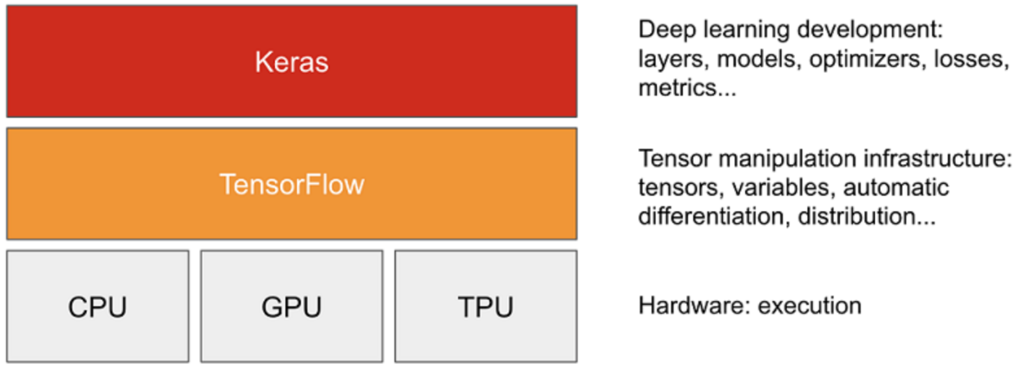
2021/11/24

Neural Networks with Keras

1. TensorFlow and Keras

TensorFlow (tensorflow.org) is a Python-based, free, open-source machine learning platform, developed primarily by Google. Keras is a deep-learning API for Python, built on top of TensorFlow, that provides a convenient way to define and train any kind of deep-learning model. Through TensorFlow, Keras can run on top different types of hardware (see figure below) — GPU, TPU, or plain CPU — and can be seamlessly scaled to thousands of machines.


Keras
TensorFlow
CPU
GPU
TPU
Deep learning development: layers, models, optimizers, losses, metrics...
Tensor manipulation infrastructure: tensors, variables, automatic differentiation, distribution...
Hardware: execution

Since 2019, Keras is now a part of TensorFlow. Installation of TensorFlow is straightforward. Just select tensorflow packages in your Anaconda environments and install the package accordingly.

Remark: Keras is now better maintained and has better integration with TensorFlow features in TensorFlow 2.0. Switching between the codes using old version of Keras and new version of Keras is as simple as changing your import lines with tensorflow preface:

Before

```
from keras... import ... # for TensorFlow 1.x
```

After

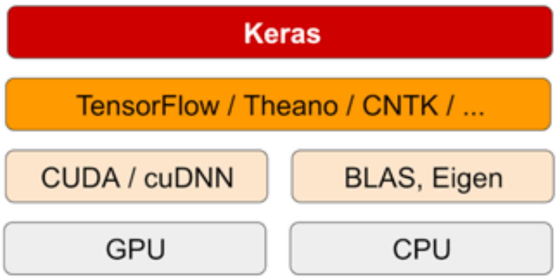
```
from tensorflow.keras... import ... # for TensorFlow 2.x
```

A Brief History of TensorFlow and Keras

Keras predates TensorFlow by eight months. It was released in March 2015, while TensorFlow was released in November 2015. Throughout 2016 and 2017, Keras became well known as the user-friendly way to develop TensorFlow applications, funneling new users into the TensorFlow ecosystem. By late 2017, a majority of TensorFlow users were using it through Keras or in combination with Keras. In 2018, the TensorFlow leadership picked Keras as TensorFlow's official high-level API. As a result, the Keras API is front and center in TensorFlow 2.0, released in September 2019 — an extensive redesign of TensorFlow and Keras that takes into account over four years of user feedback and technical progress.

Remarks:

1. We refer Keras before its merge with TensorFlow the multi-backend Keras.



Multi-backend Keras has been discontinued. It is recommended that Keras users who use multi-backend Keras with the TensorFlow backend switch to tf.keras in TensorFlow 2.0.

2. Although installation of TensorFlow is straightforward, it is sometimes frustrating (and non-trivial) to run it under your hardware or TensorFlow version (2.x). In case you meet some mysterious difficulty, use Google Colab might be a good alternative.

