In [1]:
```
# enable high-res images in notebook
%config InlineBackend.figure_format = 'retina'
```

15. Deep Learning

Objectives (1 of 2)

- What a **neural network** is and how it enables **deep learning**
- Create **Keras neural networks**
- Keras **layers**, **activation functions**, **loss functions** and **optimizers**
- Use a Keras **convolutional neural network (CNN)** trained on the **MNIST dataset** to **recognize handwritten digits**
- Use **TensorBoard** to **visualize** training progress
- Use a Keras **recurrent neural network (RNN)** trained on the **IMDb dataset** to perform **binary classification** of **positive and negative movie reviews**

===DITCH LAST TWO FROM OBJECTIVES EVEN IF WE KEEP THE CONTENT===

- List Keras **pretrained neural networks**
- Understand the value of Keras **pretrained neural networks** that were trained on the massive **ImageNet dataset** for **computer vision apps**

15.1 Introduction

- **Deep learning**—powerful subset of **machine learning**
- Has produced impressive results in **computer vision** and many other areas
- **Resource-intensive deep-learning solutions** are possible due to
  - **big data**
  - **significant processor power**
  - **faster Internet speeds**
  - advancements in **parallel computing hardware and software**

Keras and TensorFlow ===THIN THIS===

- **Keras** offers a friendly interface to Google's **TensorFlow**—the most widely used deep-learning library
  - Also Microsoft's **CNTK** and the Université de Montréal's **Theano** (ceased development in 2017)
- **François Chollet** of the **Google Mind team** developed **Keras** to make deep-learning capabilities **more accessible.**
  - His book ***Deep Learning with Python*** (https://amzn.to/303gknb) is a must read.
- **Google has thousands of TensorFlow and Keras projects** underway internally and that number is growing quickly.[1] (http://theweek.com/speedreads/654463/google-more-than-1000-artificial-intelligence-projects-works), [2] (https://www.zdnet.com/article/google-says-exponential-growth-of-ai-is-changing-nature-of-compute/)
- **Questions on Keras?** Visit the **Keras team's slack channel** (https://kerasteam.slack.com) for answers

Models

- **Deep learning models** connect multiple **layers**
- Models **encapsulate sophisticated mathematics**
  - You need only define, parameterize and manipulate objects
  - Understanding model internals requires extensive math background
  - We'll **avoid heavy mathematics** in favor of English explanations
- Keras facilitates **experimenting** with **many models**
  - Tweak until you find the one that performs best
- In general, **more data** leads to a **better trained deep learning model**

Processing Power

- **Deep learning** can require **significant processing power**
- Training models on **big-data** can take **hours**, **days** or **more**
  - Our examples can be **trained in minutes to just less than an hour** on **conventional CPUs**
- High-performance **GPUs (Graphics Processing Units)** and **TPUs (Tensor Processing Units)** developed by **NVIDIA** and **Google** are typically used to meet the extraordinary processing demands of deep-learning applications

Future of Deep Learning

- Newer **automated deep learning capabilities** are making it even easier to build deep-learning solutions.
  - **Auto-Keras** (https://autokeras.com/) from Texas A&M University's DATA Lab
  - Baidu's **EZDL** (https://ai.baidu.com/ezdl/)
  - Google's **AutoML** (https://cloud.google.com/automl/)

15.1.1 Deep Learning Applications

Game playing	Computer vision: Object recognition, pattern recognition, facial recognition
Self-driving cars	Robotics
Improving customer experiences	Chatbots
Diagnosing medical conditions	Google Search
Facial recognition	Automated image captioning and video closed captioning
Enhancing image resolution	Speech recognition
Language translation	Predicting election results
Predicting earthquakes and weather	Google Sunroof to determine whether you can put solar panels on your roof
Generative applications	
Generating original images	Processing existing images to look like a specified artist's style
Adding color to black-and-white images and video	Creating music
Creating text (books, poetry)	Much more.

15.3 Custom Anaconda Environments

- We used **TensorFlows built-in version of Keras**
- TensorFlow requires **Python 3.6.x** (3.7 support coming soon)
- Easy to set up **custom environment** for Keras and TensorFlow
  - Helps with **reproducibility** if code depends on specific Python or library versions
  - Details in my **Python Fundamentals LiveLessons videos** (https://learning.oreilly.com/videos/python-fundamentals/9780135917411) (deep learning lesson coming soon) and in **Python for Programmers, Section 15.3** (https://learning.oreilly.com/library/view/Python+for+Programmers,+First+Edition/9780135231364/ch15.x
- Preconfigured **Docker**: **`jupyter/tensorflow-notebook`** (https://hub.docker.com/r/jupyter/tensorflow-notebook/)

Creating/Activating/Deactivating an Anaconda Environment

```
conda create -n tf_env python=3.6 anaconda tensorflow ipython jup
yterlab scikit-learn matplotlib seaborn h5py pydot graphviz nodej
s
```

- Computers with **Tensorflow-compatible NVIDIA GPUs**: Replace `tensorflow` with **`tensorflow-gpu`** for better performance (https://www.tensorflow.org/install/gpu)
  - Some **AMD GPUs** also support TensorFlow (http://timdettmers.com/2018/11/05/which-gpu-for-deep-learning/)

```
conda activate tf_env

conda deactivate
```

15.4 Neural Networks

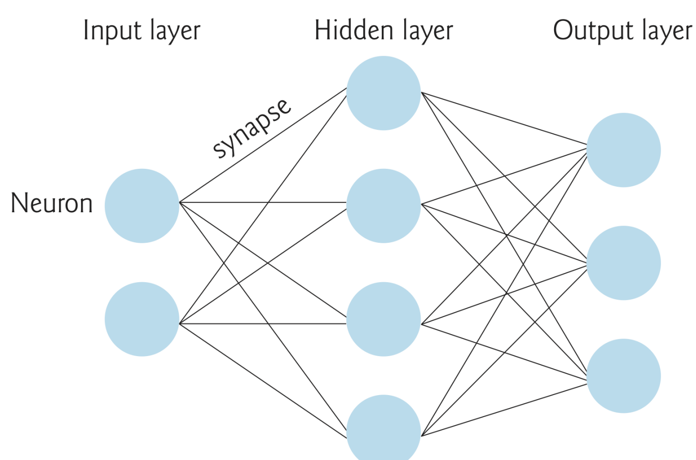
- Deep learning uses **artificial neural networks** to learn
- Similar to how scientists believe our **brains** work
- **Biological nervous systems** are controlled via **neurons** (https://en.wikipedia.org/wiki/Neuron) that communicate with one another along pathways called **synapses** (https://en.wikipedia.org/wiki/Synapse)
- **As we learn**, the **specific neurons** for a given task, like walking, **communicate with one another more efficiently**
- Neurons for a given task **activate** (https://www.sciencenewsforstudents.org/article/learning-rewires-brain) when we need to perform that task

Artificial Neurons

- In a neural network, interconnected **artificial neurons** simulate the human brain's neurons to help the network learn
- The **connections** between specific neurons are **reinforced during the learning process** with the goal of achieving a specific result

Artificial Neural Network Diagram

- The following diagram shows a three-**layer** neural network.
- **Circles** represent **neurons**, **lines** between them simulate **synapses**
- Output from a neuron becomes input to another neuron
- Diagram of a **fully connected network**—every neuron in a given layer is connected to **all** the neurons in the next layer:



Learning Is an Iterative Process (1 of 2)

- When you were a baby, you did not learn to walk instantaneously
- You learned that process over time with repetition
- You built up the smaller components of the movements that enabled you to walk—learning to stand, learning to balance to remain standing, learning to lift your foot and move it forward, etc.
- And you got feedback from your environment
- When you walked successfully your parents smiled and clapped
- When you fell, you might have bumped your head and felt pain

Learning Is an Iterative Process (2 of 2)

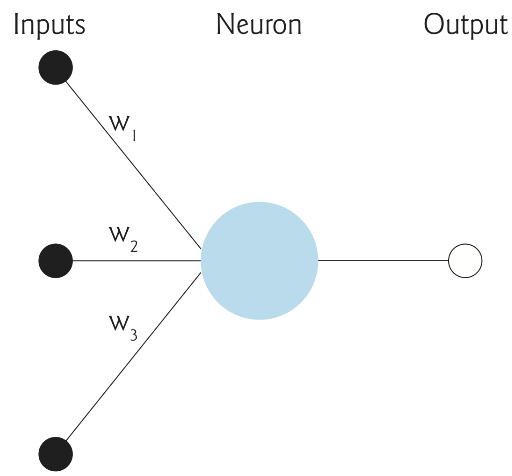
- We train neural networks iteratively over time
- Each iteration is an **epoch** and processes **every training dataset sample** once
- There's no "correct" number of epochs—a **hyperparameter** that may need tuning, based on your training data and your model
- The **inputs to the network** are the **features** in the **training samples**
- **Some layers learn new features** from **previous layers' outputs** and **others interpret those features** to make **predictions**

How Artificial Neurons Decide Whether to Activate Synapses (1 of 3)

- During **training**, the network calculates **weights** for every **connection** between the **neurons in one layer** and **those in the next**
- On a **neuron-by-neuron basis**, each of its **inputs** is **multiplied by** that **connection's weight**—**sum** of those weighted inputs is passed to the neuron's **activation function**
- **Activation function's output** determines **which neurons to activate** based on the **inputs**—just like neurons in your brain respond to inputs from your senses

How Artificial Neurons Decide Whether to Activate Synapses (2 of 3)

- Diagram of a **neuron** receiving three **inputs** (black dots) and producing an **output** (hollow circle) that would be passed to all or some of neurons in the next layer, depending on the types of the neural network's layers



- **w1**, **w2** and **w3** are **weights**
- In a **new model** that you train from scratch, these **values** are **initialized randomly** by the model

How Artificial Neurons Decide Whether to Activate Synapses (3 of 3)

- As the network **trains**, it tries to **minimize the error rate** between the **network's predicted labels** and the **samples' actual labels**
- The **error rate** is known as the **loss**, and the **calculation** that determines the **loss** is called the **loss function**
- **Backpropagation—Throughout training**, the network determines the **amount that each neuron contributes to the overall loss**, then goes back through the layers and **adjusts the weights** in an effort to **minimize that loss**
  - **Optimizing weights** occurs gradually

15.5 Tensors (1 of 5)

- Deep learning frameworks generally manipulate data in **tensors**, which they use to perform the mathematical calculations that enable neural networks to learn
- A tensor is basically a **multidimensional array**
- **Tensors** can **become quite large** as the **number of dimensions increases** and as the **richness** of the data increases (e.g., images, audios and videos are richer than text)

15.5 Tensors (2 of 5)

- **Chollet** discusses the types of tensors typically encountered in deep learning: **[Chollet, François. *Deep Learning with Python*. Section 2.2. Shelter Island, NY: Manning Publications, 2018.]**
  - **0D (0-dimensional) tensor**—This is **one value** and is known as a **scalar**.
  - **1D tensor**—This is similar to a **one-dimensional array** and is known as a **vector**. A 1D tensor might represent a sequence, such as hourly temperature readings from a sensor or the words of one movie review.
  - **2D tensor**—This is similar to a **two-dimensional array** and is known as a **matrix**. A 2D tensor could represent a **grayscale image** in which the tensor's two dimensions are the image's width and height in pixels, and the value in each element is the intensity of that pixel.

15.5 Tensors (3 of 5)

- Chollet discusses the types of tensors typically encountered in deep learning: **[Chollet, François. *Deep Learning with Python*. Section 2.2. Shelter Island, NY: Manning Publications, 2018.]**
  - **3D tensor**—This is similar to a **three-dimensional array** and could be used to represent a **color image**. The first two dimensions would represent the width and height of the image in pixels and the **depth** at each location might represent the red, green and blue (RGB) components of a given pixel's color. A 3D tensor also could represent a **collection** of 2D tensors containing grayscale images.
  - **4D tensor**—A 4D tensor could be used to represent a **collection of color images in 3D tensors**. It also could be used to represent **one video**. Each frame in a video is essentially a color image.
  - **5D tensor**—This could be used to represent a **collection of 4D tensors containing videos**.

15.5 Tensors (4 of 5)

- Tensor **dimensionality**
  - Assume we're creating a deep-learning network to identify and track objects in 4K (high-resolution; 3840-by-2160 pixels) videos that have 30 frames-per-second
  - Assume RGB for pixel colors
- **Each frame**: **3D tensor** with **24,883,200** elements (3840 × 2160 × 3)
- **Each video**: **4D tensor** containing sequence of frames

15.5 Tensors (5 of 5)

- For a one minute video: **44,789,760,000** elements **per tensor**!
- [Over 600 hours of video are uploaded to YouTube every minute (https://www.inc.com/tom-popomaronis/youtube-analyzed-trillions-of-data-points-in-2018-revealing-5-eye-opening-behavioral-statistics.html)](https://www.inc.com/tom-popomaronis/youtube-analyzed-trillions-of-data-points-in-2018-revealing-5-eye-opening-behavioral-statistics.html)
  - In just **one minute of uploads**, Google could have a tensor containing **1,612,431,360,000,000** elements to use in training deep-learning models—that's **big data**
- Tensors can quickly become **enormous**, so manipulating them efficiently is crucial
- This is why most deep learning is performed on **GPUs** or Google's **TPUs (Tensor Processing Units)** that are **optimized for tensor manipulations**

High-Performance Processors (1 of 2)

===P says consider ditching these two slides===

- Powerful processors are needed for real-world deep learning because the size of tensors can be enormous and large-tensor operations can place crushing demands on processors.
- The processors most commonly used for deep learning are from NVIDIA and Google
- NVIDIA GPUs (Graphics Processing Units)
  - Originally developed for computer gaming, GPUs are much faster than conventional CPUs for processing large amounts of data, enabling developers to train, validate and test deep-learning models more efficiently—and thus experiment with more of them.
- Optimized for the mathematical matrix operations typically performed on tensors, an essential aspect of how deep learning works "under the hood."
- NVIDIA's **Volta Tensor Cores** are specifically designed for deep learning.[1] (https://www.nvidia.com/en-us/data-center/tensorcore/), [2] (https://devblogs.nvidia.com/tensor-core-ai-performance-milestones/)
- Many NVIDIA GPUs are compatible with TensorFlow (https://www.tensorflow.org/install/gpu), and hence Keras, and can enhance the performance of your deep-learning models.

