Machine Learning

Lecture 7: Neural Networks and Keras

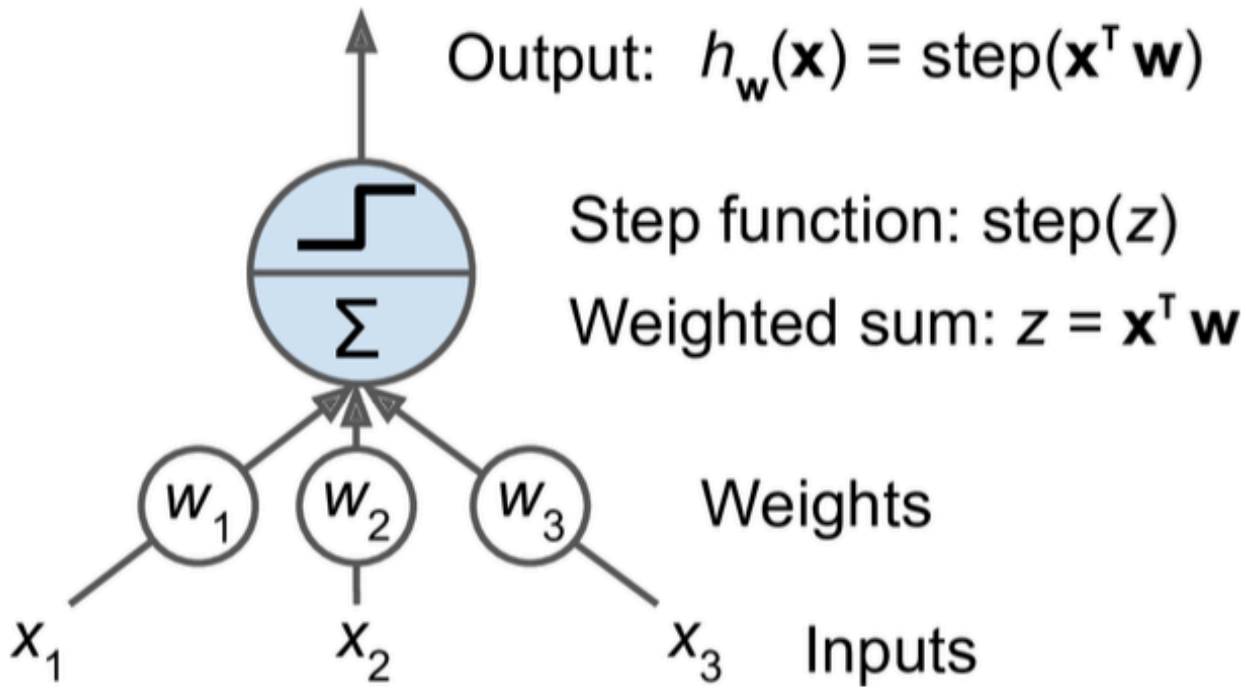
Instructor: Dr. Farhad Pourkamali Anaraki

Introduction

- We start this discussion by introducing Artificial Neural Network (ANN) architectures and Multilayer Perceptrons (MLPs)
    - History: ANNs gained popularity in 1940s and 1980s
- Why did deep learning become so popular again?
    - Massive amounts of data
    - Increase in computing power
    - Better understanding of the optimization landscape
- We discuss implementing neural networks using the Keras API
    - Simple high-level API for building and training neural networks
    - Keras is flexible enough to build various neural network architectures
    - Write custom Keras components using its lower-level API for extra flexibility

Threshold logic unit (TLU)

- TLU computes a weighted sum of its inputs and then applies a **step function**



- Common step functions

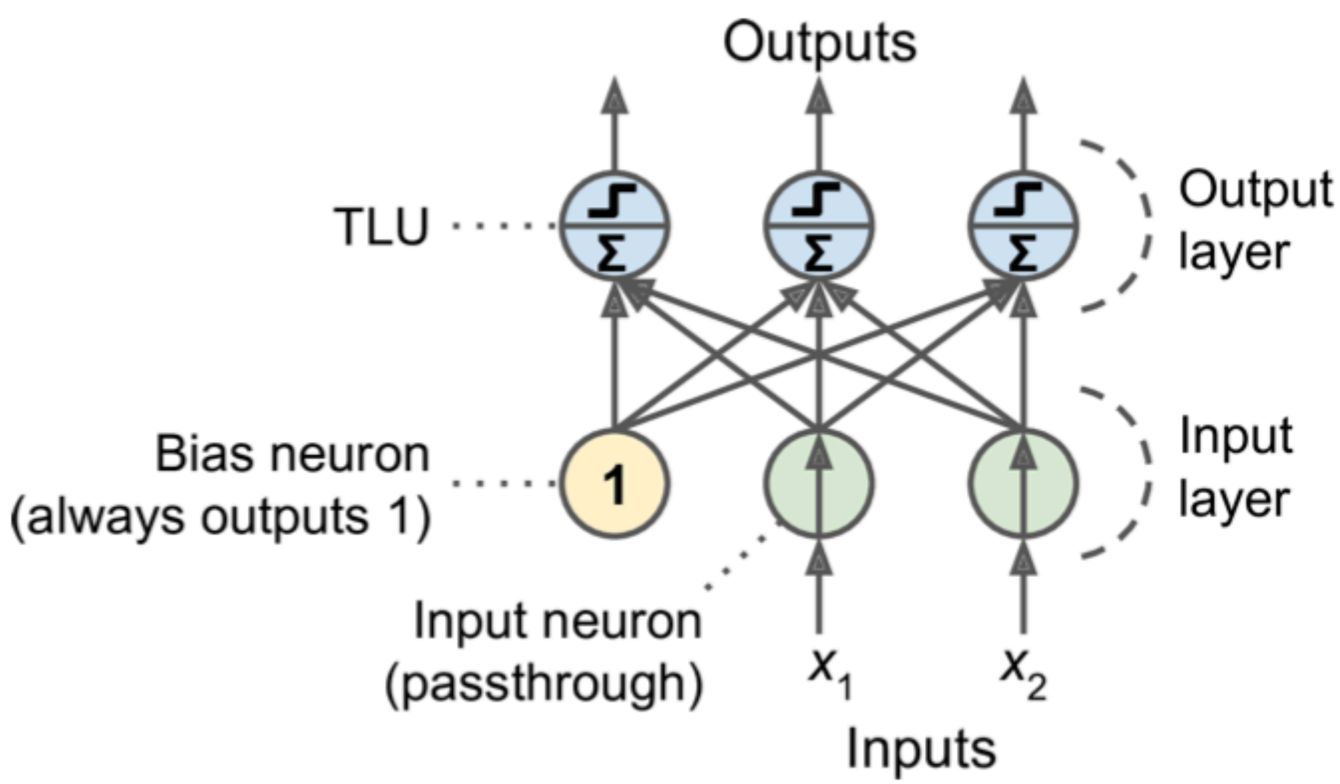
$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

- Just like logistic regression and linear SVM

Perceptron

- A perceptron is composed of a single layer of TLUs, with each TLU connected to all the inputs (neurons in the previous layer)

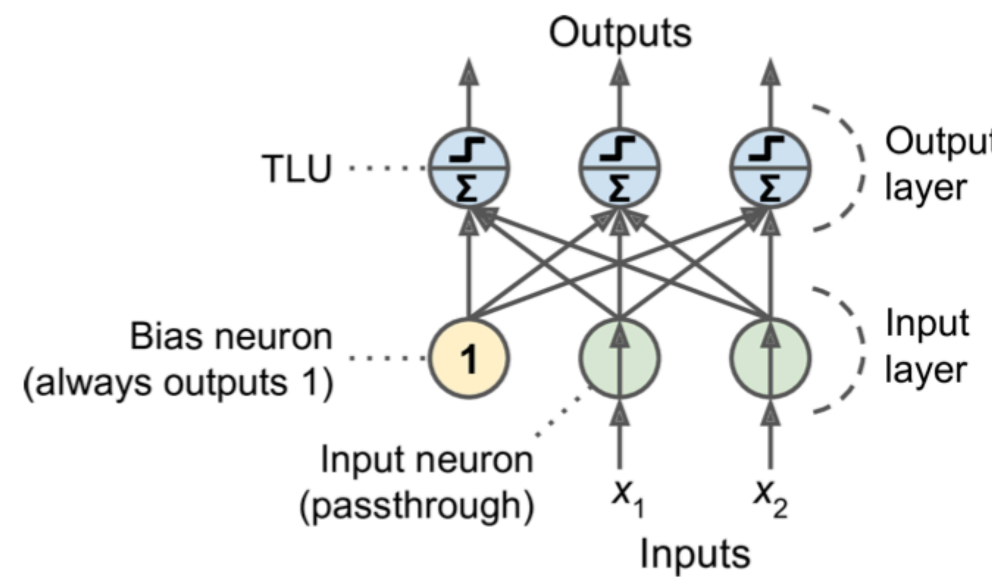
- Example: 2 input neurons, 1 bias neuron, and 3 output neurons



- Called fully connected or dense layer

Computing the outputs

- We can find a compact representation of outputs using linear algebra
  - Number of instances: n, number of features: d
  - Number of neurons: m