2. Developing with Keras: a quick overview

Keras provides high-level APIs to handle deep learning concept. These are:

- Layers, which are combined into a model
- A loss function, which defines the feedback signal used for learning
- An optimizer, which determines how learning proceeds
- Metrics to evaluate model performance, such as accuracy
- A training loop that performs mini-batch stochastic gradient descent

The typical Keras workflow is very similar to what we have learned in Scikit-Learn. The workflow looks like:

1. Define your training data: input tensors[1] and target tensors.
2. Define a network of layers (or model) that maps your inputs to your targets.
3. Configure the learning process by choosing **a loss function**, **an optimizer**, and **some metrics** to monitor.
4. Iterate on your training data by calling the fit() method of your model.

[1] Tensor is a multidimensional Numpy arrays. Scalar is a 0D tensor, vector is a 1D tensor and matrix is a 2D tensor. Tensors are a generalization to an arbitrary number of dimensions (note that in the context of tensors, a dimension is often called an axis).

5. Evaluate your model performance.

We will now take a look at a concrete example of a neural network, which makes use of the Python library `Keras` to learn to classify hand-written digits, the original MNIST dataset. The problem we are trying to solve here is to classify grayscale images of handwritten digits (28 pixels by 28 pixels, 784 pixels total), into their 10 categories (0 to 9). What we have seen before is a reduced-resolution version of MNIST dataset (8 pixels by 8 pixels, 64 pixels total).

The MNIST dataset is a classic dataset in the machine learning community, which has been around for almost as long as the field itself and has been very intensively studied. It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s.

The MNIST dataset comes pre-loaded in `Keras`, in the form of a set of four `Numpy` arrays:

```
from tensorflow.keras.datasets import mnist

# fix random seed for reproducibility
seed = 42
np.random.seed(seed)
(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()
```

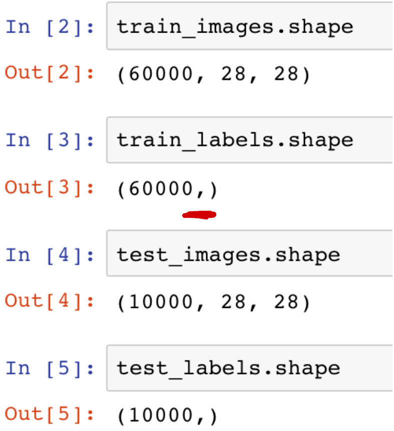
`train_images` and `train_labels` form the training set, the data that the model will learn from. The model will then be tested on the test set, `test_images` and `test_labels`. Our images are encoded as `Numpy` arrays, and the labels are simply an array of digits, ranging from 0 to 9. There is a one-to-one correspondence between the images and the labels.

Let's have a look at the dimensions of training data and testing data:

```
train_images.shape
train_labels.shape
test_images.shape
test_labels.shape
```

(60,000, 28, 28)
(60,000,)

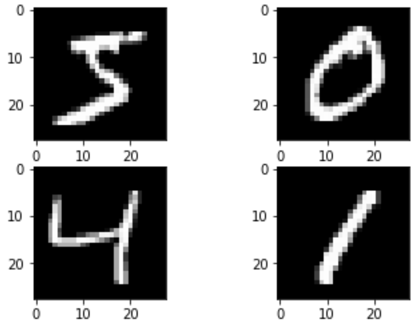
Q: what are the expected outputs?
A:



```
In [2]: train_images.shape
Out[2]: (60000, 28, 28)

In [3]: train_labels.shape
Out[3]: (60000,)

In [4]: test_images.shape
Out[4]: (10000, 28, 28)

In [5]: test_labels.shape
Out[5]: (10000,)
```

We can then plot first four images of the training set:

```
plt.subplot(221)
plt.imshow(train_images[0], cmap=plt.get_cmap('gray')) #5
plt.subplot(222)
plt.imshow(train_images[1], cmap=plt.get_cmap('gray')) #0
plt.subplot(223)
plt.imshow(train_images[2], cmap=plt.get_cmap('gray')) #4
plt.subplot(224)
plt.imshow(train_images[3], cmap=plt.get_cmap('gray')) #1
# show the plot
plt.show()
```



We now ready to go through the Keras workflow:

1. Define your training data: input tensors and target tensors.

(60000, 28,28)

The training dataset is structured as a 3-dimensional array of instance, image width and image height. For a neural network model, we must reduce the images down into a vector of pixels. In this case the 28×28 sized images will be 784 pixel input vectors. We can do this transform easily using the `reshape()` function on the `NumPy` array. The pixel values are integers, so we cast them to floating point values so that we can normalize them easily in the next step.

binary 0
1

gray scale 0.. 255

color scale R 0..255
G 〃
B 〃

The pixel values are gray scale between 0 and 255. It is almost always a good idea to perform some scaling of input values when using neural network models. Because the scale is well known and well behaved, we can very quickly normalize the pixel values to the range 0 and 1 by dividing each value by the maximum of 255.

