse_categorical_accuracy])
```

Choosing the Loss Function

- We use the "`sparse_categorical_crossentropy`" loss because we have sparse labels and the classes are exclusive.
  - To convert sparse labels (i.e., class indices) to one-hot vector labels, use the `tf.keras.utils.to_categorical`() function.
- If we had one target probability per class for each instance, then we would need to use the "`categorical_crossentropy`" loss instead.
- If we were dealing with binary classification or multilabel binary classification, then we would use the "`sigmoid`" activation function in the output layer instead of the "`softmax`" activation function, and we would use the "`binary_crossentropy`" loss.

Training and Evaluating the Model

- The default value for number of epochs is 1.
- Passing a validation set is optional but it is very useful to see how well the model really performs.
- Instead of passing a validation set, set `validation_split` to the ratio of the training set that you want to use for validation.

```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/30
1719/1719 [==============================] - 30s 16ms/step -
loss: 0.7359 - accuracy: 0.7581 - val_loss: 0.5060 - val_acc
uracy: 0.8268
Epoch 2/30
1719/1719 [==============================] - 27s 16ms/step -
loss: 0.4876 - accuracy: 0.8312 - val_loss: 0.4575 - val_acc
uracy: 0.8334
[...]
Epoch 30/30
1719/1719 [==============================] - 13s 8ms/step
- loss: 0.2247 - accuracy: 0.9203 - val_loss: 0.3025 - val
_accuracy: 0.8880
```

Error Analysis

- At each epoch during training, Keras displays the number of mini-batches processed so far on the left side of the progress bar.
- The batch size is 32 by default, and since the training set has 55,000 images, the model goes through 1,719 batches per epoch: 1,718 of size 32, and 1 of size 24.
- After the progress bar, you can see the mean training time per sample, and the loss and accuracy (or any other extra metrics you asked for) on both the training set and the validation set.

```
Epoch 30/30
1719/1719 [==============================] - 13s 8ms/step
- loss: 0.2247 - accuracy: 0.9203 - val_loss: 0.3025 - val
_accuracy: 0.8880
```

Skewed Training Set

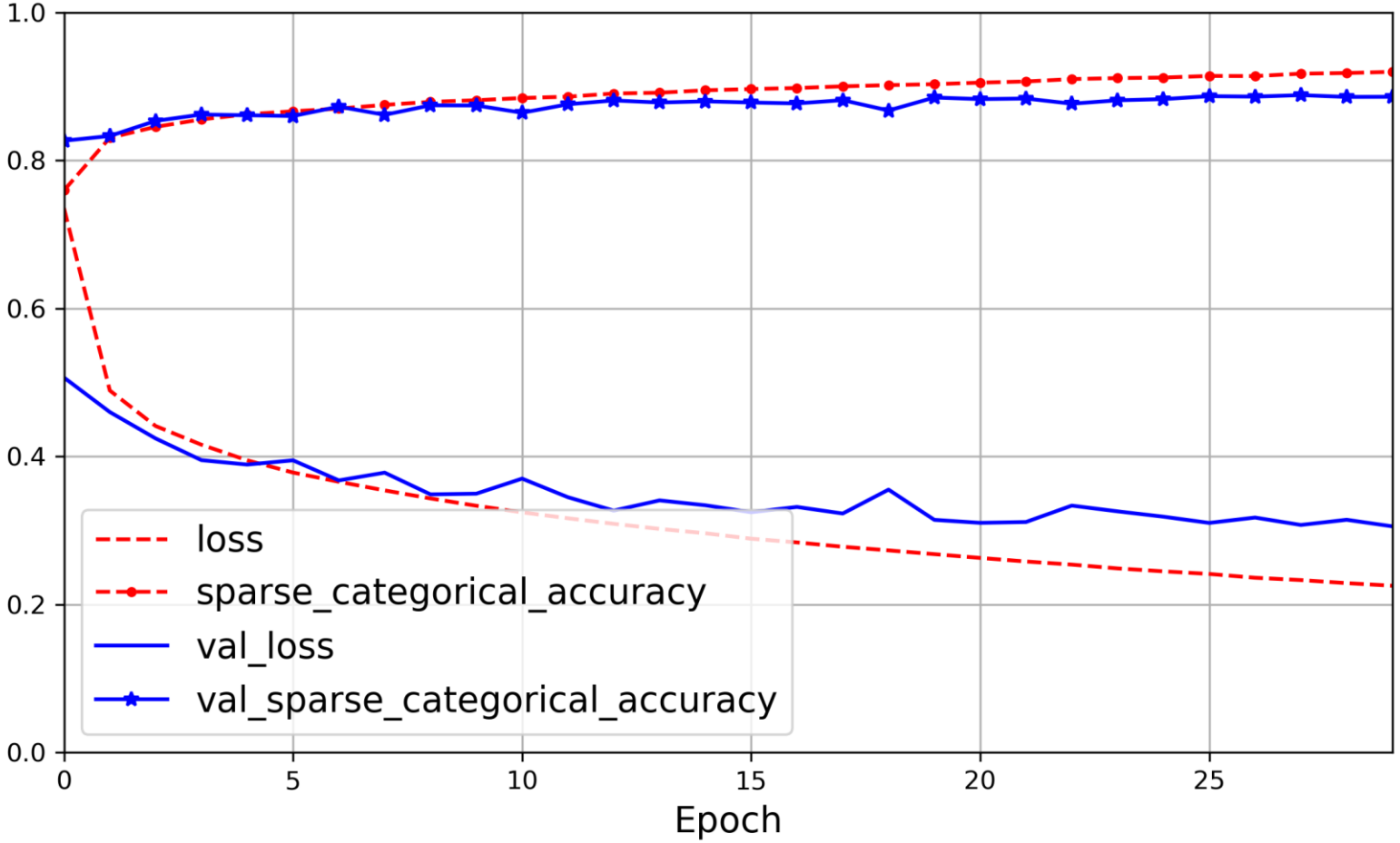
- If the training set was skewed (imbalanced), set the `class_weight` argument when calling the `fit()` method, to give a larger weight to underrepresented classes and a lower weight to overrepresented classes.
  - These weights would be used by Keras when computing the loss.
  - If you need per-instance weights, set the `sample_weight` argument.
  - If both `class_weight` and `sample_weight` are provided, then Keras multiplies them.
- Per-instance weights could be useful, e.g. if some instances were labeled by experts (more weight) while others were labeled using a crowdsourcing platform.

History

- The `fit()` method returns a `History` object containing:
  - `history.params`: the training parameters
  - `history.epoch`: the list of epochs it went through
  - `history.history`: a dictionary containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any).
    - If you use this dictionary to create a Pandas DataFrame and call its `plot()` method, you get the learning curves.