High-Performance Processors (2 of 2)

- Google TPUs (Tensor Processing Units)
- Recognizing that deep learning is crucial to its future, Google developed **TPUs (Tensor Processing Units)** (https://cloud.google.com/tpu/), which they now use in their **Cloud TPU service**, which "can provide up to 11.5 petaflops of performance in a single pod" (that's 11.5 quadrillion floating-point operations per second).
- TPUs are designed to be especially **energy efficient**—a key concern for companies like Google with already **massive computing clusters that are growing exponentially** and consuming vast amounts of energy.

15.6 Convolutional Neural Networks for Vision; Multi-Classification with the MNIST Dataset (1 of 2)

- Previously, we classified 1797 8-by-8-pixel handwritten digits
- We'll now use **`MNIST` database of handwritten digits**
  - "The MNIST Database." MNIST Handwritten Digit Database, Yann LeCun, Corinna Cortes and Chris Burges. http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/).
- We'll create a **convolutional neural network** (https://en.wikipedia.org/wiki/Convolutional_neural_network) (also called a **convnet** or **CNN**)
- Common in **computer-vision applications**
  - Recognizing handwritten digits and characters
  - Recognizing objects in images and video
- **Non-vision applications**
  - natural-language processing
  - recommender systems

15.6 Convolutional Neural Networks for Vision; Multi-Classification with the MNIST Dataset (2 of 2)

- **60,000 labeled digit image samples** for **training**, **10,000** for testing
- **28-by-28 pixel images** (**784 features**), each represented as a **NumPy array**
- **Grayscale pixel intensity** (shade) values **0-255**
- **Convnet** will perform **probabilistic classification** (https://en.wikipedia.org/wiki/Probabilistic_classification)
  - Model will output an **array of 10 probabilities** indicating **likelihood that a digit belongs to a particular class 0-9**
  - **Highest probability** is the **predicted value**

Reproducibility in Keras and Deep Learning

- In deep learning, **reproducibility is difficult** because the libraries **heavily parallelize operations** that perform floating-point calculations
- Each time operations execute, they may execute in a **different order**
- Can produce **different results** in each execution
- **Reproducibility in Keras** requires a combination of environment settings and code settings that are described in the [Keras FAQ (https://keras.io/getting-started/faq/#how-can-i-obtain-reproducible-results-using-keras-during-development)](https://keras.io/getting-started/faq/#how-can-i-obtain-reproducible-results-using-keras-during-development)

Components of a Keras Neural Network

- **Network** (also called a **model**)
  - Sequence of layers containing the neurons used to learn from the samples
  - Each layer's neurons receive inputs, process them (via an **activation function**) and produce outputs.
  - Data is fed into the network via an **input layer** that specifies the dimensions of the sample data
  - Followed by **hidden layers** of neurons that **implement the learning** and an **output layer that produces the predictions**.
  - The more layers you **stack**, the deeper the network is, hence the term **deep learning**
- **Loss function**
  - Produces a **measure of how well the network predicts target values**
  - **Lower loss values** indicate **better predictions**
- **Optimizer**
  - Attempts to **minimize the values produced by the loss function** to **tune the network** to make better predictions

Launch JupyterLab

- Activate your `tf_env` Anaconda environment
- Launch JupyterLab from the `ch14` examples folder
- Open `MNIST_CNN.ipynb` in JupyterLab

15.6.1 Loading the MNIST Dataset

In [2]:
```
from tensorflow.keras.datasets import mnist
```

- `"tensorflow."` because we're using the **version of Keras built into TensorFlow**
- **`load_data` function** loads **training** and **testing sets**

In [3]:
```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

15.6.2 Data Exploration

- Check dimensions of the **training set images (`X_train`)**, **training set labels (`y_train`)**, **testing set images (`X_test`)** and **testing set labels (`y_test`)**:

In [4]:
```
X_train.shape
```

Out[4]:
```
(60000, 28, 28)
```

In [5]:
```
y_train.shape
```

Out[5]:
```
(60000,)
```

In [6]:
```
X_test.shape
```

Out[6]:
```
(10000, 28, 28)
```

In [7]:
```
y_test.shape
```

Out[7]:
```
(10000,)
```

Visualizing Digits (1 of 2)

In [8]:
```
%matplotlib inline
```

In [9]:
```
import matplotlib.pyplot as plt
```

In [10]:
```
import seaborn as sns
```

In [11]:
```
sns.set(font_scale=2)  # 2x normal Seaborn font size
```

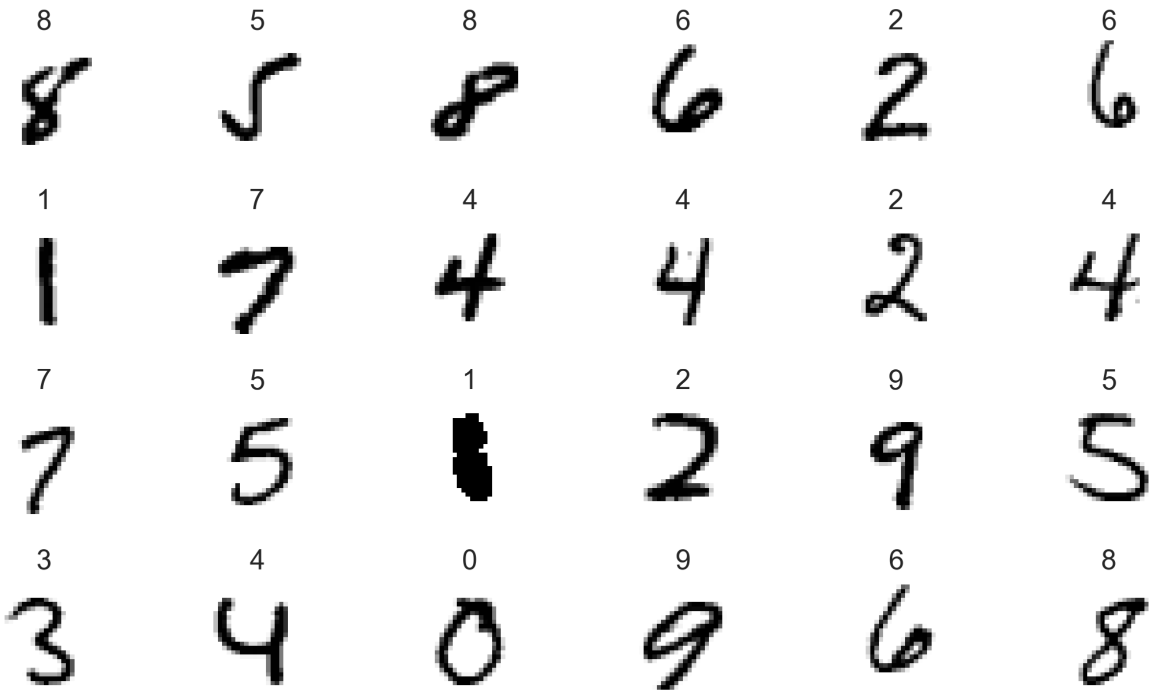
Visualizing Digits—Display a 24 Random MNIST Training Set Images (2 of 2)

- Pass a sequence of indexes as a NumPy array's subscript to select only the array elements at those indexes
- Run cell several times to view different digits and see **why handwritten digit recognition is a challenge**

In [12]:
```
import numpy as np
index = np.random.choice(np.arange(len(X_train)), 24, replace=False)
figure, axes = plt.subplots(nrows=4, ncols=6, figsize=(16, 9))

for item in zip(axes.ravel(), X_train[index], y_train[index]):
    axes, image, target = item
    axes.imshow(image, cmap=plt.cm.gray_r)
    axes.set_xticks([])  # remove x-axis tick marks
    axes.set_yticks([])  # remove y-axis tick marks
    axes.set_title(target)

plt.tight_layout()
```



In [13]:
```
sns.set(font_scale=1)  # reset font scale
```

15.6.3 Data Preparation

- **Scikit-learn's bundled datasets** were **preprocessed** into the **shapes its models required**
- In real-world studies, you'll generally have to do some or all of the **data preparation**
- **MNIST dataset requires some preparation for use in a Keras convnet**

Reshaping the Image Data (1 of 2)

- **Keras convnets** require **NumPy array inputs**
- Each **sample** must have the **shape**

```
( width , height , channels )
```

- MNIST images' *width* and *height* are 28 pixels
- Each pixel has one **channel** (grayscale shade 0-255)
- Each sample's shape will be: **`(28, 28, 1)`**
- As the **neural network learns** from the images, it **creates many more channels**
  - Rather than shade or color, the **learned channels** will **represent more complex features**, like **edges**, **curves** and **lines**
  - Enable network to **recognize digits** based on these features and how they're **combined**

Reshaping the Image Data (1 of 2)

- NumPy array method `reshape` receives a tuple representing the new shape

In [14]:
```
X_train = X_train.reshape((60000, 28, 28, 1))
```

In [15]:
```
X_train.shape
```

Out[15]:
```
(60000, 28, 28, 1)
```

In [16]:
```
X_test = X_test.reshape((10000, 28, 28, 1))
```

In [17]:
```
X_test.shape
```

Out[17]:
```
(10000, 28, 28, 1)
```

Normalizing the Image Data

- Data samples' **numeric feature values** may vary widely
- Deep learning networks **perform better** on data that's
  - Scaled into the range **0.0-1.0**, or
  - Scaled to a range for which the data's **mean is 0.0** and its **standard deviation is 1.0**
    - S. Ioffe and Szegedy, C., "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." https://arxiv.org/abs/1502.03167 (https://arxiv.org/abs/1502.03167)
- Scaling into one of these forms is known as **normalization**
- **Each pixel** has the value **0–255**

In [18]:
```
X_train = X_train.astype('float32') / 255
```

In [19]:
```
X_test = X_test.astype('float32') / 255
```

One-Hot Encoding: Converting the Labels From Integers to Categorical Data (1 of 4)

- **Predictions** for each digit will be an **array of 10 probabilities**
- To **evaluate model accuracy**, Keras **compares predictions to dataset's labels**
  - Both must have the **same shape**
  - MNIST labels are **individual integers 0-9**
- Must **transform the labels** into **categorical data arrays** matching the **prediction format**

One-Hot Encoding: Converting the Labels From Integers to Categorical Data (2 of 4)

- Use **one-hot encoding** (https://en.wikipedia.org/wiki/One-hot) to convert labels from integers into 10-element **arrays of 1.0s and 0.0s** in which **only one element is 1.0** and the **rest are 0.0s**
- A **7's categorical representation**

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0]
```

One-Hot Encoding: Converting the Labels From Integers to Categorical Data (3 of 4)

- **`tensorflow.keras.utils`** function **`to_categorical`** performs **one-hot encoding**
  - Counts unique categories then, for each item being encoded, creates an array of that length with a 1.0 in the correct position

One-Hot Encoding: Converting the Labels From Integers to Categorical Data (4 of 4)

- Transform **`y_train`** and **`y_test`** from one-dimensional arrays of `0`–`9` values into **two-dimensional arrays of categorical data**