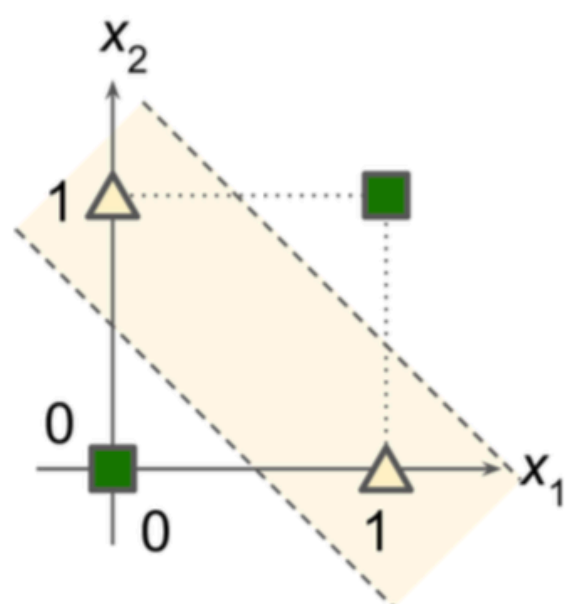


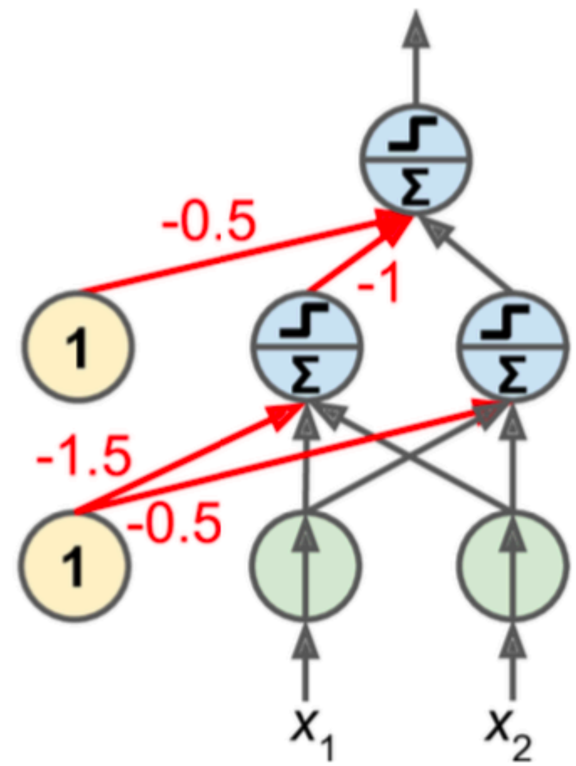
$$h(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

- Matrix of input features $\mathbf{X} \in \mathbb{R}^{n \times d}$
- Matrix of connection weights $\mathbf{W} \in \mathbb{R}^{d \times m}$
- Bias vector $\mathbf{b}$: one bias term per artificial neuron
- Activation or step function ϕ

Multilayer Perceptrons (MLPs)

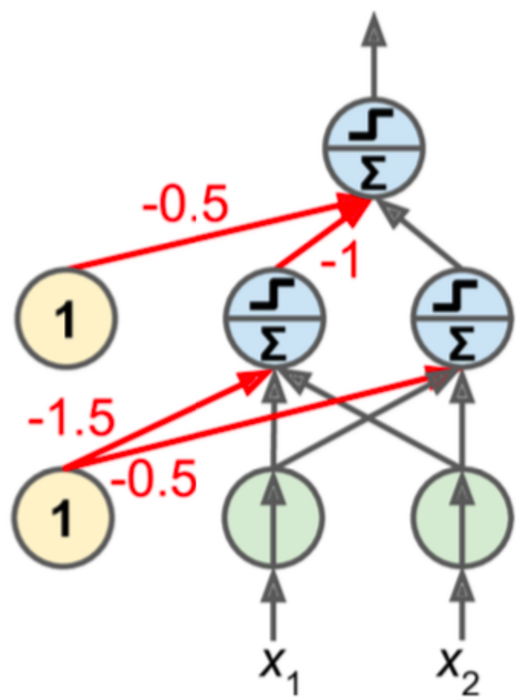
- Some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons
- For example, the following network solves the Exclusive OR (XOR) problem



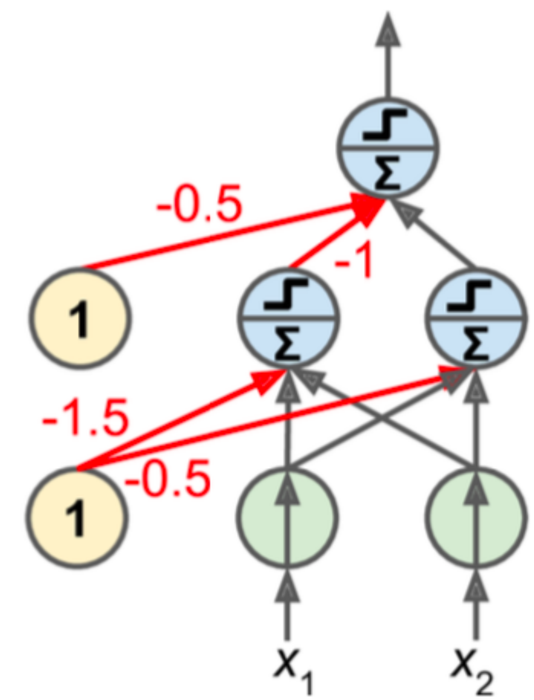


Other weights equal to 1

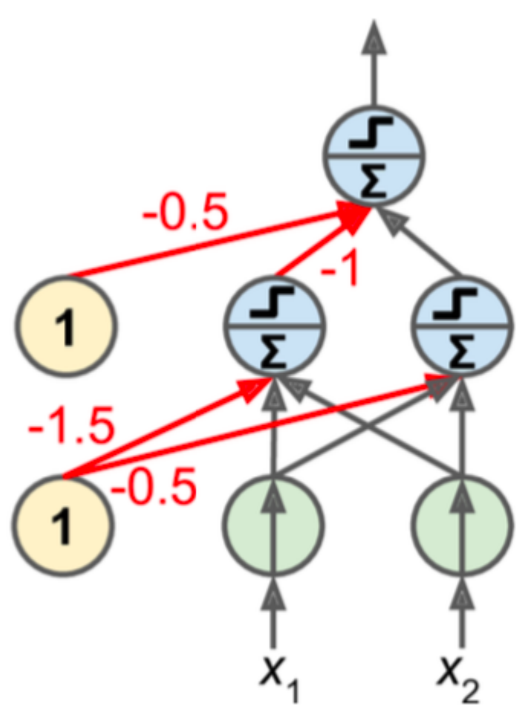
Solving XOR



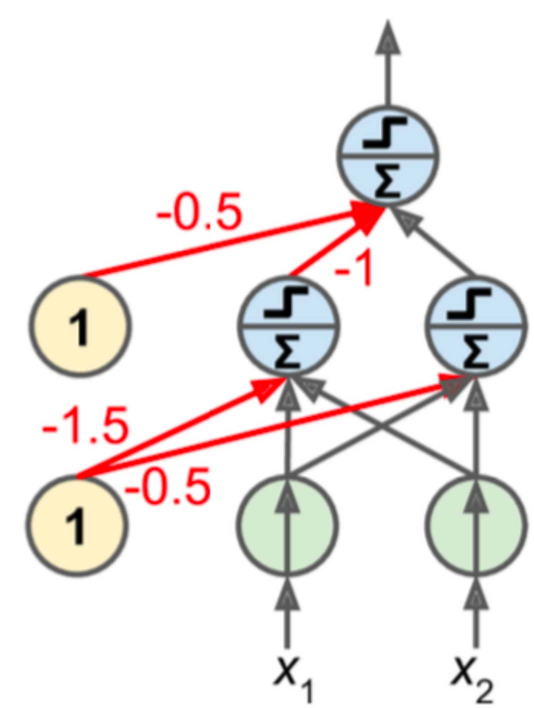
(0,0)



(0,1)



(1,0)