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Finally, the output variable is an integer from 0 to 9. This is a multiclass classification problem. As such, it is good practice to use a **one hot encoding** of the class values, transforming the vector of class integers into a binary matrix. We can easily do this using the `utils.to_categorical()` helper function in `Keras`.

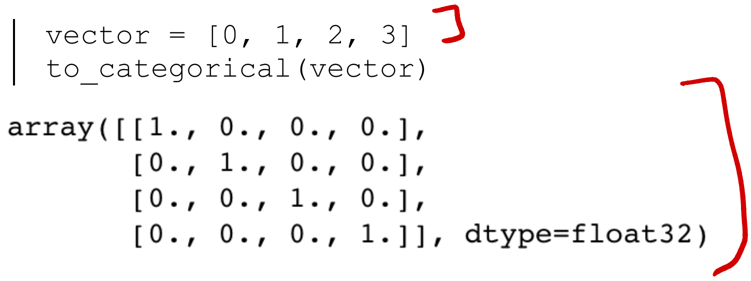
```
from tensorflow.keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Remark: One hot encoding

One hot encoding is a proven technique typically used in machine learning for categorical features (and targets). For example, a person could have features `["male", "female"]`, `["from Europe", "from US", "from Asia"]`, `["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"]`. Such features can be efficiently coded as integers, for instance `["male", "from US", "uses Internet Explorer"]` could be expressed as `[0, 1, 3]` while `["female", "from Asia", "uses Chrome"]` would be `[1, 2, 1]`.

Q: what is wrong with this approach?
A:
Such integer representation can, however, not be used directly with all `Scikit-Learn` and `Keras` estimators, as these expect continuous input, and would **interpret the categories as being ordered, which is often not desired**. Such a mapping would imply, for example, that `"male" < "female"` or even that `"uses Internet Explorer" - "uses Safari" = "uses Chrome"`, which does not make much sense.

One-hot encoding encode categorical integer features as a one-hot numeric array. It transforms each categorical feature with `n_categories` possible values into `n_categories` binary features, with one of them 1, and all others 0. For example:

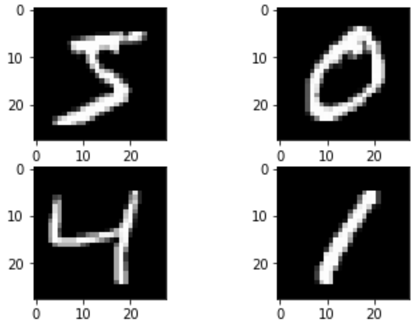


```
vector = [0, 1, 2, 3]
to_categorical(vector)
```

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

Q: Recall the labels for first four samples in the MNIST dataset is [5, 0, 4, 1]. What happens when we apply one hot encoding for the data?

```
train_labels = to_categorical(train_labels)
train_labels[:4]
```



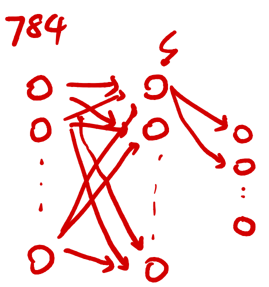
A:

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

2. **Define a network of layers (or model) that maps your inputs to your targets.**
3. **Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.**

We are now ready to create our simple neural network model. We will define our model in a function. This is handy if you want to extend the example later and try and get a better score.

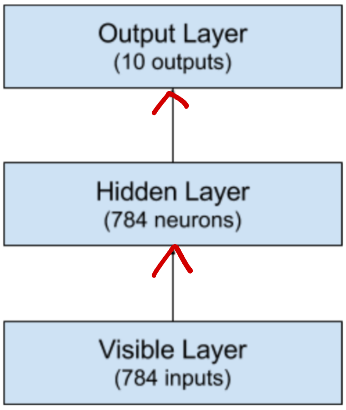
```
from tensorflow.keras import models
from tensorflow.keras import layers

def mlp_model():
# create mlp model
    model = models.Sequential()
    num_pixels = 28*28
    num_classes = 10
    model.add(layers.Dense(num_pixels, activation='relu',
input_shape=(num_pixels,)))
    model.add(layers.Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```



```
network = mlp_model()
```

There are two ways to define a model in `Keras`: using the `Sequential` class (only for **linear stacks of layers**, which is the most common network architecture by far) or the functional API (for directed acyclic graphs of layers, which lets you build completely arbitrary architectures).