In [20]:
```
from tensorflow.keras.utils import to_categorical
```

In [21]:
```
y_train = to_categorical(y_train)
```

In [22]:
```
y_train.shape
```

Out[22]:
```
(60000, 10)
```

In [23]:
```
y_train[0]  # one sample’s categorical data
```

Out[23]:
```
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

In [24]:
```
y_test = to_categorical(y_test)
```

In [25]:
```
y_test.shape
```

Out[25]:
```
(10000, 10)
```

15.6.4 Creating the Neural Network

- Configure a **convolutional neural network**
- Begin with Keras’s **`Sequential` model**

In [26]:
```
from tensorflow.keras.models import Sequential
```

In [27]:
```
cnn = Sequential()
```

- The resulting network will **execute its layers sequentially**—the **output of one layer** becomes the **input to the next**
  - **Feed-forward network**
  - When we discuss **recurrent neural networks**, you'll see that not all neural network operate this way

Adding Layers to the Network

- A typical **convnet** consists of **several layers**
  - **input layer** that receives the **training samples**
  - **hidden layers** that **learn** from the samples
  - **output layer** that **produces the prediction probabilities**
- Import layer classes for a basic **convnet**

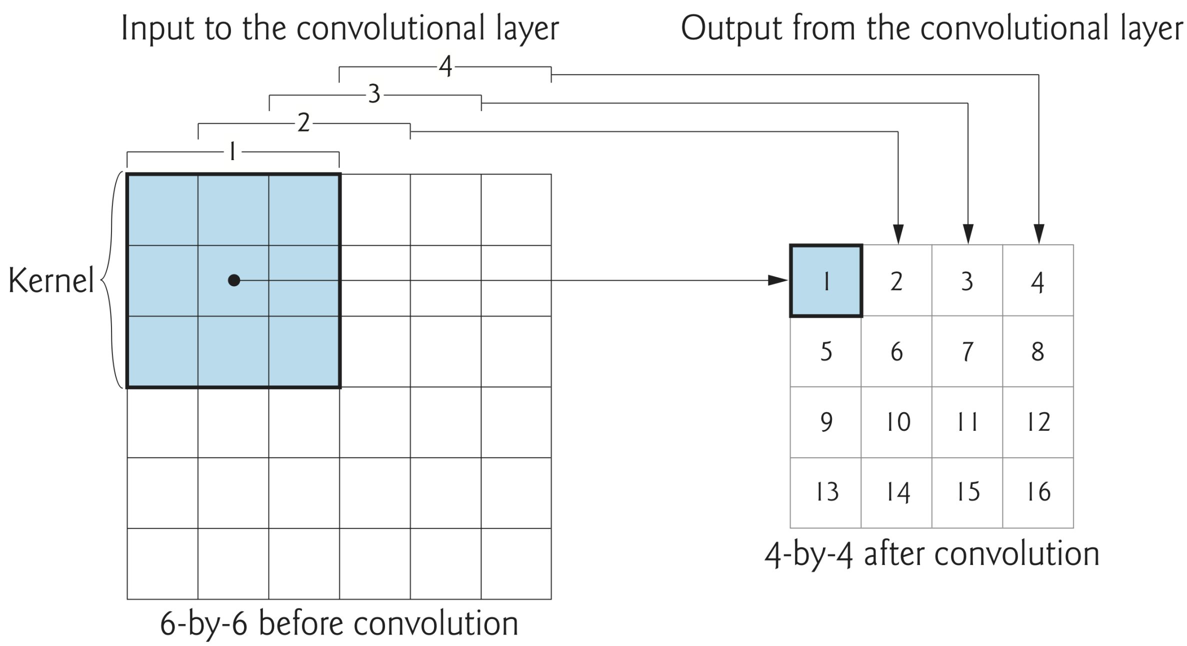
In [28]:
```
from tensorflow.keras.layers import Conv2D, Dense, Flatten, MaxPooling2D
```

Convolution (1 of 6)

- Begin our network with a **convolution layer**
- Uses the **relationships between pixels that are close to one another** to learn useful **features** (or patterns) in small areas of each sample
- These **features** become **inputs** to **subsequent layers**
- The small areas that **convolution** learns from are called **kernels** or **patches**

Convolution (2 of 6)

- Examine convolution on a 6-by-6 image
- **3-by-3 shaded square** represents the **kernel**

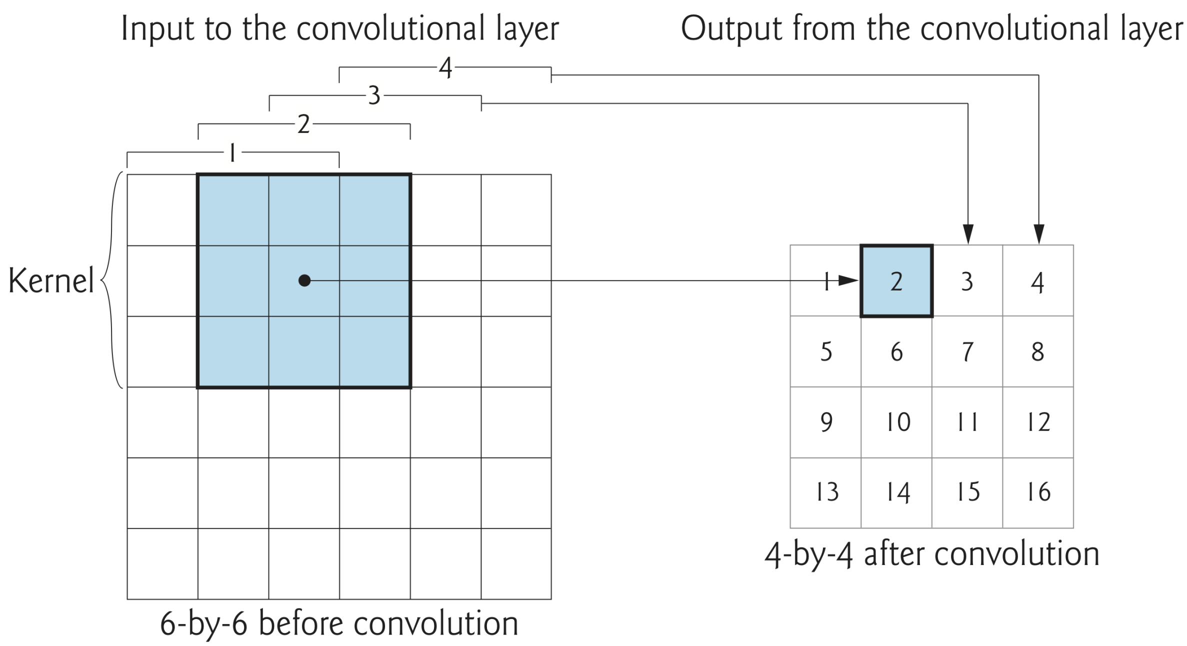


Convolution (3 of 6)

- **Kernel** is a **“sliding window”** that moves **one pixel at a time** left-to-right across the image
- When the kernel reaches the right edge, it moves one pixel **down** and repeats left-to-right process
- **Kernels typically are 3-by-3** (https://www.quora.com/How-can-I-decide-the-kernel-size-output-maps-and-layers-of-CNN), though we found convnets that used **5-by-5** and **7-by-7**
    - Kernel-size is a **tunable hyperparameter**
- **Convolution layer** performs calculations using those **nine** features to **“learn”** about them, then **outputs one new feature** to corresponding position in layer’s output
- By looking at **features near one another**, the network begins to **recognize features** like **edges**, **straight lines** and **curves**

Convolution (4 of 6)

- Next, **convolution layer** moves **kernel one pixel to the right** (known as the **stride**) to position 2 in the input layer
- **Overlaps** with two of three columns in previous position, so **convolution layer can learn** from all **features that touch one another**



Convolution (5 of 6)

- **Complete pass** left-to-right and top-to-bottom is called a **filter**
- For a **3-by-3 kernel**, the filter dimensions will be **two less than the input dimensions**
  - For each 28-by-28 MNIST image, the filter will be 26-by-26
- **Number of filters** in the **convolutional layer** is commonly **32** or **64** for small images, and each filter produces different results
  - **higher-resolution images** have **more features**, so they **require more filters**
  - **Keras team's code** for their **[pretrained convnets](https://github.com/keras-team/keras-applications/tree/master/keras_applications)** (https://github.com/keras-team/keras-applications/tree/master/keras_applications) uses 64, 128 or even 256 filters in their **first convolutional layers**
  - After studying their **convnets**, we chose **64 filters**

Convolution (6 of 6)

- **Set of filters** produced by a **convolution layer** is called a **feature map**
- Subsequent **convolution layers** combine features from previous feature maps to **recognize larger features** and so on
    - If we were doing **facial recognition**, **early layers** might recognize **lines**, **edges** and **curves**, and **subsequent layers** might begin **combining** those into **larger features** like **eyes**, **eyebrows**, **noses**, **ears** and **mouths**
- Once the **network learns a feature**, because of **convolution**, it can **recognize that feature anywhere** in the **image**
    - One reason **convnets** are popular for **object recognition** in images

Adding a `Conv2D` Convolution Layer (1 of 2)

In [29]:
```
cnn.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu',
               input_shape=(28, 28, 1)))
```

```
WARNING:tensorflow:From /Users/pauldeitel/anaconda3/envs/tf_env/lib/pyt
hon3.6/site-packages/tensorflow/python/ops/resource_variable_ops.py:43
5: colocate_with (from tensorflow.python.framework.ops) is deprecated a
nd will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

- **`filters=64`** —The number of **filters** in the resulting **feature map**.
- **`kernel_size=(3, 3)`** —The **size of the kernel** used in each **filter**
- **`activation='relu'`** —The 'relu' **(Rectified Linear Unit) activation function** is used to produce this layer's output.
    - `'relu'` is the **most widely used activation function** (Chollet, François. *Deep Learning with Python*. p. 72. Shelter Island, NY: Manning Publications, 2018)
    - [**Good for performance** because it's **easy to calculate** (https://towardsdatascience.com/exploring-activation-functions-for-neural-networks-73498da59b02)](https://towardsdatascience.com/exploring-activation-functions-for-neural-networks-73498da59b02)
    - [Commonly recommended for **convolutional layers**. (https://www.quora.com/How-should-I-choose-a-proper-activation-function-for-the-neural-network)](https://www.quora.com/How-should-I-choose-a-proper-activation-function-for-the-neural-network)

Adding a `Conv2D` Convolution Layer (2 of 2)

- This is the **first layer** in the model, so we pass the `input_shape=(28, 28,1)` to specify the shape of each sample
  - **Creates an input layer** to **load the samples** and pass them into the **`Conv2D` layer**, which is actually the **first hidden layer**
- Each subsequent layer **infers `input_shape`** from **previous layer's output shape**, making it easy to **stack** layers

Dimensionality of the First Convolution Layer's Output

- Input samples are 28-by-28-by-1—that is, **784 features each.**
- We specified **64 filters** and a **3-by-3 kernel size** for the layer, so the **output for each image is 26-by-26-by-64** for a total of **43,264 features** in the **feature map**
  - **Significant increase in dimensionality**
  - **Enormous** compared to numbers of features processed in our Machine Learning examples
- As each layer adds more features, the resulting feature maps' **dimensionality** becomes significantly larger
  - This is one of reason **deep learning studies often require tremendous processing power**

Overfitting (1 of 2)

- Recall from the previous chapter, that **overfitting** can occur when your **model is too complex** compared to what it is modeling
- **Most extreme case**: Model **memorizes** its training data
- When you make predictions with an **overfit model**, they will be **accurate** if **new data matches the training data**, but the model could **perform poorly** with **data it has never seen**
- **Overfitting** tends to occur in **deep learning** as the **dimensionality** of the **layers** becomes **too large** [1] (https://cs231n.github.io/convolutional-networks/),[2] (https://medium.com/@cxu24/why-dimensionality-reduction-is-important-dd60b5611543),[3] (https://towardsdatascience.com/preventing-deep-neural-network-from-overfitting-953458db800a)

Overfitting (2 of 2)

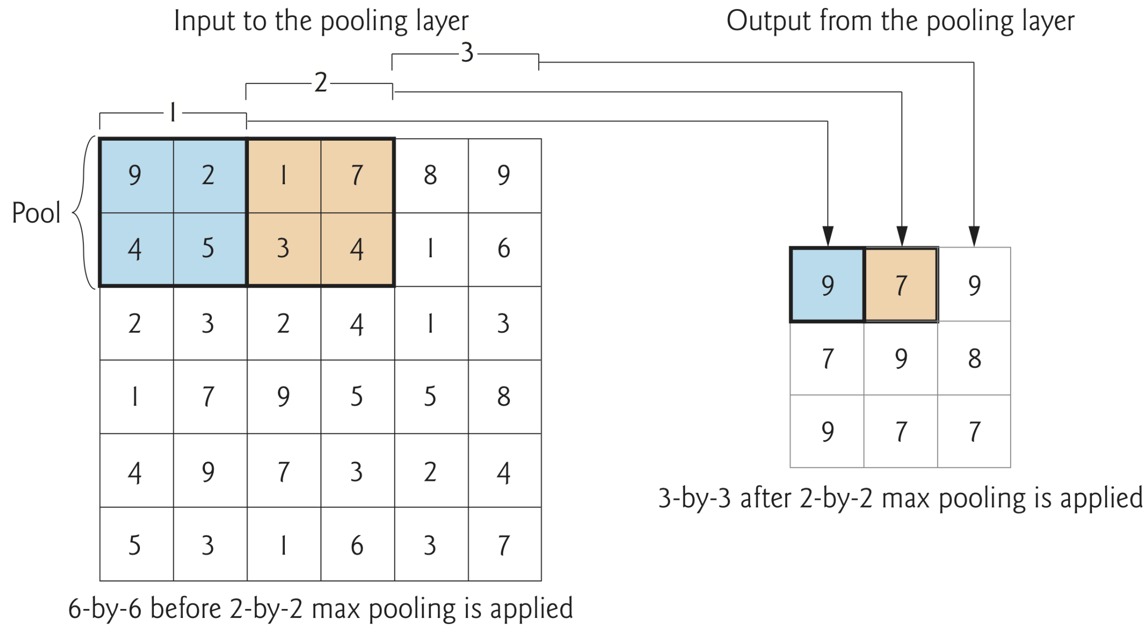
- Causes the network to **learn specific features** of the training-set digit images, **rather than learning the general features** of digit images
- Techniques to **prevent overfitting** [1] (https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d), [2] (https://www.kdnuggets.com/2015/04/preventing-overfitting-neural-networks.html)
  - **Training for fewer epochs**
  - **Data augmentation**
  - **Dropout** (discussed later)
  - **L1 or L2 regularization**
- **Higher dimensionality** also increases (and sometimes explodes) **computation time**
- For deep learning on **CPUs** rather than **GPUs** or **TPUs**, training could become **intolerably slow**

Adding a Pooling Layer (1 of 3)

- To **reduce overfitting** and **computation time**, a **convolution layer** is often followed by one or more layers that **reduce the dimensionality** of **convolution layer's output**
- A **pooling layer compresses** (or **down-samples**) the results by **discarding features**
  - Helps make the model **more general**
- **Most common pooling technique** is called **max pooling**
  - Examines a 2-by-2 square of features and keeps only the maximum feature.