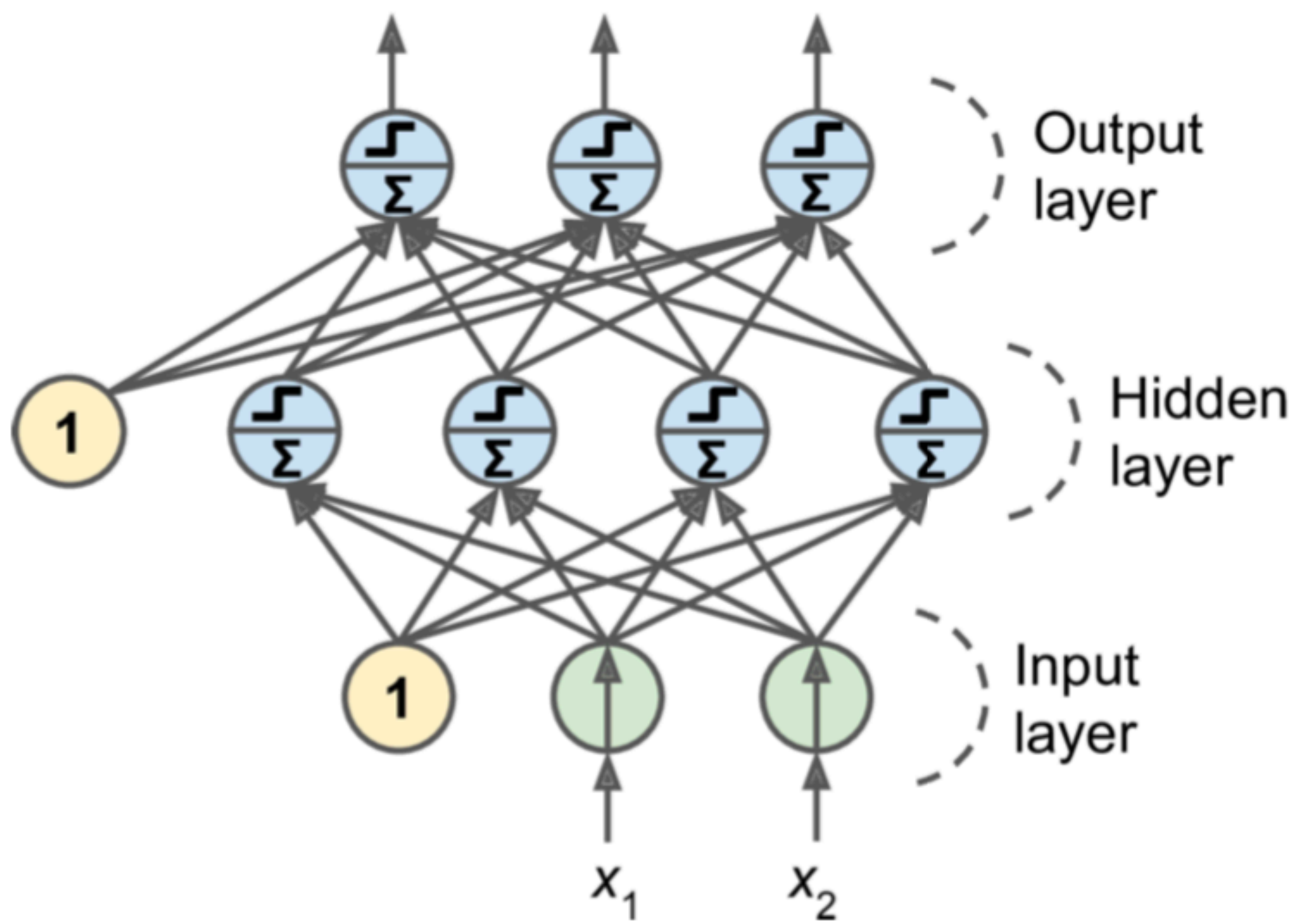


(1,1)

Architecture of Multilayer Perceptrons

- MLP is composed of one input layer, one or more layers of TLUs (called hidden layers), and one final layer of TLUs (called output layer)

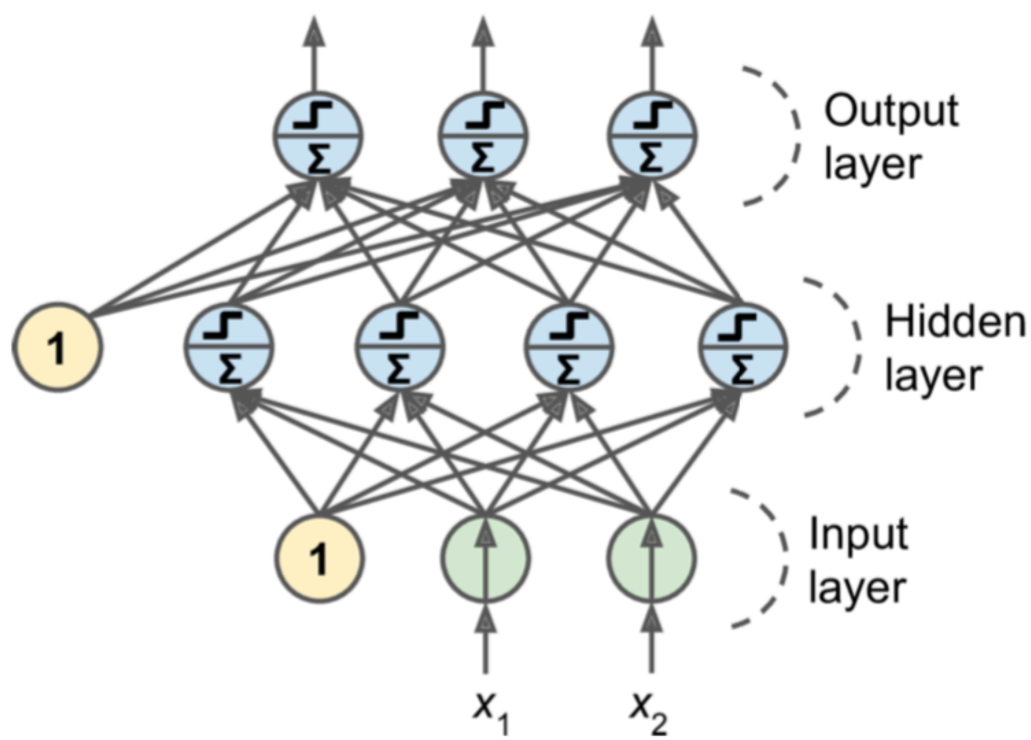
- Every layer except the output layer includes a bias neuron and is fully connected



- Input layer: 2 neurons, hidden layer: 4 neurons, and output layer: 3 neurons

Terminology

- Features flow in one direction from the inputs to the outputs, so the architecture in the previous slide is an example of **feedforward neural networks**



- When an ANN contains a deep stack of hidden layers (e.g., tens or hundreds of layers), it is called a deep neural network (DNN)

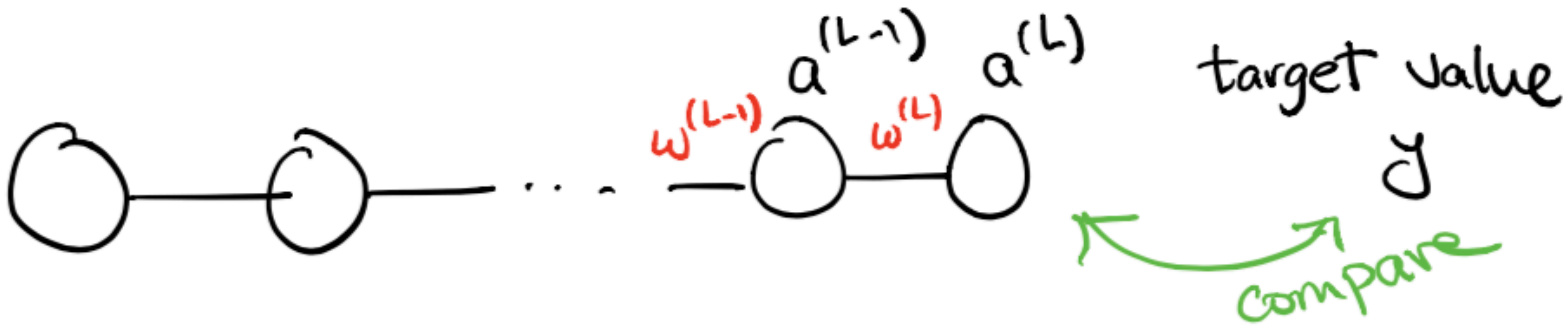
- Some people refer to networks with one or two hidden layers as Deep Learning

Training MLPs and backpropagation

- We need to find out how each connection weight and bias term should be optimized in order to reduce the error
- We can use Gradient Descent and its variants if we compute the gradients
- General approach:
    - For each training instance, we first make a prediction (forward pass)
    - Measure the output error using a loss function
    - Go through each layer in reverse to measure the error contribution from each connection (reverse pass)
    - Update the connection weights to reduce the error
- We apply the chain rule in the backpropagation algorithm
- Note that we need to initialize connection weights randomly

Example

- Consider a network with one neuron per layer



$$C = \left(a^{(L)} - y\right)^2, \quad a^{(L)} = \phi\left(z^{(L)}\right), \quad z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

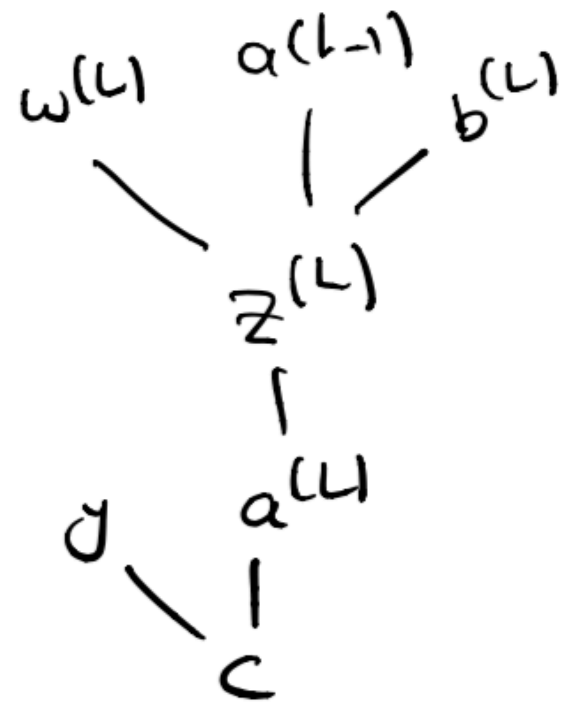
$$\frac{\partial C}{\partial w^{(L)}} = \underbrace{\frac{\partial C}{\partial a^{(L)}}}_{2\left(a^{(L)} - y\right)} \cdot \underbrace{\frac{\partial a^{(L)}}{\partial z^{(L)}}}_{\phi'\left(z^{(L)}\right)} \cdot \underbrace{\frac{\partial z^{(L)}}{\partial w^{(L)}}}_{a^{(L-1)}}$$

Example

$$\frac{\partial C}{\partial w^{(L)}} = \underbrace{\frac{\partial C}{\partial a^{(L)}}}_{2(a^{(L)}-y)} \cdot \underbrace{\frac{\partial a^{(L)}}{\partial z^{(L)}}}_{\phi'(z^{(L)})} \cdot \underbrace{\frac{\partial z^{(L)}}{\partial w^{(L)}}}_{a^{(L-1)}}$$

$$\frac{\partial C}{\partial b^{(L)}} = \frac{\partial C}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \underbrace{\frac{\partial z^{(L)}}{\partial b^{(L)}}}_{=1}$$

$$\frac{\partial C}{\partial a^{(L-1)}} = \frac{\partial C}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \underbrace{\frac{\partial z^{(L)}}{\partial a^{(L-1)}}}_{w^{(L)}}$$