The model herein is a simple neural network with one hidden layer with the same number of neurons as there are inputs (784). **A rectifier activation function** (ReLU) is used for the neurons in the hidden layer. **A softmax activation function** is used on the output layer to turn the outputs into probability-like values and allow one class of the 10 to be selected as the model's output prediction. Logarithmic loss is used as the loss function (called `categorical_crossentropy` in `Keras`). We will discuss these activation functions, loss and optimizer in the following. A summary of the network structure is provided below:



output

hidden

input

$(784+1)\times 784 + (784+1)\times 10$

$= 623290$

#073374

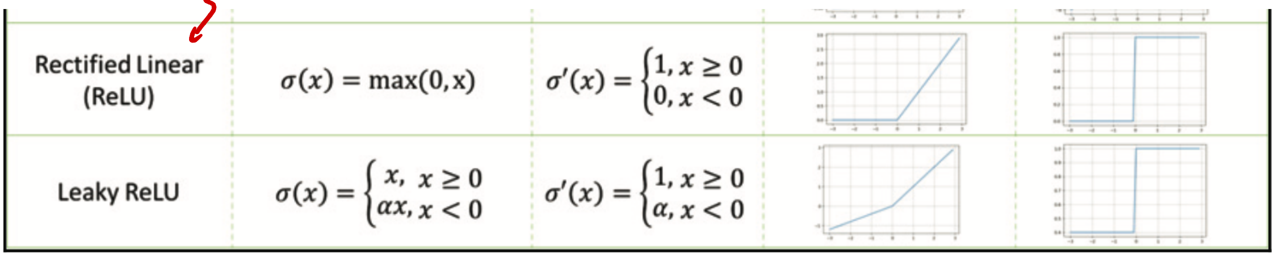
Fun Time: How many training parameters for this network model? (1) 784 (2) 622496 (3) 623290

The learning process is configured in the compilation step, where you specify the optimizer and loss function(s) that the model should use, as well as the metrics you want to monitor during training.

Remark: Activation functions

Rectified Linear Unit (RELU or ReLU)

hidden

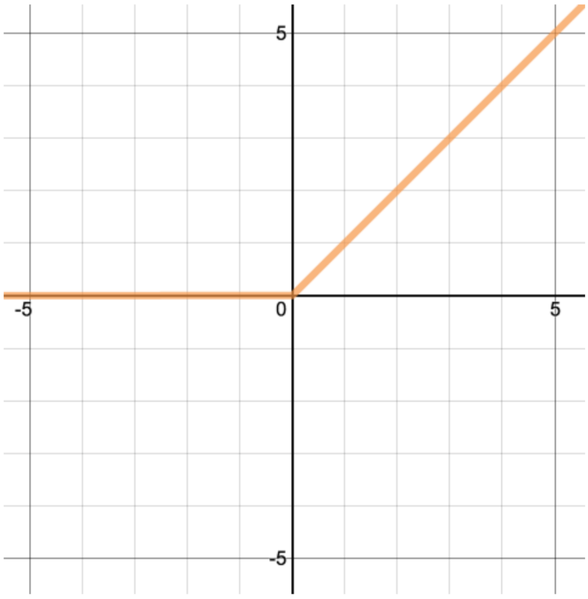


Rectified Linear (ReLU)	$\sigma(x) = \max(0,x)$	$\sigma'(x) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases}$		
Leaky ReLU	$\sigma(x) = \begin{cases} x, \ x \geq 0 \\ \alpha x, x < 0 \end{cases}$	$\sigma'(x) = \begin{cases} 1, x \geq 0 \\ \alpha, x < 0 \end{cases}$		