Adding a Pooling Layer (2 of 3)

- 2-by-2 blue square in position 1 represents the initial pool of features to examine:



Adding a Pooling Layer (3 of 3)

- Looks at the pool in position 1, then outputs the **maximum feature** from that pool
- **No overlap** between pools
- **Pool's stride** for a 2-by-2 pool is **2**
- Because every group of four features is reduced to one, 2-by-2 pooling **compresses** the number of features by **75%**
  - Reduces previous layer’s output from **26-by-26-by-64** to **13-by-13-by-64**

In [30]:
```
cnn.add(MaxPooling2D(pool_size=(2, 2)))
```

Adding Another Convolutional Layer and Pooling Layer

- **Convnets** often have **many convolution and pooling layers.**
- [Keras team's convnets (https://github.com/keras-team/keras-applications/tree/master/keras_applications)](https://github.com/keras-team/keras-applications/tree/master/keras_applications) tend to **double** the number of **filters** in subsequent convolutional layers to enable the models to learn more relationships between the features

In [31]:
```
cnn.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu'))
```

In [32]:
```
cnn.add(MaxPooling2D(pool_size=(2, 2)))
```

- **Input** to the **second convolution layer** is the 13-by-13-by-64 **output of the first pooling layer**
- **Output** of this **Conv2D layer** will be **11-by-11-by-128**
- For **odd dimensions** like 11-by-11, **Keras pooling layers round down** by default (in this case to 10-by-10), so this pooling layer's **output** will be **5-by-5-by-128**

Flattening the Results to One Dimension with a Keras `Flatten` Layer

- Previous layer's output is three-dimensional (**5-by-5-by-128**),
- **Final output** of our model will be a **one-dimensional** array of 10 probabilities that classify the digits
- To prepare for **one-dimensional final predictions**, need to **flatten** the previous layer's output to **one dimension**
- **`Flatten`** layer's output will be **1-by-3200** (5 × 5 × 128)

In [33]:
```
cnn.add(Flatten())
```

Adding a Dense Layer to Reduce the Number of Features

- Layers before the **`Flatten`** layer **learned digit features**
- Now must **learn the relationships among those features** so our model can **classify** which digit each image represents
- Accomplished with **fully connected `Dense` layers**
- The following **`Dense` layer** creates **128 neurons (`units`)** that **learn** from the 3200 outputs of the previous layer

In [34]:
```
cnn.add(Dense(units=128, activation='relu'))
```

- Many **convnets** contain at least one `Dense` layer like the one above
- **Convnets** geared to more complex image datasets with higher-resolution images like **[ImageNet](http://www.image-net.org) (http://www.image-net.org)**—a dataset of over 14 million images—often have **several `Dense` layers**, commonly with **4096 neurons**
- Several [Keras pretrained ImageNet convnets (https://github.com/keras-team/keras-applications/tree/master/keras_applications)](https://github.com/keras-team/keras-applications/tree/master/keras_applications) do this

Adding Another Dense Layer to Produce the Final Output

- Final **`Dense`** layer **classifies** inputs into **neurons** representing the classes **0-9**
- The **`softmax` activation function** converts values of these 10 neurons into **classification probabilities**
- The **neuron** that produces the **highest probability** represents the **prediction** for a given digit image

In [35]:
```
cnn.add(Dense(units=10, activation='softmax'))
```

Printing the Model’s Summary with the Model’s `summary` Method (1 of 2)

- Note layers' **output shapes** and **numbers of parameters**
- **Parameters** are the **weights** that the network **learns** during training [1] (https://hackernoon.com/everything-you-need-to-know-about-neural-networks-8988c3ee4491),[2] (https://www.kdnuggets.com/2018/06/deep-learning-best-practices-weight-initialization.html)
- **Relatively small network**, but needs to **learn nearly 500,000 parameters**!
  - This is for **tiny images** that are less than 1/4 the size of icons on smartphone home screens
  - Imagine how many features a network would have to learn to process high-resolution 4K video frames or the super-high-resolution images produced by today’s digital cameras.
- In the **`Output Shape`** column, **`None`** means the model does not know in advance how many training samples you’re going to provide

In [36]:
```
cnn.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 26, 26, 64)        640       
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 64)        0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 128)       73856     
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 128)         0         
_________________________________________________________________
flatten (Flatten)            (None, 3200)              0         
_________________________________________________________________
dense (Dense)                (None, 128)               409728    
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290      
=================================================================
Total params: 485,514
Trainable params: 485,514
Non-trainable params: 0
_________________________________________________________________
```

Printing the Model’s Summary with the Model’s `summary` Method (2 of 2)

- There are **no “non-trainable” parameters**
- **By default, Keras trains all parameters**, but it's possible to prevent training for specific layers (https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers)
  - e.g., when you’re tuning networks or using another model’s learned parameters in a new model (called **transfer learning**)

Visualizing a Model’s Structure with the `plot_model` Function from Module `tensorflow.keras.utils`

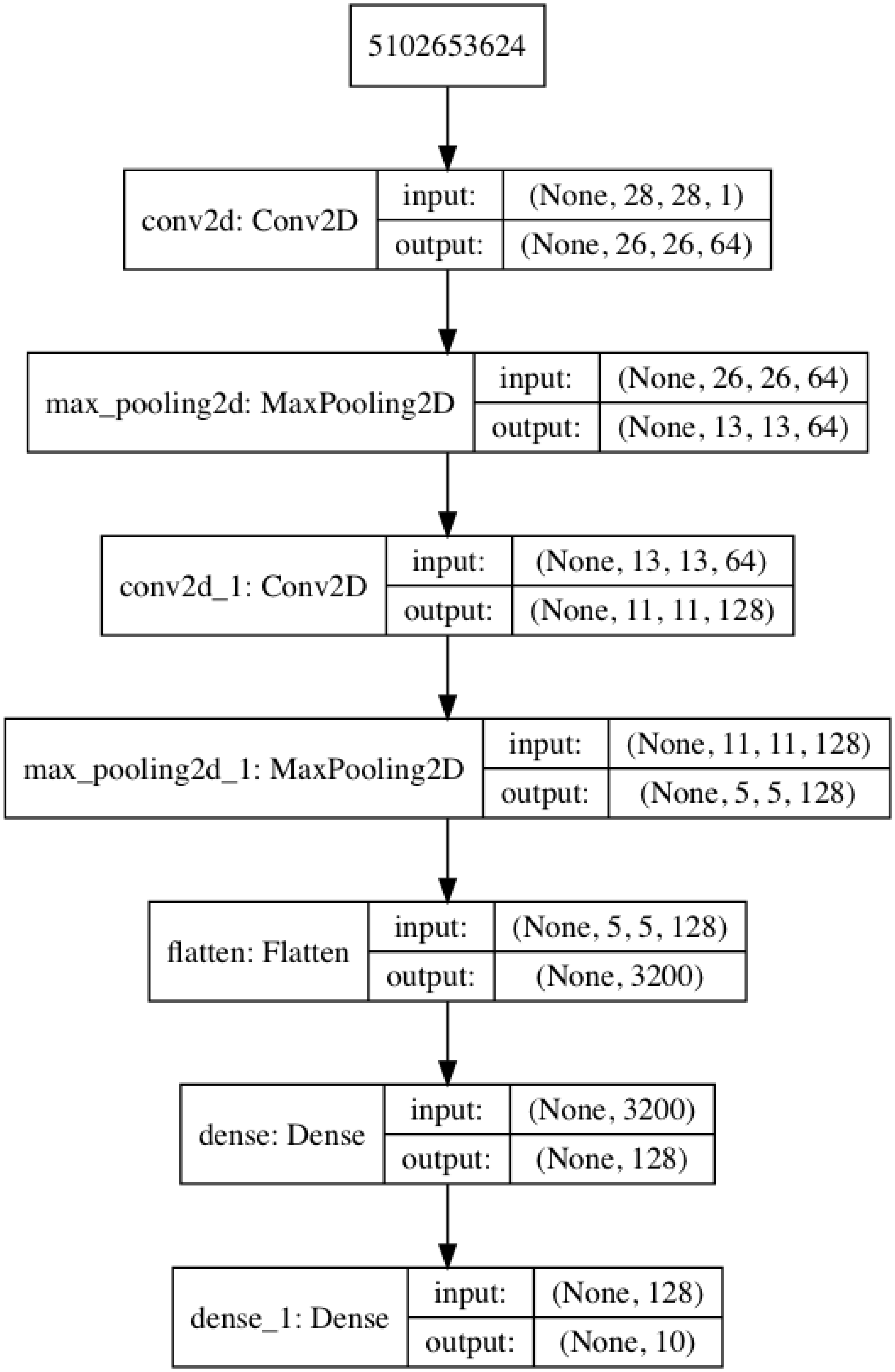
In [37]:
```
from tensorflow.keras.utils import plot_model
```

In [38]:
```
from IPython.display import Image
```

In [39]:
```
plot_model(cnn, to_file='convnet.png', show_shapes=True,
           show_layer_names=True)
```

In [40]:
```
Image(filename='convnet.png')  # display resulting image in notebook
```

Out[40]:



- Keras assigns the layer names in the image.
  - The node at the top of the diagram appears to be a bug—it represents the implicit **`InputLayer`** in our convnet

Compiling the Model (1 of 4)

- Complete the model by calling its **`compile`** **method**

In [41]:
```
cnn.compile(optimizer='adam',
            loss='categorical_crossentropy',
            metrics=['accuracy'])
```

Compiling the Model (2 of 4)

- **`optimizer='adam'`** —The **optimizer** this model uses to **adjust the weights** throughout the neural network **as it learns**
  - [**Keras optimizers** (https://keras.io/optimizers/)](https://keras.io/optimizers/)
  - `'adam'` performs well across a wide variety of models [[1] (https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2)](https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2),[[2] (https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f)](https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f)

Compiling the Model (3 of 4)

- **`loss='categorical_crossentropy'`** —The **loss function** used by the optimizer in **multi-classification networks** like our convnet, which predicts 10 classes
  - As the neural network learns, the **optimizer** attempts to **minimize the values returned by the loss function**
  - The **lower** the loss, the **better** the neural network is at predicting what each image is
  - For **binary classification**, Keras provides **`'binary_crossentropy'`** , and for **regression**, **`'mean_squared_error'`**
  - [Other loss functions (https://keras.io/losses/)](https://keras.io/losses/)

Compiling the Model (4 of 4)

- **`metrics=['accuracy']`** —List of **metrics** the network will produce to help you **evaluate the model**
  - **Accuracy** commonly used in **classification models**
  - We'll use it to check **percentage of correct predictions**
  - [Other metrics (https://keras.io/metrics/)](https://keras.io/metrics/)

15.6.5 Training and Evaluating the Model (1 of 6)

- **Train a Keras model** by calling its **`fit` method**
- As in **Scikit-learn**, the **first two arguments** are the **training data** and the **categorical target labels**
- **`epochs`** specifies the number of times the model should process the **entire set of training data**

15.6.5 Training and Evaluating the Model (2 of 6)

- **`batch_size=64`** —**number of samples to process at a time** during each epoch
  - Most models specify a **power of 2 from 32 to 512**
  - **Larger batch sizes can decrease model accuracy**
    - Keskar, Nitish Shirish, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy and Ping Tak Peter Tang. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." CoRR abs/1609.04836 (2016). [https://arxiv.org/abs/1609.04836 (https://arxiv.org/abs/1609.04836)](https://arxiv.org/abs/1609.04836).