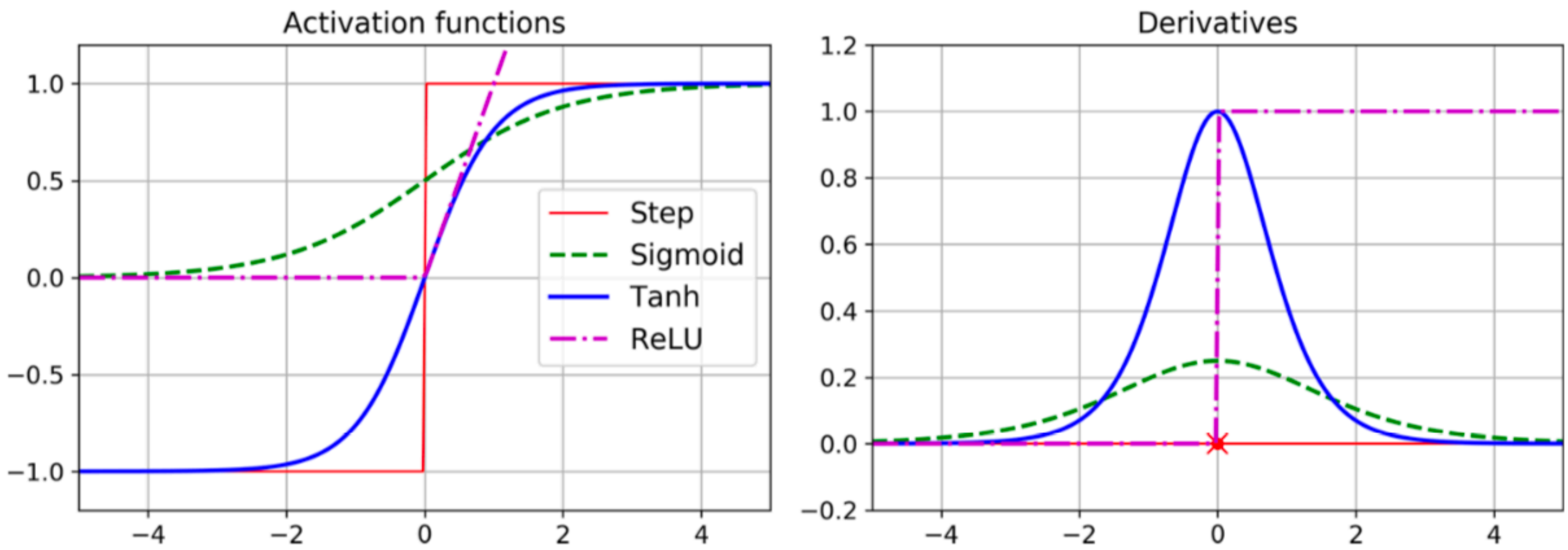


computation graph

Activation functions

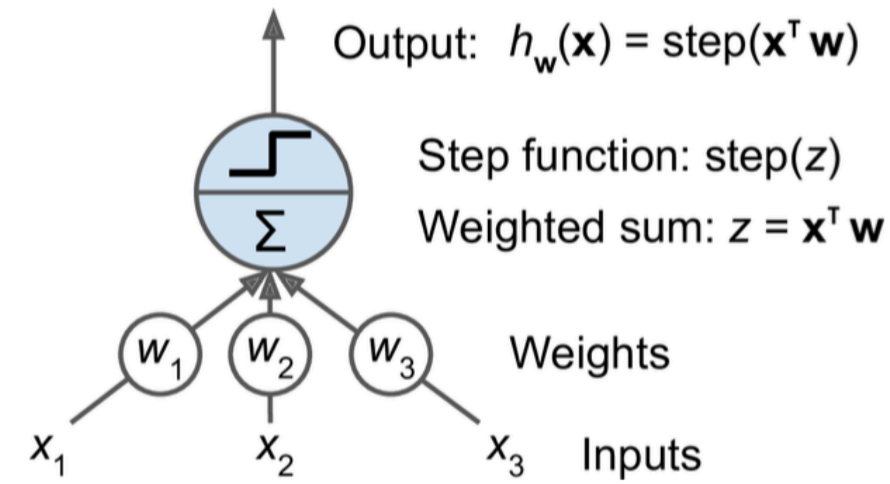
- Replaced the step function with the logistic function $\sigma(z) = \dfrac{1}{1 + \exp(-z)}$
  - Advantage: well-defined non-zero derivative everywhere
- Two other choices:
  - Hyperbolic tangent function $tanh(z) = 2\sigma(2z) - 1$
    - The output value ranges from -1 to 1
    - Each layer's output more or less centered around 0
  - Rectified Linear Unit function $ReLU(z) = \max(0,z)$
    - Continuous but not differentiable at $z = 0$
    - The default activation function because it works very well

Activation functions and their derivatives



Why activation functions?

- Why do we need activation functions in the first place?
    - Example: Consider two linear transformations
        - $f(x) = 2x + 3$
        - $g(x) = 5x - 1$
        - $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$
    - If we don't have some form of non-linearity between layers, then the network is equivalent to a single layer network, incapable of solving complex problems



- On the other hand, a large enough deep neural network with non-linear activations can **theoretically** approximate any continuous function

Regression MLPs

- For multivariate regression (i.e., predicting multiple values), you need one output neuron per output dimension

- We typically don't use any activation function for the output neurons so they are free to produce any range of values

  - If the output is always positive, you can use ReLU

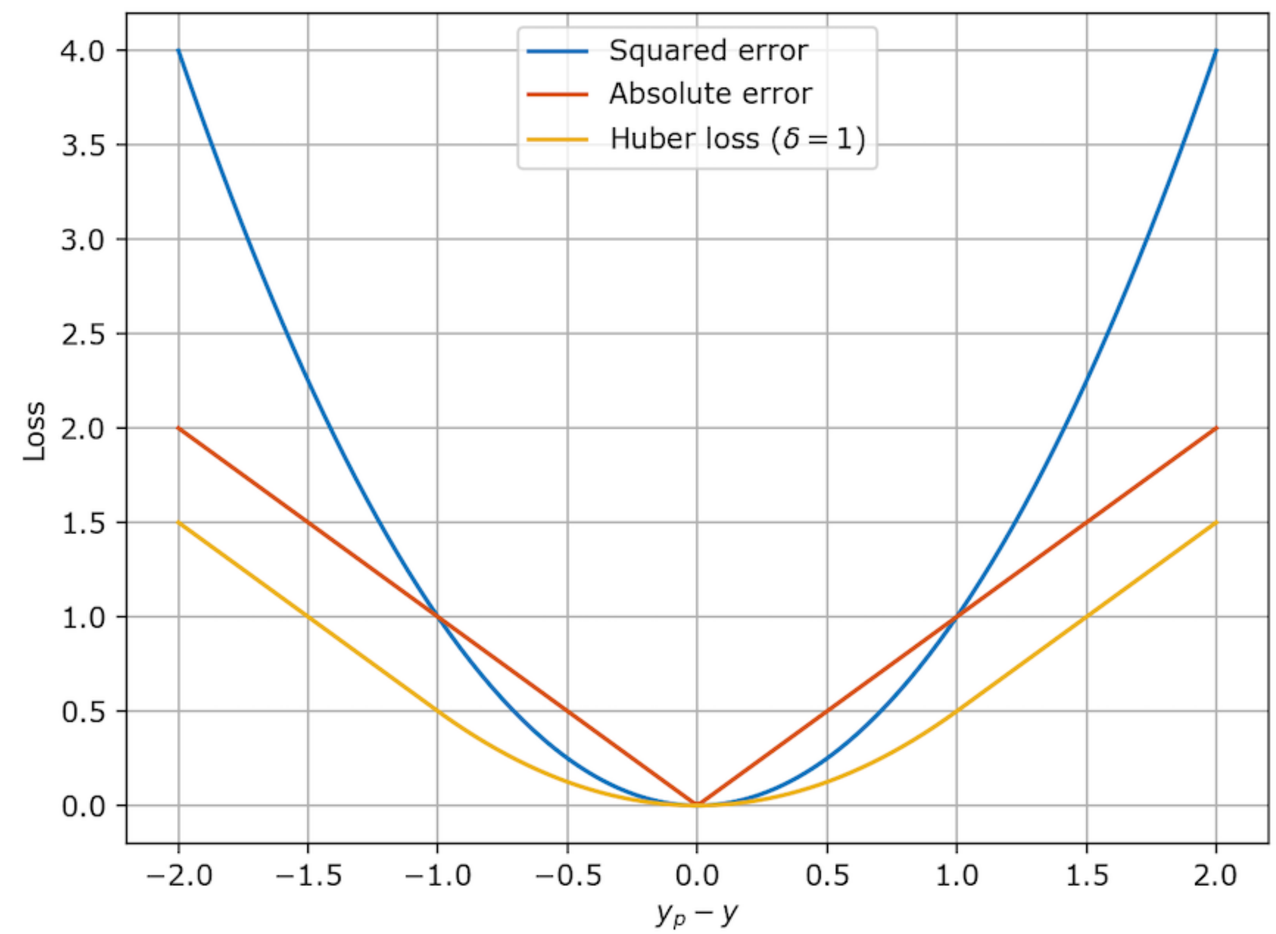
  - If the output falls within a range of values, you can use logistic function and scale the labels to the range 0 to 1

- Loss functions:

  - Mean squared error (MSE)

  - Mean absolute error (MAE)

  - Huber loss



Classification MLPs

- For binary classification tasks, you just need a single output neuron with the logistic activation function

- If each instance belongs only to a single class, then we need one neuron per class with the softmax activation function

