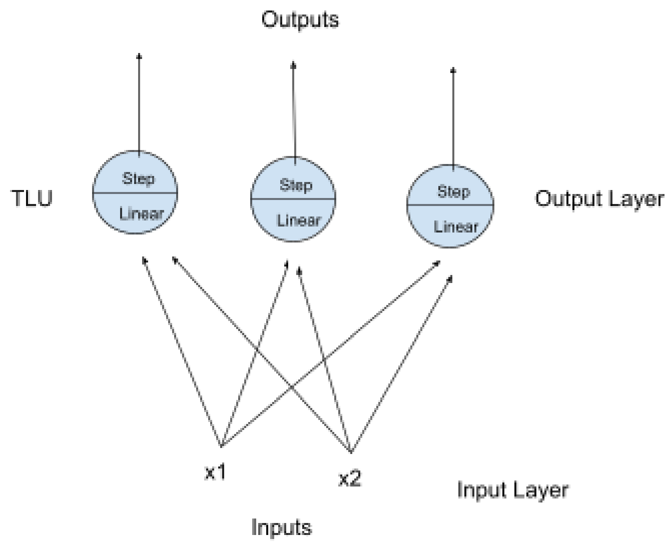
Introduction to Artificial Neural Networks with Keras

Artificial Neural Networks

- ANNs are machine learning models inspired by the networks of our brains' biological neurons.
- ANNs can handle multiple tasks, such as classification, speech recognition, providing recommendations, etc.
    - ANNs often outperform other ML techniques on very large and complex datasets.

The Perceptron

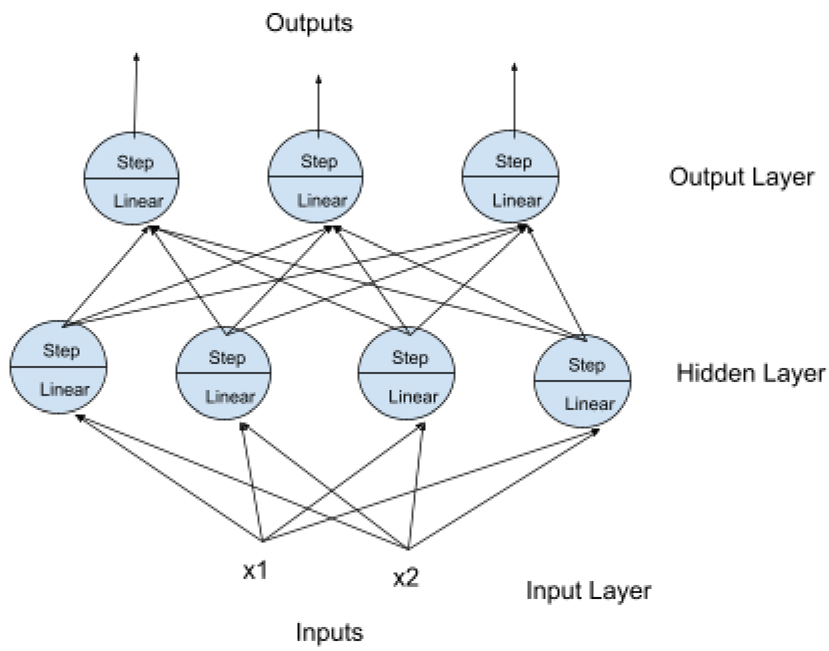
- The perceptron is one of many ANN architectures.
- ANN uses a <u>threshold logic unit</u> (TLU) where each input is associated with a weight. The TLU then computes a linear function of its inputs and applies a step function.
    - Linear function: $z = w_1x_1 + w_2x_2 + ... + w_nx_n + b = w_Tx + b$
    - Step function: $h_w(x) = step(z)$
- We can use a single TLU for different types of classification.
    - For binary classification, the TLU can compute a linear function of its inputs and output the positive or negative classes.
        - $heaviside(z) = 0 \; if \; z < 0, \; or \; 1 \; if \; z \geq 0$
    - The TLU can output three different classes for multilabel or multiclass classification.
        - $sgn(z) = -1 \; if \; z < 0,$
$0 \; if \; z = 0,$
$1 \; if \; z > 0$
        - $h_{W,b}(X) = \phi(XW + b)$ where
            - X is the matrix of input features.
            - The weight matrix W contains all the connection weights.
            - The bias vector b contains all the bias terms.
            - The function ϕ is called the activation function: when the artificial neurons are TLUs, this is the step function.
- A <u>perceptron</u> consists of one or more TLUs organized in a single layer, where every TLU is connected to every input.
    - Such a layer is called a fully connected layer or a <u>dense layer</u>.
    - The inputs are in the input layer, and the outputs are in the output layer.

    - 

- The perceptron is trained based on the Hebbian learning technique.
    - The perceptron is fed one instance at a time, making predictions for each.
    - For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.
    - Perceptron learning rule: $w_{i,j}^{(next\ step)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$ where
        - $w_{i,j}$ is the connection weight between the ith input and the jth neuron.
        - x_i is the ith input value of the current training instance.
        - $\hat{y}_j$ is the target output of the jth output neuron for the current training instance.
        - η is the learning rate.
- The decision boundary of each output neuron is linear, so perceptrons cannot learn complex patterns.
    - However, the algorithm can converge to a solution if the training instances are linearly separable.
- Scikit-Learn provides a **Perceptron()** class that can be used similarly to a classifier (e.g., it has fit and predict methods).
- We can build a <u>multilayer perceptron </u>(MLP) by stacking multiple perceptrons.