15.6.5 Training and Evaluating the Model (3 of 6)

- Some samples should be used to **validate the model**
  - If you specify **validation data**, after each **epoch**, the model will use it to **make predictions** and display the **validation loss and accuracy**
  - Study these values to **tune your layers** and the **`fit` method's hyperparameters**, or possibly change the **layer composition** of your model
- **`validation_split=0.1`** [—model should reserve the **last** 10% of the training samples for validation (https://keras.io/getting-started/faq/#how-is-the-validation-split-computed)](https://keras.io/getting-started/faq/#how-is-the-validation-split-computed)
  - For **separate validation data**, use **`validation_data` argument** to specify a tuple containing arrays of samples and target labels
- Best to get **randomly selected validation data**
  - Can use **scikit-learn's `train_test_split` function** for this purpose (as we'll do later), then pass the randomly selected data with the `validation_data` argument

15.6.5 Training and Evaluating the Model (4 of 6)

- Model took about 5 minutes to train on our CPU.
- **Lecture note: Play convnet timelapse video here**

In [42]:
```
cnn.fit(X_train, y_train, epochs=5, batch_size=64, validation_split=0.1)
```

```
Train on 54000 samples, validate on 6000 samples
WARNING:tensorflow:From /Users/pauldeitel/anaconda3/envs/tf_env/lib/pyt
hon3.6/site-packages/tensorflow/python/ops/math_ops.py:3066: to_int32
(from tensorflow.python.ops.math_ops) is deprecated and will be removed
in a future version.
Instructions for updating:
Use tf.cast instead.
Epoch 1/5
54000/54000 [==============================] - 64s 1ms/sample - loss:
0.1400 - acc: 0.9560 - val_loss: 0.0467 - val_acc: 0.9857
Epoch 2/5
54000/54000 [==============================] - 63s 1ms/sample - loss:
0.0430 - acc: 0.9868 - val_loss: 0.0493 - val_acc: 0.9872
Epoch 3/5
54000/54000 [==============================] - 63s 1ms/sample - loss:
0.0283 - acc: 0.9911 - val_loss: 0.0404 - val_acc: 0.9892
Epoch 4/5
54000/54000 [==============================] - 63s 1ms/sample - loss:
0.0212 - acc: 0.9932 - val_loss: 0.0353 - val_acc: 0.9908
Epoch 5/5
54000/54000 [==============================] - 63s 1ms/sample - loss:
0.0153 - acc: 0.9950 - val_loss: 0.0318 - val_acc: 0.9918
```

Out[42]:
```
<tensorflow.python.keras.callbacks.History at 0x13c139518>
```

15.6.5 Training and Evaluating the Model (5 of 6)

- As training proceeds, **`fit`** shows the **progress** of each epoch, **how long** the epoch took to execute, and the **evaluation metrics** for that epoch
- Impressive **training accuracy (`acc`)** and **validation accurracy (`acc`)**, given that **we have not yet tried to tune the hyperparameters** or **tweak the number and types of the layers**
    - Doing so could lead to **even better (or worse) results**

15.6.5 Training and Evaluating the Model (6 of 6)

- Soon we'll show **TensorBoard**
  - TensorFlow tool for **visualizing data from deep-learning models**
  - View charts showing how **accuracy and loss values** change through the epochs

===P says I think we need to ditch this for the course. Just don't have enough time.===

- We'll also demonstrate **Andrej Karpathy's ConvnetJS tool**, which **trains convnets in your web browser** and **dynamically visualizes the layers' outputs**, including **what each convolutional layer "sees" as it learns**
- Try running his **MNIST** and **CIFAR10 models** to help you better understand neural networks' complex operations

Evaluating the Model on Unseen Data with Model's `evaluate` Method

- Displays how long it took to process test samples

In [43]:
```
loss, accuracy = cnn.evaluate(X_test, y_test)
```

```
10000/10000 [==============================] - 3s 332us/sample - loss:
0.0281 - acc: 0.9913
```

In [44]:
```
loss
```

Out[44]:
```
0.028109445479651912
```

In [45]:
```
accuracy
```

Out[45]:
```
0.9913
```

- Our **convnet model** is **99+% accurate** for unseen data samples
  - Again, **we have not tried to tune the model**
  - Can find models online that predict MNIST with even **higher accuracy**
  - **Experiment** with different numbers of layers, types of layers and layer parameters and observe how those changes affect your results

Thought Experiment

=====================================

- Now that you've seen this tremendous accuracy on your first try, assume you've never heard of ML or DL and you've never heard about learning from data
- Go back to your years of programming
- Your boss comes to you with 70K images in MNIST and says, we're building an app where the post office, by machine, needs to recognize handwritten digits for speeding the routing and deliver of mail via zip codes
- If you never heard of ML, DL and learning from data, think back 10 or 20 years to how you'd try solving that problem and what percentage correct you'd be likely to get on the first try
- The boss says, can you handle this? What would you have said?
- This is what's exciting about the field today
- With ML and DL we **are** able to solve problems like this
- All of the sudden, computer vision is reasonable to do
- Self-driving cars must recognize objects
- **ML/DL open entire new classes of problems that you can now solve**
- You're becoming aware of big data and big data sources, accumulating big data in your companies, and using that data for things like fraud detection, sentiment analysis, ...

=====================================

Making Predictions with the Model's `predict` Method

In [46]:
```
predictions = cnn.predict(X_test)
```

- The first digit should be a 7 (shown as `1.` at index 7)

In [47]:
```
y_test[0]
```

Out[47]:
```
array([0., 0., 0., 0., 0., 0., 0., 1., 0., 0.], dtype=float32)
```

- Check the **probabilities** returned by `predict` method for the first test sample

In [48]:
```
for index, probability in enumerate(predictions[0]):
    print(f'{index}: {probability:.10%}')
```

```
0: 0.0000006316%
1: 0.0000011190%
2: 0.0000728502%
3: 0.0002137894%
4: 0.0000000244%
5: 0.0000001597%
6: 0.0000000001%
7: 99.9993801117%
8: 0.0000056546%
9: 0.0003242511%
```

- Our model believes this digit is a 7 with **nearly** 100% certainty
- Not all predictions have this level of certainty

Locating the Incorrect Predictions (1 of 2)

- View some **incorrectly predicted images** to get a sense of digits **our model has trouble with**
  - If the model always mispredicts 8s, perhaps we need more 8s in our training data
- To determine whether a prediction was correct, compare the index of the largest probability in `predictions[0]` to the index of the element containing **`1.0` in `y_test[0]`**
  - If **indices** are the same, **prediction was correct**

Locating the Incorrect Predictions (2 of 2)

- In the following snippet, **`p` is the predicted value array**, and **`e` is the expected value array**
- **Reshape the samples** from the shape `(28, 28, 1)` that Keras required for learning back to `(28, 28)`, which **Matplotlib requires to display the images**

In [49]:
```
images = X_test.reshape((10000, 28, 28))
```

In [50]:
```
incorrect_predictions = []
```

- **NumPy's `argmax` function** determines **index** of an array's **highest valued element**

In [51]:
```
for i, (p, e) in enumerate(zip(predictions, y_test)):
    predicted, expected = np.argmax(p), np.argmax(e)

    if predicted != expected:  # prediction was incorrect
        incorrect_predictions.append(
            (i, images[i], predicted, expected))
```

In [52]:
```
len(incorrect_predictions)  # number of incorrect predictions
```

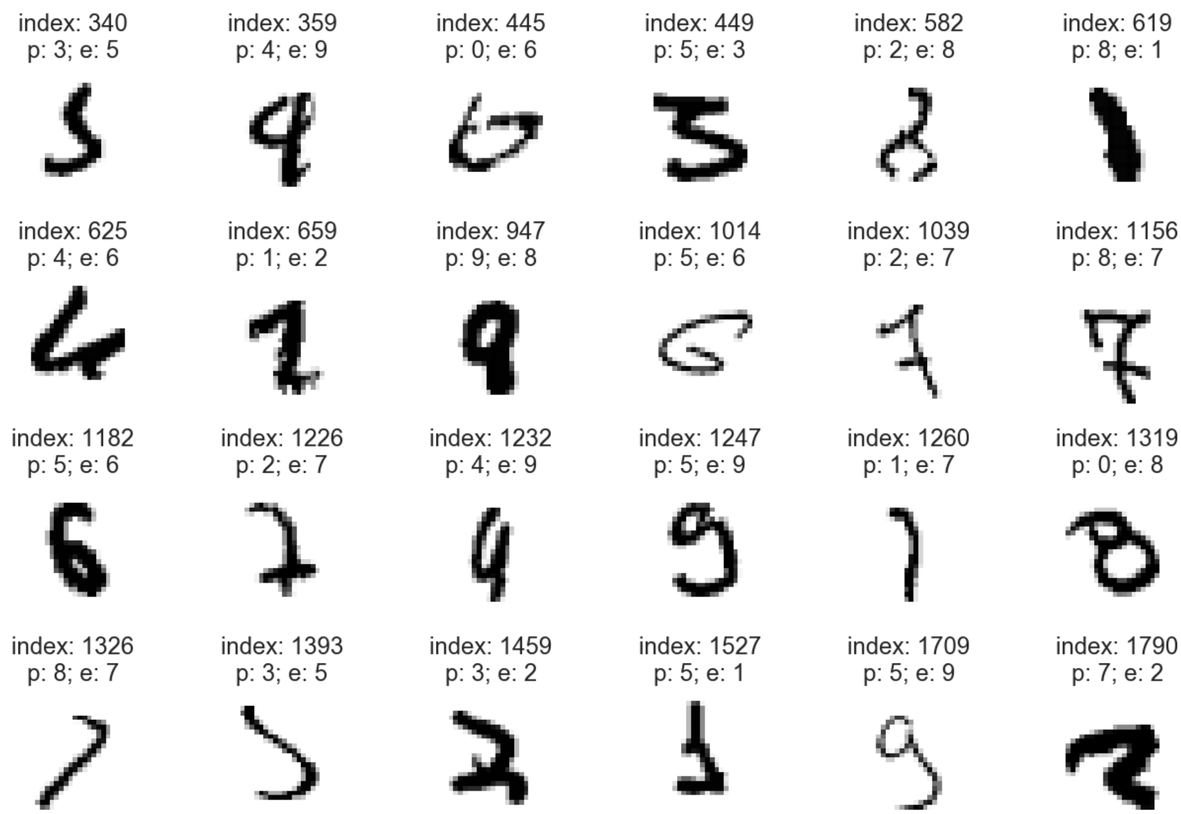
Out[52]:
```
87
```

Visualizing Incorrect Predictions

- **Display 24 of the incorrect images** labeled with each image's index, predicted value (`p`) and expected value (`e`)
- Before reading the expected values, look at each digit and write down what digit you think it is
- This is an important part of **getting to know your data**

In [56]:
```
figure, axes = plt.subplots(nrows=4, ncols=6, figsize=(9, 6))

for axes, item in zip(axes.ravel(), incorrect_predictions):
    index, image, predicted, expected = item
    axes.imshow(image, cmap=plt.cm.gray_r)
    axes.set_xticks([])  # remove x-axis tick marks
    axes.set_yticks([])  # remove y-axis tick marks
    axes.set_title(f'index: {index}\np: {predicted}; e: {expected}')
plt.tight_layout()
```



Displaying the Probabilities for Several Incorrect Predictions

- The following function displays the probabilities for the specified prediction array:

In [57]:
```
def display_probabilities(prediction):
    for index, probability in enumerate(prediction):
        print(f'{index}: {probability:.10%}')
```

- **Lecture Note: Consider loading the saved model to make predictions**

In [58]:
```
display_probabilities(predictions[359])
```

```
0: 0.0000012315%
1: 0.0001513258%
2: 0.0000569705%
3: 0.0002494840%
4: 46.8818992376%
5: 0.0004904704%
6: 0.0000002439%
7: 0.0010067255%
8: 9.0622462332%
9: 44.0539032221%
```

In [60]:
```
display_probabilities(predictions[625])
```

```
0: 0.0016457274%
1: 0.0000025435%
2: 0.0159131130%
3: 0.0000000637%
4: 83.9228630066%
5: 0.0000002097%
6: 16.0485476255%
7: 0.0000001066%
8: 0.0000100639%
9: 0.0110190755%
```

In [61]:
```
display_probabilities(predictions[659])
```

```
0: 0.0083214283%
1: 45.8956420422%
2: 11.0333994031%
3: 2.9275920242%
4: 0.0051769490%
5: 0.0001196304%
6: 0.0000356249%
7: 40.0392353535%
8: 0.0595510937%
9: 0.0309274561%
```

15.6.6 Saving and Loading a Model (1 of 2)

- Once you've designed and tested a model that suits your needs, you can **save its state**
- Can **load it later** to **make more predictions**
- Sometimes models are loaded and **further trained** for new problems
  - For example, layers in our model can **recognize features such as lines and curves**, which could be useful in **handwritten character recognition** (as in the EMNIST dataset) as well
- Could potentially load the existing model and **use it as the basis for a more robust model**
- Called **transfer learning**[1] (https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751), [2] (https://medium.com/nanonets/nanonets-how-to-use-deep-learning-when-you-have-limited-data-f68c0b512cab)—transfer an existing model's knowledge into a new model

15.6.6 Saving and Loading a Model (2 of 2)

- **`save` method** stores a model's architecture and state information in a format called **Hierarchical Data Format (HDF5; `.h5` file extension)**

In [62]:
```
cnn.save('mnist_cnn.h5')
```

- Load a saved model with **`load_model` function**

```
from tensorflow.keras.models import load_model
cnn = load_model('mnist_cnn.h5')
```

- Can then invoke its methods
  - Could call **`predict`** to make additional predictions on new data
  - Could call **`fit`** to start training with the additional data.
- Additional functions that enable you to **save and load various aspects of your models** (https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model)

15.7 Visualizing Neural Network Training with TensorBoard

- With deep learning networks, there's **so much complexity** and **so much going on internally** that's **hidden** from you that it's difficult to know and fully understand all the details
- Creates challenges in testing, debugging and updating models and algorithms
- Deep learning learns enormous numbers of features that may not be apparent to you
- Google provides **TensorBoard** (https://github.com/tensorflow/tensorboard/blob/master/README.md) ([1] (https://www.tensorflow.org/guide/summaries_and_tensorboard)) for **visualizing TensorFlow and Keras neural networks**
- A **TensorBoard dashboard** can give you insights into how well your model is learning and potentially help you **tune its hyperparameters**

Executing TensorBoard

===IF WE'RE GOING TO SHOW THE TIMELAPSE OF THIS, KEEP CURRENT SLIDE. TO SAVE TIME, I COULD JUST SHOW THE NEXT SLIDE AND TALK ABOUT IT, THEN HAVE THEM GO TO PYTHON FUNDAMENTALS FOR THE FULL DISCUSSION===

- TensorBoard **monitors a folder** on your system looking for files containing the data it will visualize in a **web browser**
- Must create that folder, **execute the TensorBoard server**, then access it via a web browser