  - Example, classes 0 through 9 for digit image classification

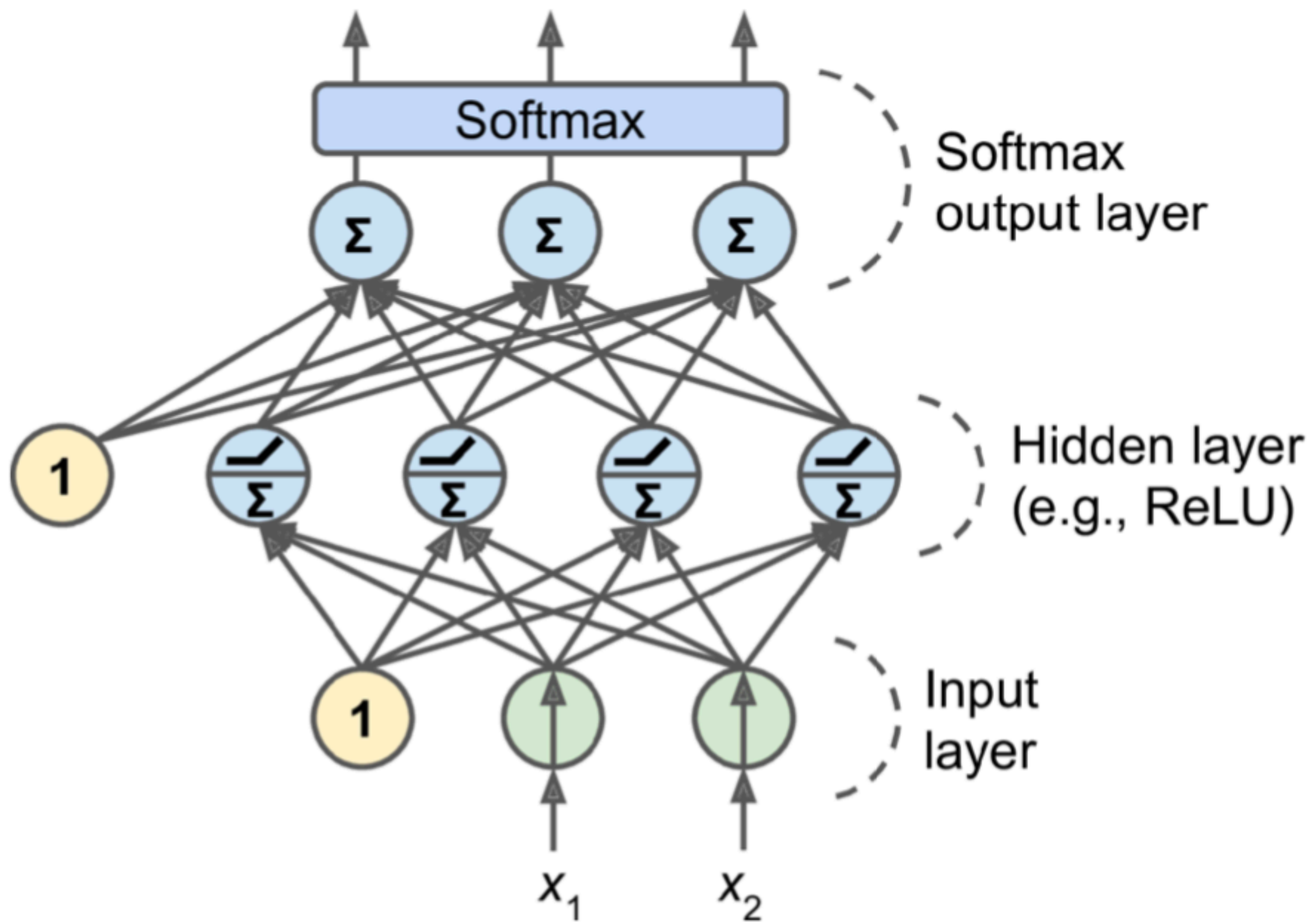
- Loss function: we use the cross-entropy loss because the output layer is predicting probability distributions

$$-\sum_{c=1}^{C} y_c \log(p_c)$$

  - y_c: binary indicator (0 or 1)

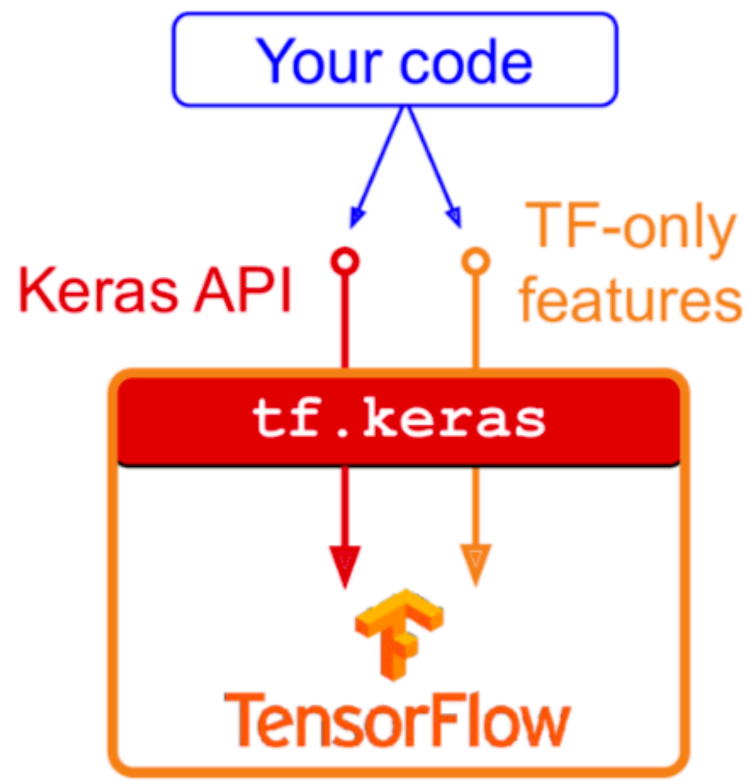
  - p_c: predicted probabilities

A modern MLP architecture for classification



Implementing MLPs with Keras

- Keras is a high-level deep learning API: https://keras.io/
- TensorFlow comes bundled with its own Keras implementation tf.keras
  - It supports TensorFlow's Data API (load an preprocess data)



- Another popular library is PyTorch (quite similar to Keras and inspired by sklearn)

Installing TensorFlow

- To test your installation, open a Python shell or Jupyter notebook

```
import tensorflow as tf
```

```
from tensorflow import keras
```

```
tf.__version__
```

```
'2.3.0'
```

```
keras.__version__
```

```
'2.4.0'
```

Building an image classifier

- We work with the Fashion MNIST data set
    - 70,000 grayscale images of 28x28 pixels with 10 classes


```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

```
print(X_train_full.shape, np.unique(y_train_full))
```

```
(60000, 28, 28) [0 1 2 3 4 5 6 7 8 9]
```

Loading data

- When loading Fashion MNIST using Keras, the pixel intensities are represented as integers from 0 to 255

```
X_train_full.dtype
```

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

- We scale the pixel intensities to the range 0-1 range and also generate a validation set

```
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.
```

```
print(X_train.shape, X_train.dtype)
```

```
(55000, 28, 28) float64
```

- List of class names

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

Visualizing data

- Let's look at the first image in the training set and its label

```
plt.imshow(X_train[0], cmap="binary")
plt.axis('off')
plt.show()
```

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

```
'Coat'
```

Creating the model using the Sequential API

- Classification MLP with two hidden layers

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

- “Flatten layer” converts each input image into a 1D array
  - No parameters because it only does preprocessing
  - You should specify the “input_shape” because it is the first layer in the model
- “Dense” hidden layers with 300 and 100 neurons with ReLU activation functions
- “Dense” output layer with 10 neurons (one per class) with softmax

Easier way to define model

- We can pass a list of layers when creating the sequential model

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

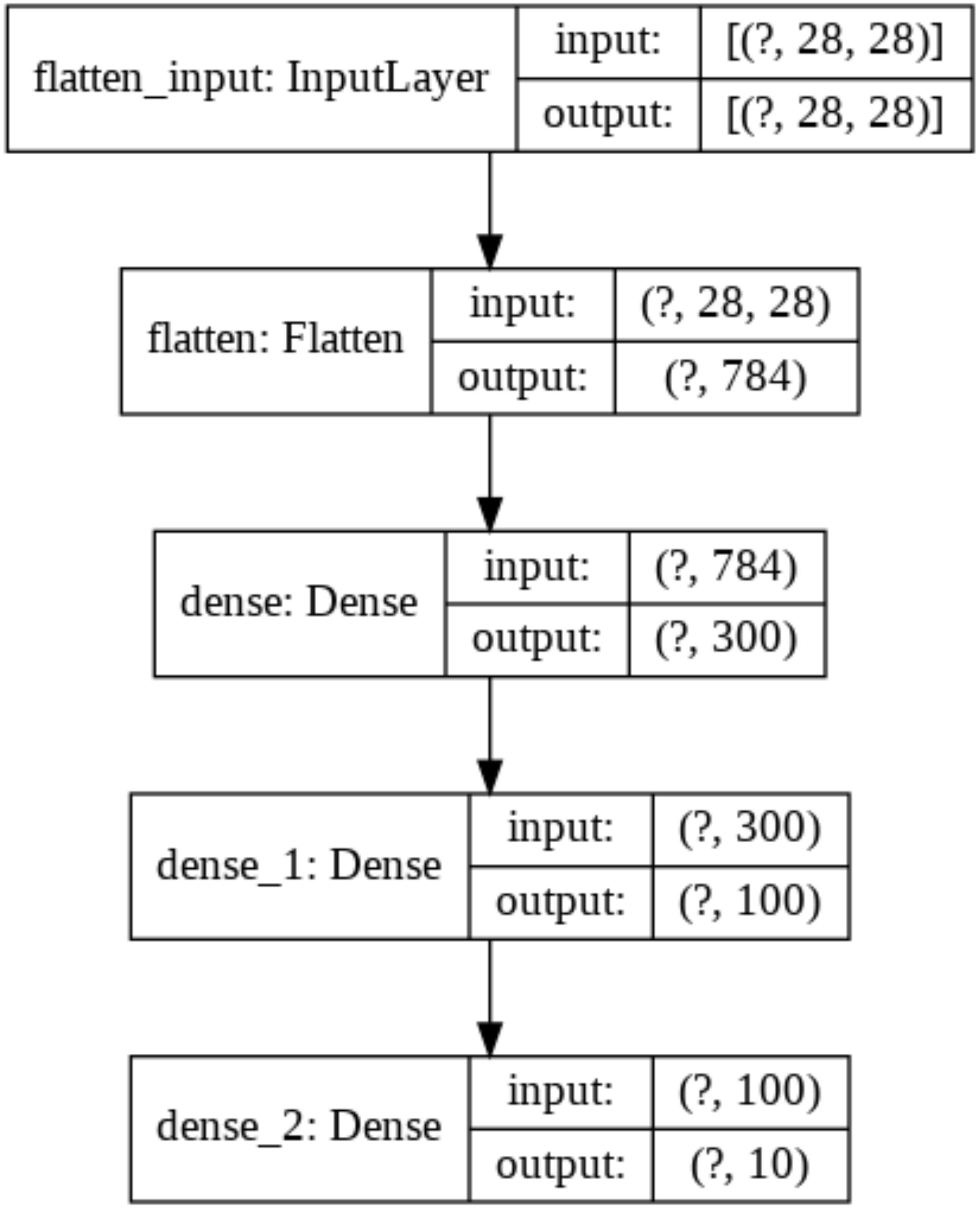
```
model.summary()
```

$784 \times 300 + 300 = 785 \times 300 = 235{,}500$

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Visualizing the model