A rectified linear unit is defined as:

$$\sigma(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

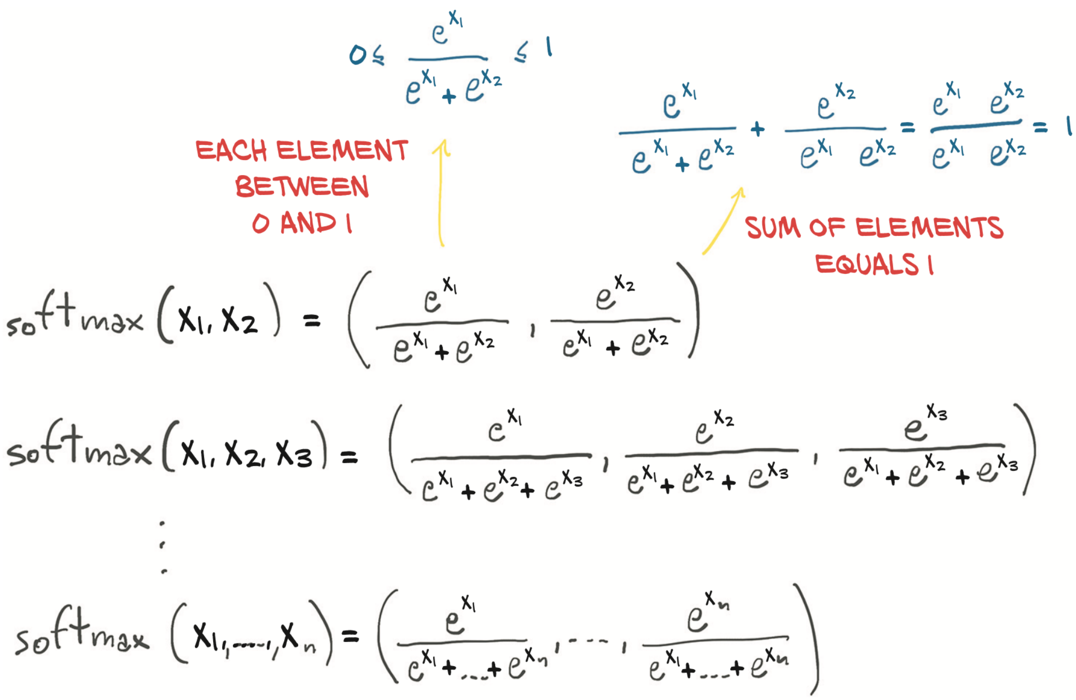
Rectified linear unit (RELU) is a more interesting transform that activates a node only if the input is above a certain quantity. While the input is below zero, the output is zero, but when the input rises above a certain threshold, it has a linear relationship with the dependent variable x, as demonstrated in the figure below.



RELU is a typical choice **for hidden layers** because they have proven to work in many different situations. RELU activation functions have shown to train better in practice than sigmoid activation functions such as `tanh`.

Softmax

Softmax is a generalization of sigmoidal function as illustrated in the figure below. It takes a vector of values and produces another vector of the same dimension, where the values are probability distribution over mutually exclusive output classes.


$0 \leq \frac{e^{x_1}}{e^{x_1}+e^{x_2}} \leq 1$
EACH ELEMENT BETWEEN 0 AND 1
$\frac{e^{x_1}}{e^{x_1}+e^{x_2}} + \frac{e^{x_2}}{e^{x_1}\ e^{x_2}} = \frac{e^{x_1}\ e^{x_2}}{e^{x_1}\ e^{x_2}} = 1$
SUM OF ELEMENTS EQUALS 1
$\text{softmax}(x_1, x_2) = \left(\frac{e^{x_1}}{e^{x_1}+e^{x_2}}, \frac{e^{x_2}}{e^{x_1}+e^{x_2}}\right)$
$\text{softmax}(x_1, x_2, x_3) = \left(\frac{e^{x_1}}{e^{x_1}+e^{x_2}+e^{x_3}}, \frac{e^{x_2}}{e^{x_1}+e^{x_2}+e^{x_3}}, \frac{e^{x_3}}{e^{x_1}+e^{x_2}+e^{x_3}}\right)$
$\vdots$
$\text{softmax}(x_1, \ldots, x_n) = \left(\frac{e^{x_1}}{e^{x_1}+\ldots+e^{x_n}}, \ldots, \frac{e^{x_n}}{e^{x_1}+\ldots+e^{x_n}}\right)$

Softmax is the function you will often find at **the output layer** of a classifier. If we have a multiclass modeling problem yet we care only about the best score across these classes, we'd use a softmax output layer to get the highest score of all the classes.