```
import matplotlib.pyplot as plt
import pandas as pd

pd.DataFrame(history.history).plot(
    figsize=(8, 5), xlim=[0, 29], ylim=[0, 1], grid=True, xlabel="Epoch",
    style=["r--", "r--.", "b-", "b-*"])
plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25
Epoch
loss
sparse_categorical_accuracy
val_loss
val_sparse_categorical_accuracy

Poor Performance

- If you are not satisfied with the performance of your model, you should go back and tune the hyperparameters.
- The first one to check is the learning rate.
  - If that doesn't help, try another optimizer and always retune the learning rate after changing any hyperparameter.
- Next try tuning model hyperparameters such as the number of layers, the number of neurons per layer, and the types of activation functions to use for each hidden layer.
- You can try tuning other hyperparameters, such as the batch size.
  - It can be set in the `fit`() method using the `batch_size` argument, which defaults to 32.

Generalization Error

- Once you are satisfied with your model's validation accuracy, you should evaluate it on the test set to estimate the generalization error before you deploy the model to production.
- You can do this using the `evaluate`() method.

```
model.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 2s 6ms/step - loss:
0.3275 - sparse_categorical_accuracy: 0.8851
```

- It is common to get slightly lower performance on the test set than on the validation set, because the hyperparameters are tuned on the validation set, not the test set.

Making Predictions

- Use the model's `predict()` method to make predictions on new instances.

```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
```

```
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.02, 0.  , 0.97],
       [0.  , 0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

- For each instance the model estimates one probability per class, from class 0 to class 9.
  - This is similar to the output of the `predict_proba()` method in Scikit-Learn classifiers.

Making Predictions

- If you only care about the class with the highest estimated probability (even if that probability is quite low), then you can use the `argmax()` method to get the highest probability class index for each instance:

```
y_pred = y_proba.argmax(axis=-1)
y_pred
```

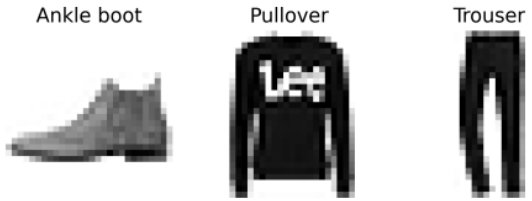
```
array([9, 2, 1], dtype=int64)
```

```
np.array(class_names)[y_pred]
```

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

```
y_new = y_test[:3]
y_new
```

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



Review: Regression MLPs in Scikit-Learn

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False)
```

Regression MLP using Sequential API

- We don't need a *Flatten* layer, instead we're using a *Normalization* layer as the first layer.
  - Similar to Scikit-Learn's StandardScaler, but must be fitted to the training data using the `adapt`() method *before* you call the model's `fit`() method.
- The output layer has a single neuron (since we only want to predict a single value) and it uses no activation function.

```
tf.random.set_seed(42)
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(1)
])
```

Regression MLP using Sequential API

- The loss function is the mean squared error, the metric is the RMSE.
- We're using an Adam optimizer like Scikit-Learn's MLPRegressor did.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
norm_layer.adapt(X_train)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

```
rmse_test
```

```
0.5297096967697144
```

```
y_pred
```

```
array([[0.4969182],
       [1.195265 ],
       [4.9428763]], dtype=float32)
```