```
keras.utils.plot_model(model, "my_fashion_mnist_model.png", show_shapes=True)
```



Accessing parameters for each layer

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

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

```
weights
```

Weights are initialized randomly and biases are initialized to zeros
https://keras.io/api/layers/initializers/

```
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
         0.03859074, -0.06889391],
       [ 0.00476504, -0.03105379, -0.0586676 , ...,  0.00602964,
        -0.02763776, -0.04165364],
       [-0.06189284, -0.06901957,  0.07102345, ..., -0.04238207,
         0.07121518, -0.07331658],
       ...,
       [-0.03048757,  0.02155137, -0.05400612, ..., -0.00113463,
         0.00228987,  0.05581069],
       [ 0.07061854, -0.06960931,  0.07038955, ..., -0.00384101,
         0.00034875,  0.02878492],
       [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
         0.00272203, -0.06793761]], dtype=float32)
```

```
print(weights.shape, biases.shape)
```

```
(784, 300) (300,)
```

Compiling the model

- After a model is created, you must call its compile() method to specify the loss function and the optimizer to use
  - Additionally, specify a list of extra metrics to compute

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

- Why do we use "sparse" categorical cross entropy loss?
  - Labels are provided as integers
  - If labels are provided using one-hot representation, then we use categorical cross entropy loss

```
[0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]
```

Training and evaluating the model

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

- Keras will measure the loss and the extra metrics at the end of each epoch

```
loss: 0.7237 - accuracy: 0.7643 - val_loss: 0.5213 - val_accuracy: 0.8226

loss: 0.4842 - accuracy: 0.8316 - val_loss: 0.4349 - val_accuracy: 0.8528

loss: 0.4391 - accuracy: 0.8456 - val_loss: 0.5331 - val_accuracy: 0.7986

loss: 0.4123 - accuracy: 0.8565 - val_loss: 0.3916 - val_accuracy: 0.8654

loss: 0.3937 - accuracy: 0.8621 - val_loss: 0.3740 - val_accuracy: 0.8698
```

- If the performance on the training set is much better than on the validation set, the model is probably overfitting

After training

- The returned "history" object contains
  - Training parameters: history.params
  - List of epochs: history.epoch
  - Dictionary including the loss and extra metrics: history.history

```
history.params
```

```
{'epochs': 30, 'steps': 1719, 'verbose': 1}
```

```
history.epoch
```

```
[0,
 1,
 2,
```

0,1,…, 29

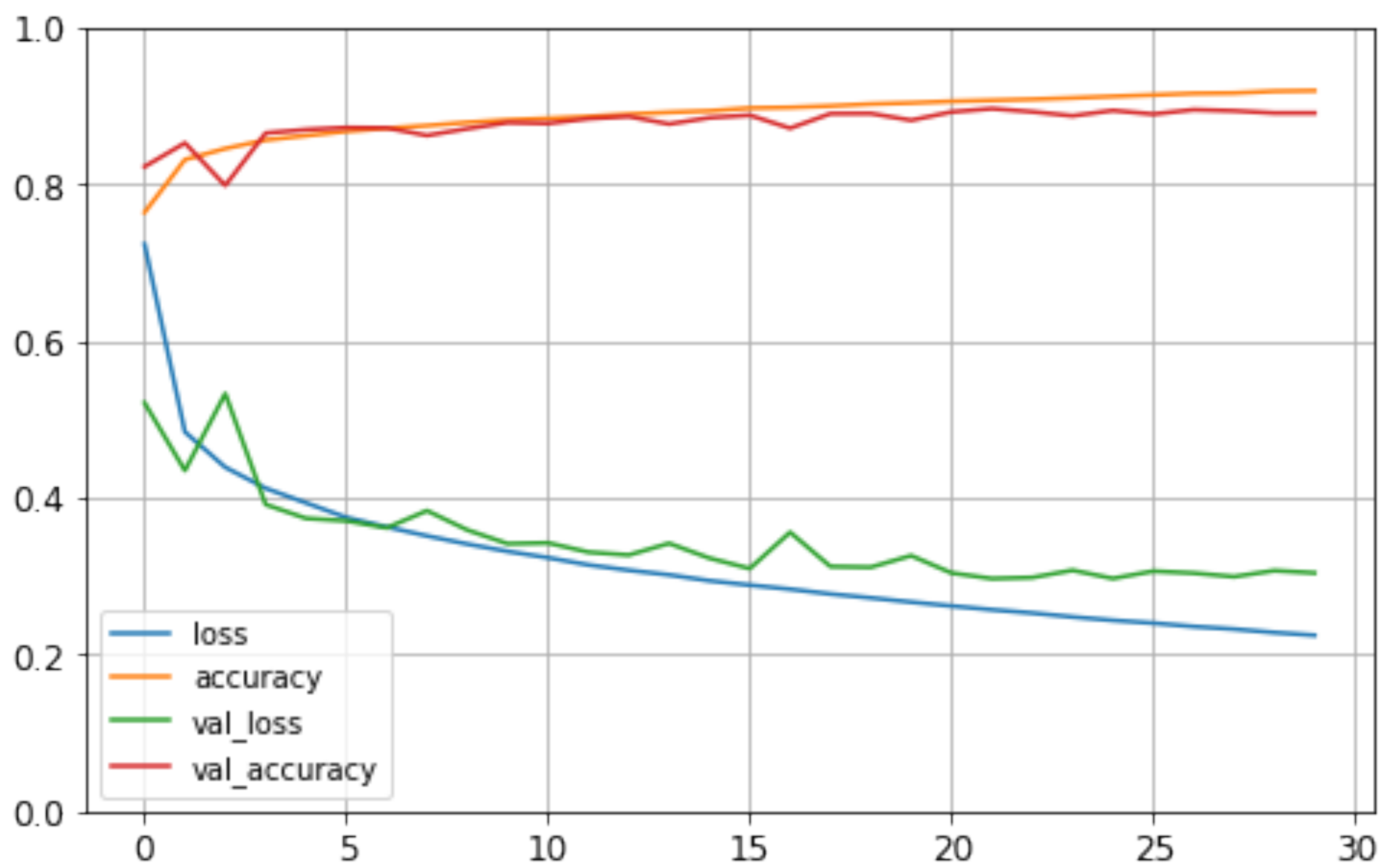
```
history.history.keys()
```

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

Plotting learning curves

```
import pandas as pd

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



Compute the generalization error

- Once you are satisfied with your model's validation accuracy, you should evaluate it on the test set

```
model.evaluate(X_test, y_test,batch_size=1)
```

```
10000/10000 [==============================] - 13s 1ms/step - loss: 0.3376 - accuracy: 0.8819
[0.33763089776039124, 0.8819000124931335]
```

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

```
313/313 [==============================] - 0s 1ms/step - loss: 0.3376 - accuracy: 0.8819
[0.33763277530670166, 0.8819000124931335]
```

```
model.evaluate(X_test, y_test,batch_size=32)
```

```
313/313 [==============================] - 0s 1ms/step - loss: 0.3376 - accuracy: 0.8819
[0.33763277530670166, 0.8819000124931335]
```