1. At your command line, change to the `ch15` folder
2. Ensure that your custom Anaconda environment `tf_env` is activated:

```
conda activate tf_env
```

3. Create a **subfolder named `logs`** — deep-learning models will output **information** here to **visualize**
4. **Execute TensorBoard**

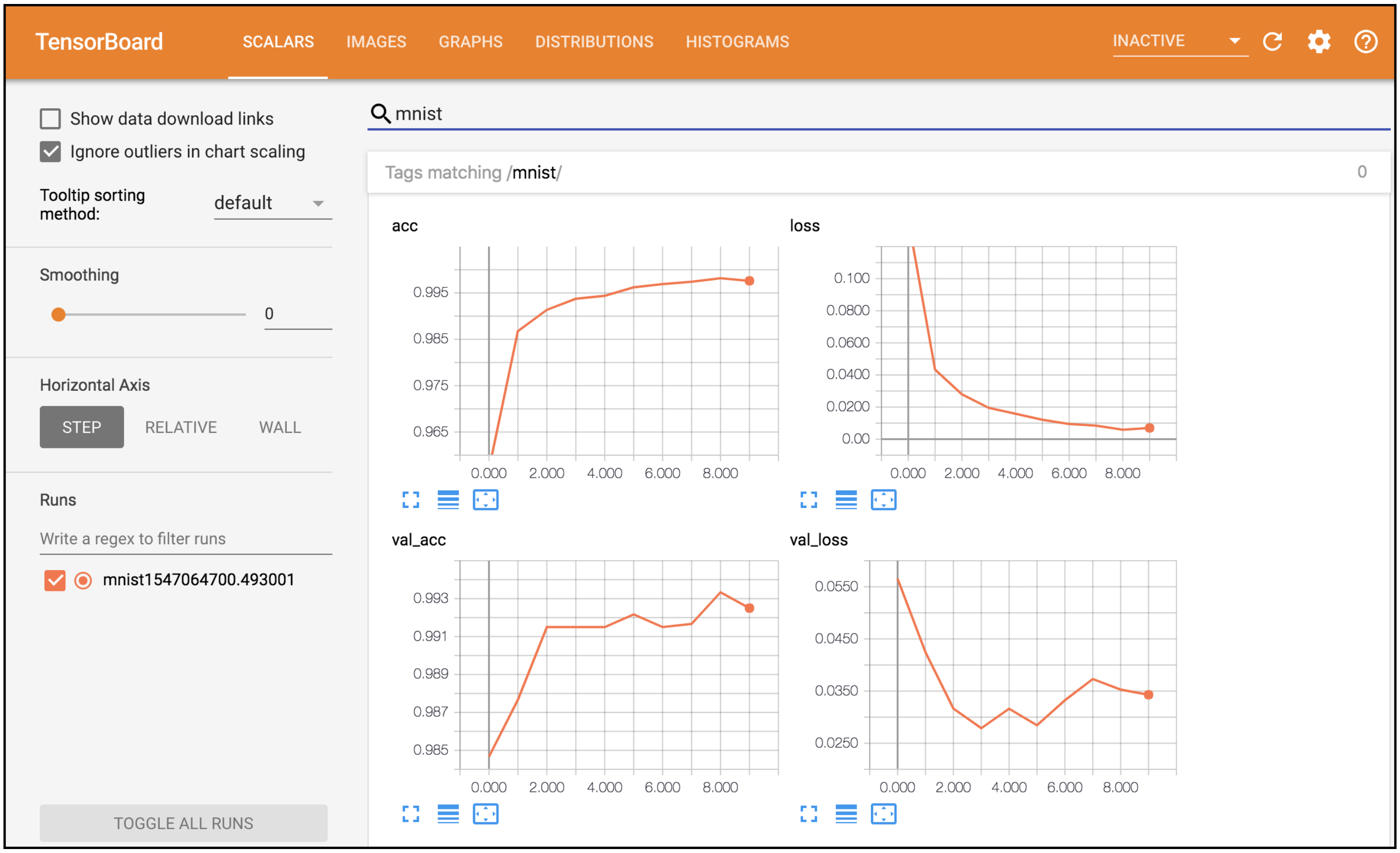
```
tensorboard --logdir=logs
```

5. Access TensorBoard in your web browser at

> http://localhost:6006 (http://localhost:6006)

The TensorBoard Dashboard (1 of 4)

- When TensorBoard sees updates in the **`logs`** folder, it loads the data into the dashboard


TensorBoard
SCALARS
IMAGES
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method:
default
Smoothing
0
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
mnist1547064700.493001
TOGGLE ALL RUNS
mnist
Tags matching /mnist/
0
acc
0.995
0.985
0.975
0.965
0.000
2.000
4.000
6.000
8.000
loss
0.100
0.0800
0.0600
0.0400
0.0200
0.00
0.000
2.000
4.000
6.000
8.000
val_acc
0.993
0.991
0.989
0.987
0.985
0.000
2.000
4.000
6.000
8.000
val_loss
0.0550
0.0450
0.0350
0.0250
0.000
2.000
4.000
6.000
8.000

The TensorBoard Dashboard (2 of 4)

===CONSIDER ditching rest of this section and referencing the video/book presentation. Full details of this section are presented in my **Python Fundamentals LiveLessons videos** (https://learning.oreilly.com/videos/python-fundamentals/9780135917411) (deep learning lesson coming soon) and in **Python for Programmers, Section 15.7** (https://learning.oreilly.com/library/view/python-for-programmers/9780135231364/ch15.xhtml#ch15lev1sec7) **===**

- Can **view data as you train** or **after training completes**
- **Dashboard** above shows the **TensorBoard `SCALARS` tab**, which displays **charts** for individual values that change over time, such as the **training accuracy (`acc`)** and **training loss (`loss`)** shown in the first row, and the **validation accuracy (`val_acc`)** and **validation loss (`val_loss`)** shown in the second row
- The diagrams visualize a **10-epoch run of our MNIST convnet**, which we provided in the notebook **`MNIST_CNN_TensorBoard.ipynb`**
- The **epochs** are displayed **along the x-axes** starting from 0 for the first epoch
- The **accuracy** and **loss values** are displayed on the **y-axes**

The TensorBoard Dashboard (3 of 4)

- Looking at the **training and validation accuracies**, you can see in the first **5 epochs** similar results to our **5-epoch run**
- For the **10-epoch run**, the **training accuracy** continued to **improve** through the **9th epoch**, then **decreased** slightly
- **This might be the point at which we're starting to overfit**, but we might need to train longer to find out
- For the **validation accuracy**, you can see that it jumped up quickly, then was **relatively flat** for **five epochs** before **jumping** up then **decreasing**
- For the **training loss**, you can see that it drops quickly, then continuously **declines through the ninth epoch**, before a slight increase
- The **validation loss** dropped quickly then bounced around

The TensorBoard Dashboard (1 of 4)

- We could **run this model for more epochs** to see whether results improve, but based on these diagrams, it appears that around the **sixth epoch** we get a nice combination of **training and validation accuracy** with **minimal validation loss**
- Normally these diagrams are stacked vertically in the dashboard
- We used the search field (above the diagrams) to show any that had the name “mnist” in their folder name —we’ll configure that in a moment
- **TensorBoard** can **load data** from **multiple models at once** and you can **choose** which to **visualize**
- This makes it **easy to compare** several different **models** or multiple runs of the same model

Copy the MNIST Convnet’s Notebook

- To create the new notebook for this example:
  1. Right-click the **`MNIST_CNN.ipynb` notebook** in **JupyterLab’s `File Browser` tab** and select `Duplicate` to make a copy of the notebook.
  2. Right-click the new notebook named **`MNIST_CNN-Copy1.ipynb`**, then select `Rename`, enter the name **`MNIST_CNN_TensorBoard.ipynb`** and press **Enter**.
- Open the notebook by double-clicking its name.

Configuring Keras to Write the TensorBoard Log Files (1 of 2)

- To use **TensorBoard**, before you **`fit`** the model, you need to configure a **`TensorBoard`** object (**module `tensorflow.keras.callbacks`**), which the model will use to **write data into a specified folder** that TensorBoard monitors
  - This object is known as a **callback** in Keras
- In the notebook, click to the left of snippet that calls the **model’s `fit` method**, then type **`a`**, which is the shortcut for adding a new code cell **above** the current cell (use **`b`** for **below**)

Configuring Keras to Write the TensorBoard Log Files (2 of 2)

- In the new cell, enter the following code to **create the `TensorBoard` object**

```
from tensorflow.keras.callbacks import TensorBoard

import time

tensorboard_callback = TensorBoard(log_dir=f'./logs/mnist{time.time()}',
    histogram_freq=1, write_graph=True)
```

- **`log_dir`** —Folder in which this **model's log files** will be **written**
  - Names based on time ensure that each new executio will have its own folder
  - Enables you to **compare multiple executions** in TensorBoard
- **`histogram_freq`** —Frequency in **epochs** that Keras will output to model's logs— `1` means every epoch
- **`write_graph`** —When true, outputs a graph of the model
  - View the graph in the **GRAPHS** tab in TensorBoard

Updating Our Call to `fit`

- Finally, we need to modify the **original `fit` method call**
- For this example, we **set the number of epochs to 10**, and we added the [**`callbacks` argument**, which is a list of callback objects (https://keras.io/callbacks/)](https://keras.io/callbacks/)

```
cnn.fit(X_train, y_train, epochs=10, batch_size=64,
        validation_split=0.1, callbacks=[tensorboard_callback])
```

- Re-execute the notebook by selecting **Kernel > Restart Kernel and Run All Cells** in JupyterLab
- After the first epoch completes, you'll start to see data in TensorBoard

15.8 ConvnetJS: Browser-Based Deep-Learning Training and Visualization

===COOL, but we need to save time. I think we should cut. H says: Refine this to say why it's so nice. What's goes on is so complicated and the number of features so enormous, not the kind of thing that you can think about on a step by step basis. More and more tools like this will appear, helping you get a better sense of what's going on inside by visualizing the training. Have to experiment, since you can't understand directly.===

- **Andrej Karpathy's JavaScript-based ConvnetJS tool enables training and visualizing convolutional neural networks in your web browser** (https://cs.stanford.edu/people/karpathy/convnetjs/)
- Can run his **sample convnets** or **create your own**
- **ConvnetJS MNIST demo** presents a **scrollable dashboard** that updates dynamically as the model trains

Training Stats

- **Pause** button enables you to **stop the learning** and **"freeze" the current dashboard visualizations**
- Clicking the **resume** button **continues training**
- Presents **training statistics**, including the **training and validation accuracy** and a **graph of the training loss**

Instantiate a Network and Trainer

- Contains the **JavaScript code** that **creates the convolutional neural network**.
- Similar layers to the convnet we created
- The ConvnetJS documentation (https://cs.stanford.edu/people/karpathy/convnetjs/docs.html) shows the **supported layer types** and how to **configure** them
- Can **experiment** with **different layer configurations** in the provided textbox and begin training an updated network by clicking the **change network** button

Network Visualization

- **Shows one training image at a time** and **how the network processes that image through each layer**
- Click the **Pause** button to inspect all the layers' outputs for a given digit to get a sense of **what the network "sees" as it learns**
- The network's **last layer** produces the **probabilistic classifications**.
- It shows **10 squares—9 black and 1 white**—indicating the **predicted class** of the **current digit image**

Example Predictions on Test Set

Shows a **random selection** of the **test set images** and the **top three possible classes for each digit**.

- The one with the **highest probability** is **shown on a green bar** and the **other two** are displayed on **red bars**
- **Length** of each bar is a **visual indication** of class's **probability**

15.9 Recurrent Neural Networks for Sequences; Sentiment Analysis with the IMDb Dataset (1 of 4)

- Our **convnet** used **stacked layers** that were applied **sequentially**
- **Non-sequential models** are possible, such as **recurrent neural networks (RNN)**
- We use **Keras's bundled IMDb (the Internet Movie Database) movie reviews dataset** to perform **binary classification**, predicting whether a given **review's sentiment** is **positive** or **negative**
  - Dataset: Maas, Andrew L. and Daly, Raymond E. and Pham, Peter T. and Huang, Dan and Ng, Andrew Y. and Potts, Christopher, "Learning Word Vectors for Sentiment Analysis," *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, June 2011. Portland, Oregon, USA. Association for Computational Linguistics, pp. 142–150. http://www.aclweb.org/anthology/P11-1015 (http://www.aclweb.org/anthology/P11-1015).
- **RNNs** process **sequences of data**, such as **time series** or **text in sentences**
- **"Recurrent"** comes from the fact that the **neural network contains loops**
  - The **output of a given layer** becomes the **input to that same layer** in the **next time step**

15.9 Recurrent Neural Networks for Sequences; Sentiment Analysis with the IMDb Dataset (2 of 4)

- In a **time series**, a **time step** is the **next point in time**
- In a **text sequence**, a **time step** is the **next word in a sequence of words**
- **Looping in RNNs** enables them to **learn and remember relationships** among the data in the sequence

15.9 Recurrent Neural Networks for Sequences; Sentiment Analysis with the IMDb Dataset (3 of 4)

- **"Good" on its own has positive sentiment**
- When **preceded by "not,"** which appears **earlier in the sequence**, the **sentiment becomes negative**
- **RNNs** take into account the **relationships among the earlier and later parts of a sequence**
- In the preceding example, the words that determined sentiment were adjacent
- However, when determining the meaning of text there can be **many words to consider** and an **arbitrary number of words in between them**