Remark: loss function

The `categorical_crossentropy` belongs to the cross entropy loss function, a logarithmic loss function that's used as a feedback signal for learning the weight tensors, and which the training phase will attempt to minimize. For binary classification, this loss function of a real-valued probability prediction $\hat{y} \in (0,1)$ on a data sample with binary label $y \in (0,1)$ is defined as:

$$Loss(y, \hat{y}) = -y\ln(\hat{y}) - (1-y)\ln(1-\hat{y})$$

The cross entropy loss function has foundations in information theory and measures the amount of disagreement between y and $\hat{y}$.

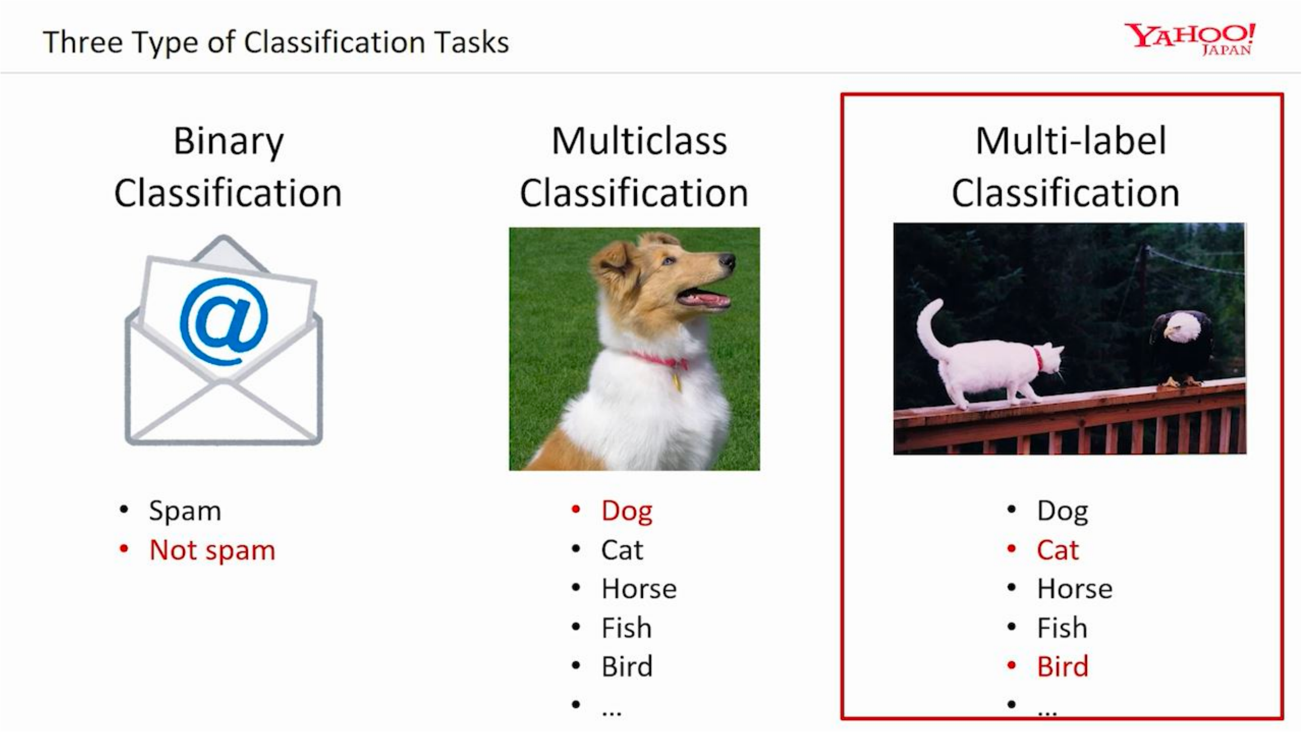
generalization training

Note that it isn't always possible to directly optimize for the metric that measures success on a problem. Sometimes there is no easy way to turn a metric into a loss function.

For instance, the widely used classification metric `ROC AUC` can't be directly optimized. Hence, in classification tasks, it's common to optimize for a proxy metric of `ROC AUC`, such as `crossentropy`. In general, you can hope that the lower the `crossentropy` gets, the higher the `ROC AUC` will be.

Table below can help you choose a last-layer activation and a loss function for a few common problem types.