Using the model to make predictions

- We can use the model's predict() method to make predictions on new instances

  - Probabilities

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

```
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.03, 0.  , 0.96],
       [0.  , 0.  , 0.98, 0.  , 0.02, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

  - Classes

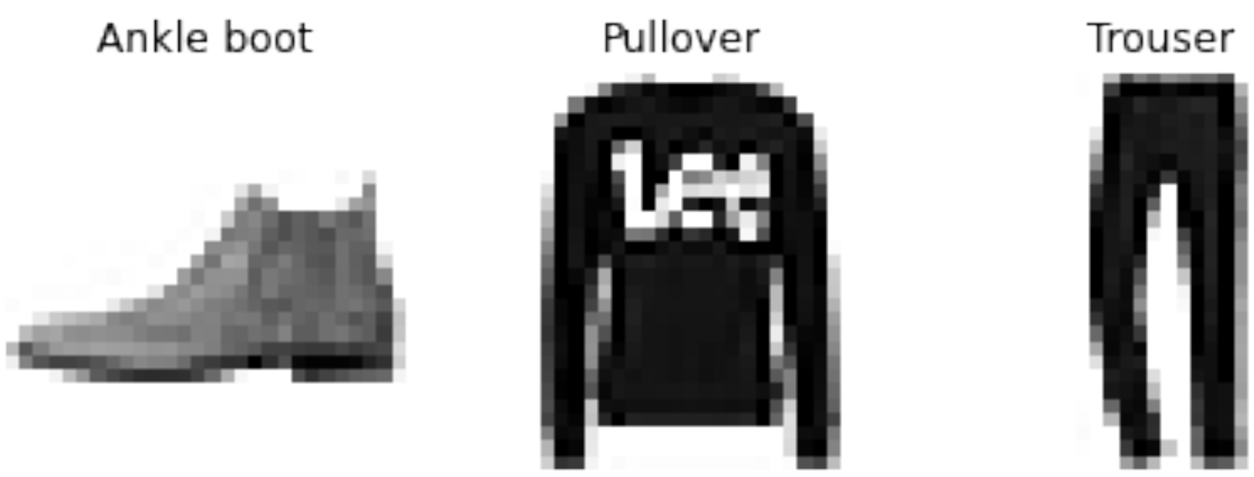
```
y_pred = model.predict_classes(X_new)
y_pred
```

```
WARNING:tensorflow:From <ipython-input-58-
Instructions for updating:
Please use instead:* `np.argmax(model.pred
array([9, 2, 1])
```

Visualization

- Let's look at these three predictions

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



Regression MLP using the sequential API

- We use the California housing data set

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

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

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

```
print(X_train.shape,X_valid.shape,X_test.shape)
```

```
(11610, 8) (3870, 8) (5160, 8)
```

Training and evaluation

- The output layer has a single neuron, one hidden layer, the loss function is MSE

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer=keras.optimizers.SGD(lr=1e-3))
history = model.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

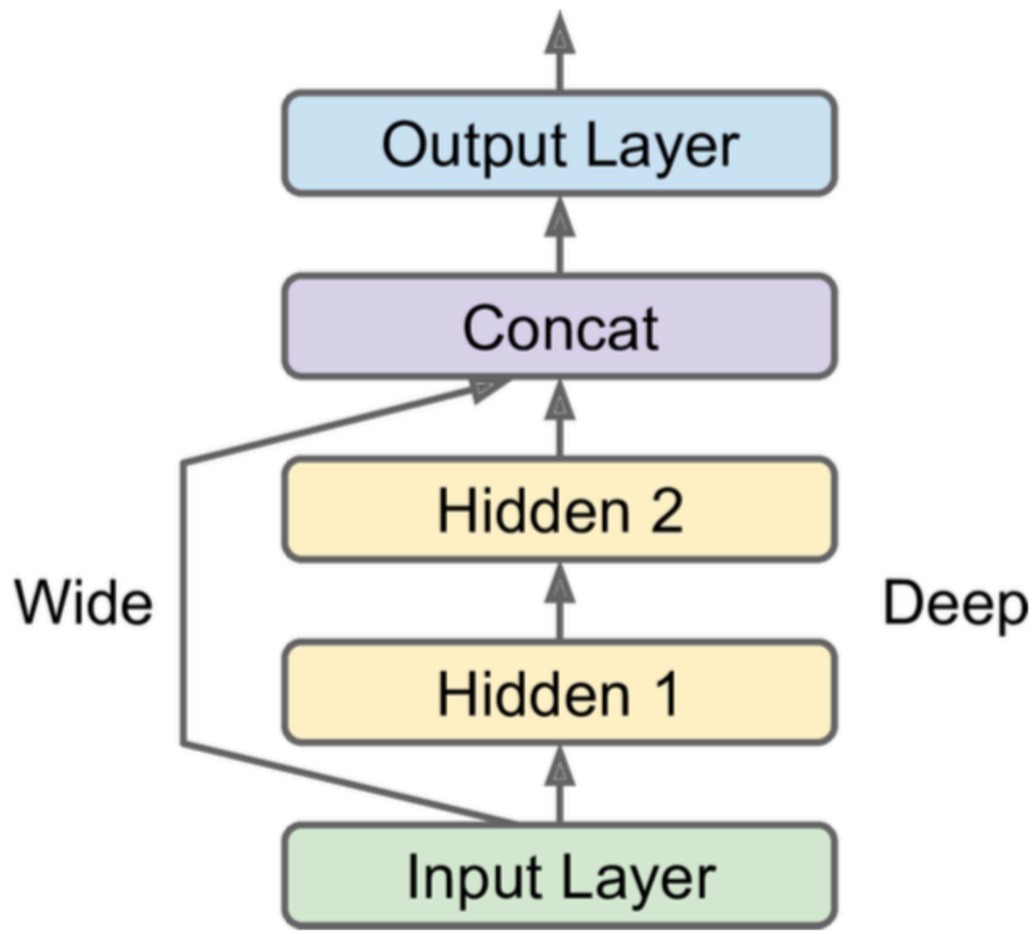
```
Epoch 1/20
363/363 [==============================] - 1s 1ms/step - loss: 1.6419 - val_loss: 0.
Epoch 2/20
363/363 [==============================] - 0s 1ms/step - loss: 0.7047 - val_loss: 0.
Epoch 3/20
363/363 [==============================] - 0s 1ms/step - loss: 0.6345 - val_loss: 0.
```

```
y_pred
```

```
array([[0.38856652],
       [1.6792021 ],
       [3.1022797 ]], dtype=float32)
```

Building complex models using Keras

- One example of non-sequential neural network is a "Wide & Deep" network
  - Connects the inputs directly to the output layer to learn both deep patterns (deep path) and simple rules (short path)



Building a complex model

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input_], outputs=[output])
```

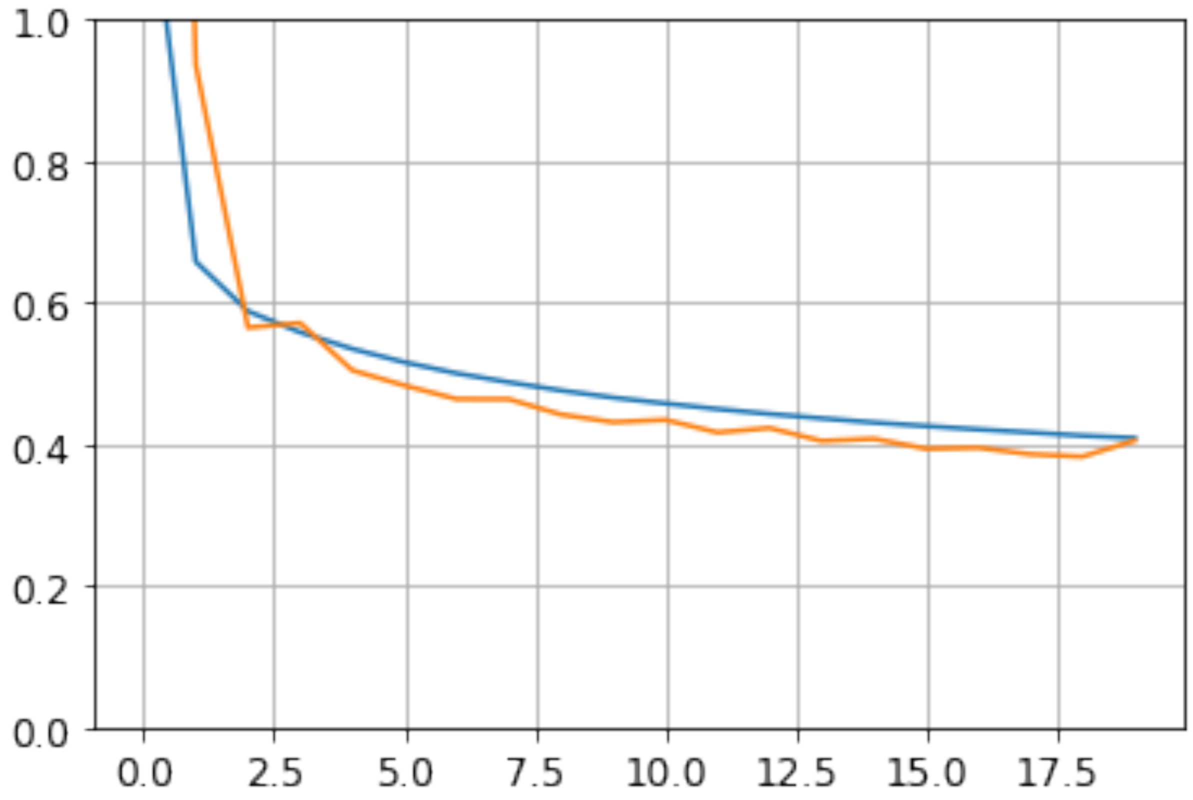
```
model.summary()
```

```
Model: "functional_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 8)]          0
__________________________________________________________________________________________________
dense_5 (Dense)                 (None, 30)           270         input_1[0][0]
__________________________________________________________________________________________________
dense_6 (Dense)                 (None, 30)           930         dense_5[0][0]
__________________________________________________________________________________________________
concatenate (Concatenate)       (None, 38)           0           input_1[0][0]
                                                                 dense_6[0][0]
__________________________________________________________________________________________________
dense_7 (Dense)                 (None, 1)            39          concatenate[0][0]
==================================================================================================
Total params: 1,239
Trainable params: 1,239
Non-trainable params: 0
```