15.9 Recurrent Neural Networks for Sequences; Sentiment Analysis with the IMDb Dataset (4 of 4)

- We'll use a **Long Short-Term Memory (LSTM)** layer, which makes the neural network **recurrent** and is optimized to handle learning from sequences like the ones we described above.
- RNNs have been used for many tasks including:[1] (https://www.analyticsindiamag.com/overview-of-recurrent-neural-networks-and-their-applications/),[2] (https://en.wikipedia.org/wiki/Recurrent_neural_network#Applications),[3] (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)
  - **predictive text input**—displaying possible next words as you type,
  - **sentiment analysis**
  - **responding to questions with the predicted best answers** from a corpus,
  - **inter-language translation**
  - **automated video closed captioning**

15.9.1 Loading the IMDb Movie Reviews Dataset (1 of 2)

- Contains **25,000 training samples** and **25,000 testing samples**, each **labeled** with its positive (1) or negative (0) sentiment

In [1]:
```
from tensorflow.keras.datasets import imdb
```

- Module's **`load_data` function** returns the **IMDb training and testing sets**
- **Over 88,000 unique words** in the dataset
- Can specify the **number of unique words to import** as part of the **training and testing data**
- We loaded only the top **10,000 most frequently occurring words** due to the **memory limitations of our system** and the fact that we're (intentionally) **training on a CPU**
  - Most people don't have systems with Tensorflow-compatible **GPUs** or **TPUs**
- The **more data** you load, the **longer training will take**, but more data may help produce **better models**

15.9.1 Loading the IMDb Movie Reviews Dataset (1 of 2)

- In a given review, **`load_data`** **replaces** any words **outside the top 10,000** with a **placeholder** value (discussed shortly)

In [2]:
```
number_of_words = 10000
```

Note: Following cell was added to work around a **known issue with TensorFlow/Keras and NumPy**—this issue is already fixed in a forthcoming version. [See this cell's code on StackOverflow. (https://stackoverflow.com/questions/55890813/how-to-fix-object-arrays-cannot-be-loaded-when-allow-pickle-false-for-imdb-loa)](https://stackoverflow.com/questions/55890813/how-to-fix-object-arrays-cannot-be-loaded-when-allow-pickle-false-for-imdb-loa)

In [3]:
```
import numpy as np

# save np.load
np_load_old = np.load

# modify the default parameters of np.load
np.load = lambda *a,**k: np_load_old(*a, allow_pickle=True, **k)
```

In [4]:
```
(X_train, y_train), (X_test, y_test) = imdb.load_data(
    num_words=number_of_words)
```

In [5]:
```
# This cell completes the work around mentioned above

# restore np.load for future normal usage
np.load = np_load_old
```

15.9.2 Data Exploration

- Check sample and target dimensions

In [6]:
```
X_train.shape
```
Out[6]:
```
(25000,)
```

In [7]:
```
y_train.shape
```
Out[7]:
```
(25000,)
```

In [8]:
```
X_test.shape
```
Out[8]:
```
(25000,)
```

In [9]:
```
y_test.shape
```
Out[9]:
```
(25000,)
```

- The **arrays `y_train` and `y_test`** are **one-dimensional** arrays containing **1s and 0s**, indicating whether each review is **positive** or **negative**
- `X_train` and `X_test` are **lists** of integers, each representing one review's contents
- **Keras deep learning models require numeric data**, so the Keras team **preprocessed the IMDb dataset for you**

In [10]:
```
%pprint  # toggle pretty printing, so elements don't display vertically
```
```
Pretty printing has been turned OFF
```

In [11]:
```
X_train[123]
```

Out[11]:
```
[1, 307, 5, 1301, 20, 1026, 2511, 87, 2775, 52, 116, 5, 31, 7, 4, 91, 1
220, 102, 13, 28, 110, 11, 6, 137, 13, 115, 219, 141, 35, 221, 956, 54,
13, 16, 11, 2714, 61, 322, 423, 12, 38, 76, 59, 1803, 72, 8, 2, 23, 5,
967, 12, 38, 85, 62, 358, 99]
```

Movie Review Encodings (1 of 3)

- Because the **movie reviews** are **numerically encoded**, to view their original text, you need to know the word to which each number corresponds
- **Keras's IMDb dataset** provides a **dictionary** that **maps the words to their indexes**
- **Each word's corresponding value** is its **frequency ranking** among all the words in the entire set of reviews
- The **word with the ranking 1** is the **most frequently occurring word** (calculated by the Keras team from the dataset), the word with ranking 2 is the second most frequently occurring word, and so on

Movie Review Encodings (2 of 3)

- Though the dictionary values begin with 1 as the most frequently occurring word, in each encoded review (like `X_train[123]` shown previously), the ranking values are **offset by 3**.
- So any review containing the most frequently occurring word will have the value 4 wherever that word appears in the review.

Movie Review Encodings (3 of 3)

- **Keras reserves the values 0, 1 and 2 in each encoded review**:
  - **0 in a review** represents **padding**
    - **Keras deep learning algorithms expect all the training samples to have the same dimensions**, so some reviews may need to be expanded to a given length and some shortened to that length
    - **Reviews that need to be expanded are padded with 0s**
  - **1** represents a **token** that **Keras uses internally** to indicate the **start of a text sequence** for learning purposes
  - **2** represents an **unknown word**—typically a word that was **not loaded** because you **called `load_data` with the `num_words` argument**
    - Any words with **frequency rankings greater than `num_words`** are replaced with **2**
- **Reviews' numeric values are offset by 3**—must account for this when **decoding reviews**

Decoding a Movie Review (1 of 4)

- Get the **word-to-index dictionary**

In [12]:
```
word_to_index = imdb.get_word_index()
```

- The word `'great'` might appear in a positive movie review, so let's see whether it's in the dictionary

In [13]:
```
word_to_index['great']  # 84th most frequent word
```

Out[13]:
```
84
```

Decoding a Movie Review (2 of 4)

- To transform frequency ratings into words, **reverse the `word_to_index` dictionary's mapping**, so we can **look up every word** by its **frequency rating**

In [14]:
```
index_to_word = {index: word for (word, index) in word_to_index.items()}
```

- Show the **top 50 words—most frequent word** has the key **1** in the **new dictionary**

In [15]:
```
[index_to_word[i] for i in range(1, 51)]
```

Out[15]:
```
['the', 'and', 'a', 'of', 'to', 'is', 'br', 'in', 'it', 'i', 'this', 't
hat', 'was', 'as', 'for', 'with', 'movie', 'but', 'film', 'on', 'not',
'you', 'are', 'his', 'have', 'he', 'be', 'one', 'all', 'at', 'by', 'a
n', 'they', 'who', 'so', 'from', 'like', 'her', 'or', 'just', 'about',
"it's", 'out', 'has', 'if', 'some', 'there', 'what', 'good', 'more']
```

Decoding a Movie Review (3 of 4)

- Most of these are **stop words**
- Depending on the application, you might want to **remove or keep the stop words**
  - If you were creating a **predictive-text application**, **you'd want to keep the stop words** so they can be displayed as predictions

Decoding a Movie Review (4 of 4)

- Now, we can **decode a review**
- **`i - 3`** accounts for the **frequency ratings offsets in the encoded reviews**
- For `i` values `0`–`2`, `get` returns `'?'`; otherwise, `get` returns the word with the **key `i - 3`** in the **`index_to_word` dictionary**

In [16]:
```
' '.join([index_to_word.get(i - 3, '?') for i in X_train[123]])
```

Out[16]:
```
'? beautiful and touching movie rich colors great settings good acting
and one of the most charming movies i have seen in a while i never saw
such an interesting setting when i was in china my wife liked it so muc
h she asked me to ? on and rate it so other would enjoy too'
```

- Can see from **`y_train[123]`** that this **review** is **classified as positive**

In [17]:
```
y_train[123]
```

Out[17]:
```
1
```

15.9.3 Data Preparation (1 of 3)

- Number of words per review varies, but Keras **requires all samples to have the same dimensions**
- So, we need to perform some **data preparation**
  - Restrict every review to the **same** number of words
  - Some reviews will need to be **padded** with additional data and others will need to be **truncated**
- Keras **`pad_sequences` utility function** reshapes `X_train`'s samples to the number of features specified by the **`maxlen` argument** and **returns a two-dimensional array**

In [18]:
```
words_per_review = 200
```

In [19]:
```
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

In [20]:
```
X_train = pad_sequences(X_train, maxlen=words_per_review)
```

15.9.3 Data Preparation (2 of 3)

- If a sample has **more features**, `pad_sequences` **truncates** it to the specified length
- If a sample has **fewer features**, `pad_sequences` **adds `0`s** to the beginning of the sequence to **pad it** to the specified length

In [21]:
```
X_train.shape
```

Out[21]:
```
(25000, 200)
```

15.9.3 Data Preparation (3 of 3)

- Must also **reshape `X_test`** evaluating the model later

In [22]:
```
X_test = pad_sequences(X_test, maxlen=words_per_review)
```

In [23]:
```
X_test.shape
```

Out[23]: (25000, 200)

Splitting the Test Data into Validation and Test Data

- Our **convnet** used **`fit` method's `validation_split` argument** to set aside **10%** of training data to **validate the model** as it trains
- Here, we'll manually split the **25,000 test samples** into **20,000 test samples** and **5,000 validation samples**, then pass the 5,000 validation samples to the model's `fit` method via the argument **`validation_data`**
- Use **Scikit-learn's `train_test_split` function**

In [24]:
```
from sklearn.model_selection import train_test_split
```

In [25]:
```
X_test, X_val, y_test, y_val = train_test_split(
    X_test, y_test, random_state=11, test_size=0.20)
```

- Confirm the split by checking `X_test` 's and `X_val` 's shapes:

In [26]:
```
X_test.shape
```

Out[26]: (20000, 200)

In [27]:
```
X_val.shape
```

Out[27]: (5000, 200)

15.9.4 Creating the Neural Network

- Begin with a **`Sequential` model** and import the other layers

In [28]:
```
from tensorflow.keras.models import Sequential
```

In [29]:
```
rnn = Sequential()
```

```
In [30]: from tensorflow.keras.layers import Dense, LSTM, Embedding
```

Adding an Embedding Layer (1 of 3)

- Our convnet example used **one-hot encoding** to convert the **MNIST's integer labels** into **categorical** data
  - **Result for each label** was a **vector** in which **all but one element was 0**
- Could do that for the index values that represent our words
- However, this example processes **10,000 unique words**
  - We'd need a **10,000-by-10,000 array** to represent all the words
  - **100,000,000 elements** and **almost all** would be **0**
  - **Not efficient** way to **encode** the data
  - For **all 88,000+ unique words** in the dataset, we'd need an array of nearly **eight billion elements**!

Adding an Embedding Layer (2 of 3)

- To **reduce dimensionality**, RNNs that process **text sequences** typically begin with an **embedding layer** that encodes each word in a more compact **dense-vector representation**
  - These vectors capture the **word's context**—that is, **how a given word relates to the words around it**
  - Enables the **RNN** to **learn word relationships among the training data**
- There are also **predefined word embeddings**, such as **Word2Vec** and **GloVe**
  - Can **load** into neural networks to **save training time**
  - Sometimes used to **add basic word relationships** to a model when **smaller amounts of training data** are available
  - Can improve the model's accuracy by allowing it to **build upon previously learned word relationships**, rather than trying to learn those relationships with insufficient amounts of data

Adding an `Embedding` Layer (3 of 3)

In [31]:
```
rnn.add(Embedding(input_dim=number_of_words, output_dim=128,
                  input_length=words_per_review))
```

```
WARNING:tensorflow:From /Users/pauldeitel/anaconda3/envs/tf_env/lib/pyt
hon3.6/site-packages/tensorflow/python/ops/resource_variable_ops.py:43
5: colocate_with (from tensorflow.python.framework.ops) is deprecated a
nd will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

- **`input_dim=number_of_words`** —Number of **unique words**
- **`output_dim=128`** —Size of each word embedding
  - If you [load pre-existing embeddings (https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html)](https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html) like **Word2Vec** and **GloVe**, you must set this to **match the size of the word embeddings you load**
- **`input_length=words_per_review`** —Number of words in each input sample

Adding an LSTM Layer

In [32]:
```
rnn.add(LSTM(units=128, dropout=0.2, recurrent_dropout=0.2))
```

```
WARNING:tensorflow:From /Users/pauldeitel/anaconda3/envs/tf_env/lib/pyt
hon3.6/site-packages/tensorflow/python/keras/backend.py:4010: calling d
ropout (from tensorflow.python.ops.nn_ops) with keep_prob is deprecated
and will be removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate =
1 - keep_prob`.
```

- **`units`** —**number of neurons** in the layer
  - **More neurons** means **network can remember more**
  - Guideline: [**Start with a value between the length of the sequences you're processing** (200 in this example) and the **number of classes you're trying to predict** (2 in this example) (https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046)](https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046)
- **`dropout`** —**percentage of neurons to randomly disable** when processing the layer's input and output
  - Like **pooling layers** in a **convnet**, **dropout** is a proven technique that **reduces overfitting**
    - Yarin, Ghahramani, and Zoubin. "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks." October 05, 2016. https://arxiv.org/abs/1512.05287 (https://arxiv.org/abs/1512.05287)
    - Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15 (June 14, 2014): 1929-1958. http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf (http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf)
  - Keras also provides a **`Dropout`** layer that you can add to your models
- **`recurrent_dropout`** —**percentage of neurons to randomly disable** when the **layer's output** is **fed back into the layer** again to allow the network to learn from what it has seen previously
  - **Mechanics of how the LSTM layer performs its task are beyond scope**.
    - Chollet says: "you don't need to understand anything about the specific architecture of an LSTM cell; **as a human, it shouldn't be your job to understand it**. Just keep in mind what the LSTM cell is meant to do: allow past information to be reinjected at a later time."
    - Chollet, François. *Deep Learning with Python*. p. 204. Shelter Island, NY: Manning Publications, 2018.