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`



Three Type of Classification Tasks

YAHOO! JAPAN

Binary Classification

- Spam
- Not spam

Multiclass Classification

- Dog
- Cat
- Horse
- Fish
- Bird
- …

Multi-label Classification

- Dog
- Cat
- Horse
- Fish
- Bird
- …

Remark: optimizer

The reduction of the loss happens via **gradient descent**. A lot of variations to gradient descent have been used over the years. SGD (Stochastic Gradient Descent), RMSprop and Adam are three popular optimizers. The gist of RMSprop is to maintain a moving (discounted) average of the square of gradients and divide the gradient by the root of this average. Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. Adam is computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters".

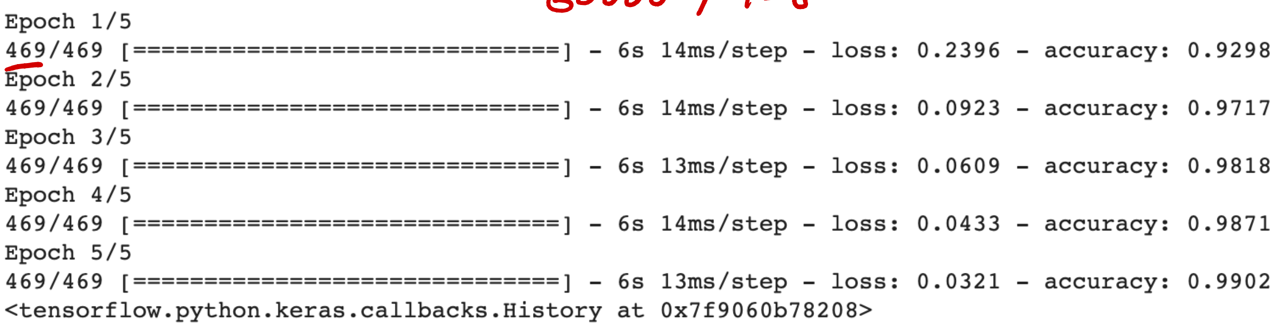
The exact rules (for example, learning rate) governing a specific use of gradient descent are defined by the `rmsprop` optimizer passed as the first argument.

4. **Iterate on your training data by calling the `fit()` method of your model.**
5. **Evaluate your model performance.**

We can now `fit` and `evaluate` the model. The model is `fit` over 5 epochs with updates every 128 images. The test data is used as the validation dataset, allowing you to see the skill of the model as it trains. Finally, the test dataset is used to evaluate the model and a classification score is printed.

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

60000 / 128

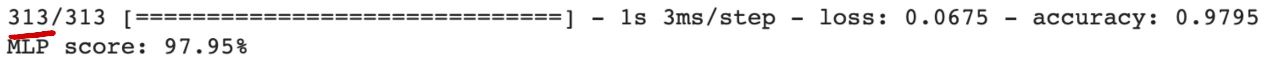


```
Epoch 1/5
469/469 [==============================] - 6s 14ms/step - loss: 0.2396 - accuracy: 0.9298
Epoch 2/5
469/469 [==============================] - 6s 14ms/step - loss: 0.0923 - accuracy: 0.9717
Epoch 3/5
469/469 [==============================] - 6s 13ms/step - loss: 0.0609 - accuracy: 0.9818
Epoch 4/5
469/469 [==============================] - 6s 14ms/step - loss: 0.0433 - accuracy: 0.9871
Epoch 5/5
469/469 [==============================] - 6s 13ms/step - loss: 0.0321 - accuracy: 0.9902
<tensorflow.python.keras.callbacks.History at 0x7f9060b78208>
```

```
# Final evaluation of the model
scores = network.evaluate(test_images, test_labels)
print("MLP score: %.2f%%" % (scores[1]*100))
```

feed forward

10000 / 32



```
313/313 [==============================] - 1s 3ms/step - loss: 0.0675 - accuracy: 0.9795
MLP score: 97.95%
```

When you call `fit`: the network will start to iterate on the training data in mini-batches of 128 samples, 5 times over (**each iteration over all the training data is called an epoch**). At each iteration, the network will compute the gradients of the weights with regard to the loss on the batch, and update the weights accordingly. For each epoch, we have total $60000/128 \approx 469$ iterations.

After these 5 epochs, the loss of the network will be sufficiently low that the network will be capable of classifying handwritten digits with high accuracy.

We then use `evaluate` to evaluate the performance of the model on the test data. If not specified, the mini-batch size is 32 samples. We thus have $10000/32 \approx 313$ for this forward pass.

Remark: as randomness involves in stochastic mini-batch selection, your answers might be slightly different from mine.

You can download the above code `Keras_MNIST.ipynb` from course website.