Training and evaluation

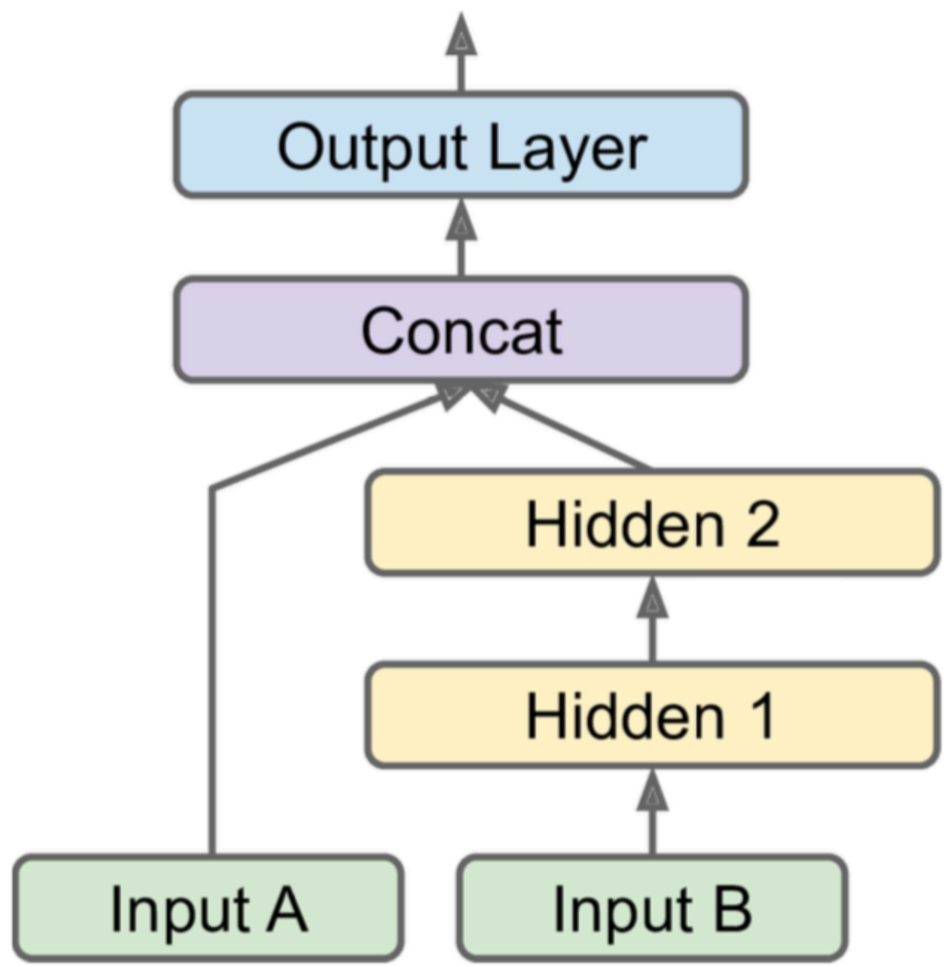
- Similar to what we had before

```
model.compile(loss="mean_squared_error", optimizer=keras.optimizers.SGD(lr=1e-3))
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
plt.plot(pd.DataFrame(history.history))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



More complex model

- Send a subset of the features through the wide path and a different subset through the deep path



- One solution is to use “multiple inputs”

Implementation

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="output")(concat)
model = keras.models.Model(inputs=[input_A, input_B], outputs=[output])
```

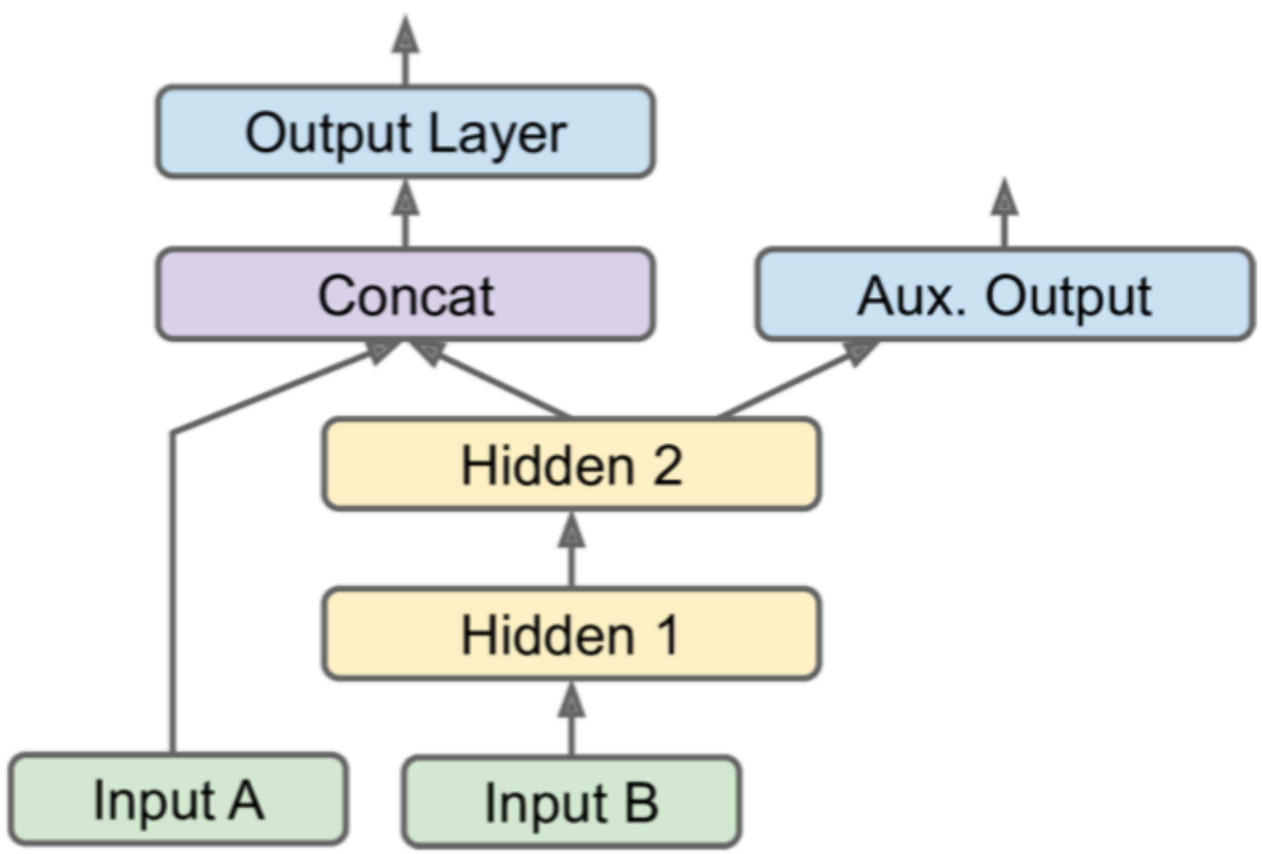
```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))

X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                    validation_data=((X_valid_A, X_valid_B), y_valid))
```

Case of multiple outputs

- There are many cases in which you want to have multiple outputs
  - Example 1: want to locate and classify main object in a picture (both regression and classification tasks)
  - Example 2: multi-task classification (e.g., classify facial expression and identify whether wearing glasses)
  - Example 3: some auxiliary outputs to ensure that the underlying network learns something useful



Implementation

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.models.Model(inputs=[input_A, input_B],
                           outputs=[output, aux_output])
```

- Each output will need its own loss function and a weight parameter

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1],
              optimizer=keras.optimizers.SGD(lr=1e-3))
```

- Training

```
history = model.fit([X_train_A, X_train_B], [y_train, y_train], epochs=20,
                    validation_data=([X_valid_A, X_valid_B],
                                     [y_valid, y_valid]))
```

Saving and restoring a model

- Now, let's see how we can save a trained Keras model

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=[8]),
    keras.layers.Dense(30, activation="relu"),
    keras.layers.Dense(1)
])
```

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
```

- Keras uses HDF5 format to save both the model's architecture and the values of all the model parameters for every layer (i.e., connection weights and biases)

```
model.save("my_keras_model.h5")
```

- We can also load the saved model

```
model_new = keras.models.load_model("my_keras_model.h5")
```

Using Callbacks during training

- When the training process takes a few hours, you should not only save your model at the end
    - Instead, save checkpoints at regular intervals to avoid losing everything

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=[8]),
    keras.layers.Dense(30, activation="relu"),
    keras.layers.Dense(1)
])
```

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
# Model weights are saved at the end of every epoch, if it's the best seen
# so far.
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
```

Fine-tuning neural network hyperparameters

- The flexibility of neural networks is also one of their main drawbacks
    - There are many hyperparameters
        - Number of layers (shallow vs deep models)
        - Number of neurons per layer
        - Activation functions
        - Weight initialization
        - Learning rate
        - Optimizer
        - Batch size

Exploring hyperparameter space

- We can wrap our Keras models in objects that mimic regular Scikit-Learn regressors and then utilize GridSearchCV and RandomizedSearchCV

- Create a simple Sequential model for univariate regression

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model
```

- Create a KerasRegressor based on the above function

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

Exploring hyperparameter space

- Training one model

```
keras_reg.fit(X_train, y_train, epochs=100,
              validation_data=(X_valid, y_valid))
```

- Training different models

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distribs = {
    "n_hidden": [0, 1, 2, 3]
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distribs, n_iter=10, cv=3, verbose=2)
rnd_search_cv.fit(X_train, y_train, epochs=2,
                  validation_data=(X_valid, y_valid))
```

- Finding the best estimator and its hyperparameters

```
rnd_search_cv.best_params_
```

```
{'n_hidden': 3}
```

```
rnd_search_cv.score(X_test, y_test)
```

```
162/162 [==============================] - 0s 781us/step - loss: 0.5202
-0.5202128887176514
```

Reading Assignment: Chapter 10 of Textbook