Adding a Dense Output Layer

- Reduce the **LSTM layer's output** to **one result** indicating whether a review is **positive** or **negative**, thus the value **`1` for the `units` argument**
- **`'sigmoid'` activation function** is preferred for **binary classification**
  - Chollet, François. *Deep Learning with Python*. p.114. Shelter Island, NY: Manning Publications, 2018.
  - Reduces arbitrary values into the range **0.0–1.0**, producing a probability

In [33]:
```
rnn.add(Dense(units=1, activation='sigmoid'))
```

Compiling the Model and Displaying the Summary

- **Two possible outputs**, so we use the **`binary_crossentropy` loss function**:

In [34]:
```
rnn.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy'])
```

- Even though we have **fewer layers** than our **convnet**, the **RNN** has nearly **three times as many trainable parameters** (the network's **weights**)
  - **More parameters means more training time**
  - The large number of parameters primarily comes from the **number of words in the vocabulary** (we loaded 10,000) **times the number of neurons in the `Embedding` layer's output (128)**

In [35]:
```
rnn.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding (Embedding)        (None, 200, 128)          1280000   
_________________________________________________________________
lstm (LSTM)                  (None, 128)               131584    
_________________________________________________________________
dense (Dense)                (None, 1)                 129       
=================================================================
Total params: 1,411,713
Trainable params: 1,411,713
Non-trainable params: 0
_________________________________________________________________
```

15.9.5 Training and Evaluating the Model (1 of 2)

- For each **epoch** the **RNN model** takes **significantly longer to train** than our **convnet**
  - Due to the **larger numbers of parameters** (weights) our **RNN model** needs to learn

In [36]:
```
rnn.fit(X_train, y_train, epochs=10, batch_size=32,
        validation_data=(X_test, y_test))
```

```
Train on 25000 samples, validate on 20000 samples
WARNING:tensorflow:From /Users/pauldeitel/anaconda3/envs/tf_env/lib/pyt
hon3.6/site-packages/tensorflow/python/ops/math_ops.py:3066: to_int32
(from tensorflow.python.ops.math_ops) is deprecated and will be removed
in a future version.
Instructions for updating:
Use tf.cast instead.
Epoch 1/10
25000/25000 [==============================] - 297s 12ms/sample - loss:
0.4827 - acc: 0.7673 - val_loss: 0.3925 - val_acc: 0.8324
Epoch 2/10
25000/25000 [==============================] - 291s 12ms/sample - loss:
0.3327 - acc: 0.8618 - val_loss: 0.3614 - val_acc: 0.8461
Epoch 3/10
25000/25000 [==============================] - 272s 11ms/sample - loss:
0.2662 - acc: 0.8937 - val_loss: 0.3503 - val_acc: 0.8492
Epoch 4/10
25000/25000 [==============================] - 272s 11ms/sample - loss:
0.2066 - acc: 0.9198 - val_loss: 0.3695 - val_acc: 0.8623
Epoch 5/10
25000/25000 [==============================] - 271s 11ms/sample - loss:
0.1612 - acc: 0.9403 - val_loss: 0.3802 - val_acc: 0.8587
Epoch 6/10
25000/25000 [==============================] - 291s 12ms/sample - loss:
0.1218 - acc: 0.9556 - val_loss: 0.4103 - val_acc: 0.8421
Epoch 7/10
25000/25000 [==============================] - 295s 12ms/sample - loss:
0.1023 - acc: 0.9634 - val_loss: 0.4634 - val_acc: 0.8582
Epoch 8/10
25000/25000 [==============================] - 273s 11ms/sample - loss:
0.0789 - acc: 0.9732 - val_loss: 0.5103 - val_acc: 0.8555
Epoch 9/10
25000/25000 [==============================] - 273s 11ms/sample - loss:
0.0676 - acc: 0.9775 - val_loss: 0.5071 - val_acc: 0.8526
Epoch 10/10
25000/25000 [==============================] - 273s 11ms/sample - loss:
0.0663 - acc: 0.9787 - val_loss: 0.5156 - val_acc: 0.8536
```

Out[36]:
```
<tensorflow.python.keras.callbacks.History object at 0x141462e48>
```

- At the time of this writing, TensorFlow displayed a **warning when we called `fit`**
- This is a **known TensorFlow issue** and, according to the forums, you can **safely ignore the warning**

15.9.5 Training and Evaluating the Model (2 of 2)

- Function **`evaluate`** returns the **loss and accuracy values**

In [37]:
```
results = rnn.evaluate(X_test, y_test)
```

```
20000/20000 [==============================] - 36s 2ms/sample - loss: 0.5156 - acc: 0.8536
```

In [38]:
```
results
```

Out[38]:
```
[0.5155695990145206, 0.85355]
```

- **Accuracy of this model seems low** compared to our **MNIST convnet's results**, but this is a **much more difficult problem**
- If you search online for other **IMDb sentiment-analysis binary-classification studies**, you'll find **lots of results in the high 80s.**
- We did **reasonably well** with our **small recurrent neural network** of only **three layers**
  - We have not tried to tune our model
  - Study some online models and try to produce a better model

15.10 Tuning Deep Learning Models (1 of 4)

- **Testing accuracy** and **validation accuracy** were **significantly less** than the **training accuracy**
  - Such **disparities** are usually the **result of overfitting**, so there is **plenty of room for improvement** in our model.[1] (https://towardsdatascience.com/deep-learning-overfitting-846bf5b35e24),[2] (https://hackernoon.com/memorizing-is-not-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42)
- **Each epoch's output** shows both the **training and validation accuracy continue to increase**
- **Training for too many epochs can lead to overfitting**, but it's **possible we have not yet trained enough**
  - Perhaps one **hyperparameter tuning** option for this model might be to **increase the number of epochs**

15.10 Tuning Deep Learning Models (2 of 4)

- Some **variables that affect your models' performance** include:
  - having **more or less data to train with**
  - having **more or less data to test with**
  - having **more or less data to validate with**
  - having **more or fewer layers**
  - the **types of layers** you use
  - the **order of the layers**

15.10 Tuning Deep Learning Models (3 of 4)

- Some **things we could tune** include:
  - trying **different amounts of training data**—we used only the top 10,000 words
  - different **numbers of words per review**—we used only 200
  - different **numbers of neurons** in our layers
  - **more layers**
  - possibly **loading pre-trained word vectors** rather than having our `Embedding` layer learn them from scratch.

15.10 Tuning Deep Learning Models (4 of 4)

- The **compute time** required to train models multiple times is **significant** so, in **deep learning**, you generally **do not tune hyperparameters with techniques like k-fold cross-validation or grid search** [1] (https://www.quora.com/Is-cross-validation-heavily-used-in-deep-learning-or-is-it-too-expensive-to-be-used)
- There are **various tuning techniques**, but one particularly promising area is **automated machine learning (AutoML)** [1] (https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a),[2] (https://medium.com/machine-learning-bites/deeplearning-series-deep-neural-networks-tuning-and-optimization-39250ff7786d),[3] (https://flyyufelix.github.io/2016/10/03/fine-tuning-in-keras-part1.html),[4] (https://flyyufelix.github.io/2016/10/08/fine-tuning-in-keras-part2.html),[5] (https://towardsdatascience.com/a-comprehensive-guide-on-how-to-fine-tune-deep-neural-networks-using-keras-on-google-colab-free-daaaa0aced8f)
  - **Auto-Keras** (https://autokeras.com/) is specifically geared to **automatically choosing** the **best configurations** for your **Keras models**
  - **Google's Cloud AutoML** and **Baidu's EZDL** are among various other automated machine learning and deep learning efforts

15.11 Convnet Models Pretrained on ImageNet

- With deep learning, rather than starting fresh on every project with costly training, validating and testing, you can use **pretrained deep neural network models** to:
  - **make new predictions**
  - **continue training** them further with **new data**
  - **transfer the weights learned by a model** for a similar problem into a new model—this is called **transfer learning.**

Keras Pretrained Convnet Models

===Can't find info on why these and why so many. Some articles mention these were from the ImageNet competition.===

- Keras comes bundled with [pretrained convnet models (https://keras.io/applications/)](https://keras.io/applications/), each pretrained on [**ImageNet** (http://www.image-net.org)](http://www.image-net.org)—a growing dataset of 14+ million images:
  - Xception
  - VGG16
  - VGG19
  - ResNet50
  - Inception v3
  - Inception-ResNet v2
  - MobileNet v1
  - DenseNet
  - NASNet
  - MobileNet v2

Reusing Pretrained Models

- **ImageNet is too big for efficient training** on most computers, so most people interested in using it **start with one of the smaller pretrained models**
- You can **reuse just the architecture of each model** and **train it with new data**, or you can **reuse the pretrained weights**
- [Simple examples of using pretrained models (https://keras.io/applications/)](https://keras.io/applications/)

ImageNet Challenge (1 of 3)

- **ImageNet Large Scale Visual Recognition Challenge** for evaluating object-detection and image-recognition models (http://www.image-net.org/challenges/LSVRC/).
  - This competition ran from 2010 through 2017.
- ImageNet now has a **continuously running challenge on the Kaggle competition site** called the **ImageNet Object Localization Challenge** (https://www.kaggle.com/c/imagenet-object-localization-challenge).
  - Goal: **Identify** "**all objects** within an **image**, so those images can then be **classified** and **annotated.**"
  - ImageNet releases the current participants leaderboard once per quarter.

ImageNet Challenge (2 of 3)

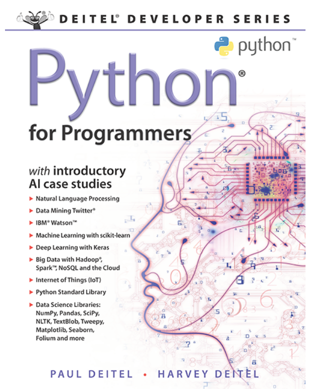
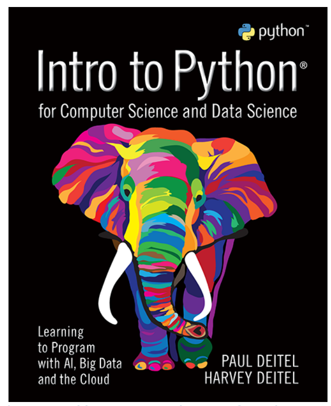
- A lot of what you've seen in the machine learning and deep learning chapters is what the **Kaggle competition website** is all about
- There's **no obvious optimal solution** for many **machine learning** and **deep learning tasks**
- People's creativity is really the only limit
- On **Kaggle**, companies and organizations **fund competitions** where they encourage people worldwide to develop better-performing solutions than they've been able to do for something that's important to their business or organization

ImageNet Challenge (3 of 3)

- Sometimes companies offer **prize money**, which has been as high as **$1,000,000** on the famous **Netflix competition**
- Netflix wanted to get a 10% or better improvement in their model for determining whether people will like a movie, based on how they rated previous ones (https://netflixprize.com/rules.html)
- They used the results to help make better recommendations to members
- Even if you do not win a **Kaggle competition**, it's a great way to get experience working on problems of current interest

More Info

- See **video** Lesson 15 in **Python Fundamentals LiveLessons** here on Safari Online Learning (https://learning.oreilly.com/videos/python-fundamentals/9780135917411)
- See Chapter 15 in **Python for Programmers** on Safari Online Learning (https://learning.oreilly.com/library/view/python-for-programmers/9780135231364/)
- See Chapter 16 in **Intro Python for Computer Science and Data Science** on VitalSource.com (https://www.vitalsource.com/products/intro-to-python-for-computer-science-and-data-paul-j-deitel-harvey-deitel-v9780135404812) or RedShelf.com (https://redshelf.com/book/1157786/intro-to-python-for-computer-science-and-data-science-1157786-9780135404812-paul-j-deitel-harvey-deitel)
- Interested in a print book? Check out:

Python for Programmers (640-page professional book)	**Intro to Python for Computer Science and Data Science (880-page college textbook)**
 (https://amzn.to/2VvdnxE)	 (https://amzn.to/2LiDCmt)

> Please **do not** purchase both books—our professional book ***Python for Programmers*** is a subset of our college textbook ***Intro to Python for Computer Science and Data Science***

===DUMP FILE===

- Cooking show approach--slicing, dicing, spicing. Show what it looks like at each stage. Use saved models as appropriate.
- Just covered convnet and RNN. No slide was enormous. Used to think about lines of code to get the detailed algorithm right just to build one key aspect of your app. This style of programming is object-based--the vast majority of the code uses other people's objects and sending them method calls. They encapsulate the complexity. Thinking at a much higher level than 20 or even 10 years ago. This is what people love about Python and open source libraries in just about any application field. So powerful. Such an enabler. Enabling you to tackle apps you never would have dreamed of a few years back. Python is now just the glue to weave it all together. Learning process becomes what do I want to develop and what libraries can help me get there. Find tutorials, code, etc. to help you...