MLP and Backpropagation

- An MLP consists of one input layer, one or more layers of TLUs called <u>hidden layers</u>, and one final layer of TLUs called the output layer.



- When an ANN contains a deep stack of hidden layers, it is called a deep neural network (DNN).
- MLPs are trained using a combination of reverse-mode automatic differentiation and gradient descent called backpropagation.
  - The algorithm does two passes through the neural network: one forward, one backward.
  - More specifically, backpropagation follows a process like this:
    - The algorithm makes predictions for a mini-batch (forward pass).
    - It measures the error for the mini-batch.
    - Then, it goes through each layer in reverse to measure the error contribution from each parameter (reverse pass).
    - Finally, it tweaks the connection weights and biases to reduce the error (gradient descent step).
  - The hidden layers' connection weights should be randomly initialized so the algorithm can distinguish between each neuron.
  - Backpropagation uses several activation functions to replace the step function.
    - The sigmoid function forces the step function to have curves, allowing the gradient descent to move. Values can range from 0 to 1.
    - The hyperbolic tangent function (tanh) is S-shaped, continuous, and differentiable. Values can range from -1 to 1.
    - The rectified linear unit function (ReLU) is continuous but not differentiable at z=0. It is the default function used in place of the step function. It does not have a maximum value output, which helps the gradient descent.

Regression MLPs

- MLPs can be used for regression tasks: They can predict a value using a single output neuron.
  - For multivariate regression, we would need one output neuron per output dimension.
- We can use Scikit-Learn's **MLPRegressor()** class to perform regression.
  - Since neural networks use gradient descent, we must scale the input features (e.g., using StandardScaler() with the regressor in a pipeline).
  - We can use the *activation* parameter to set the activation function. By default, the regressor does not use an activation function.
  - The MLPRegressor() class uses the mean squared error as the performance metric.
- A typical architecture for a regression MLP looks like this:
  -

Hyperparameter	Typical value
Hidden layers	Typically 1 to 5
Neurons per hidden layer	Typically 10 to 100
Output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None, ReLU/soft-plus (if positive outputs), or sigmoid/tanh (if bounded outputs)
Loss function	MSE, or Hubber if outliers

Classification MLPs

- We need a single output neuron using the sigmoid function for binary classification: the output will be between 0 and 1.
- We need more than one output neuron for multilabel classification, all using the sigmoid function.
- If each instance can only belong to a single class out of several classes (multiclass classification), we need one output neuron per class, each using a softmax function.
- Scikit-Learn's **MLPClassifier()** class works similarly to the MLPRegressor() class, except the classifier class uses cross-entropy.
- A typical classification MLP architecture looks like this:
  -

Hyperparameter	Binary	Multilabel	Multiclass
Hidden layers	1 to 5	1 to 5	1 to 5
Output neurons	1	1 per binary label	1 per class

Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

Implementing MLPs with TensorFlow's Keras

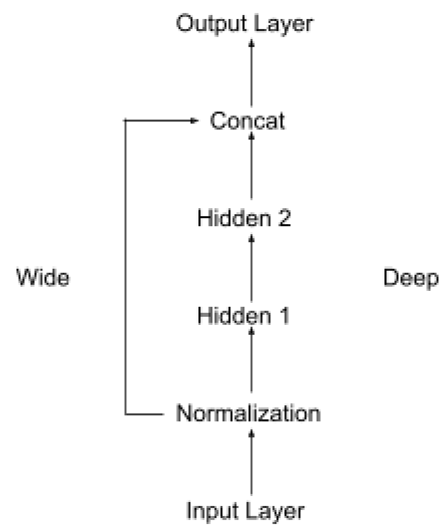
Image Classification with the Sequential API

- Like Scikit-Learn, TensorFlow has a database of datasets that can be accessed with **tensorflow.keras.datasets**.
    - Moving forward in this chapter, any class or function mentioned will come from TensorFlow's Keras.
- We can use the **tensorflow.random.set_seed()** method to create reproducible examples.
- Then, we can create a model using the **Sequential()** class. We can use its **add()** method to add layers to the model.
    - The types of layers are available in the **keras.layers** package.
        - Input() adds inputs to the model. We specify the shape of the inputs, which typically is the same as the shape of the training set.
        - Flatten() converts the inputs (in this case, images) into 1D arrays.
        - Dense() creates a dense hidden layer. We can specify the activation function with the *activation* parameter.
- Here are some useful methods for the Sequential model.
    - **summary()** displays the model's layers with their names and shapes.
    - **layers** and **get_layers()** returns a list of the model's layers.
    - **get_weights()** and **set_weights()** returns the model's parameters.
- Note that the model may have several thousands of parameters, making it flexible to fit the training data but prone to overfitting.
- We call the model's **compile()** method to set its loss function, optimizer, and performance metric.
    - Recall that classification MLPs use the cross entropy loss function. Multiple types of cross-entropy loss functions exist, such as categorical and binary.
    - We can use stochastic gradient descent as the optimizer.
    - It's useful to use accuracy as a metric for classification.
- Calling the model's **fit()** method trains the model and creates a history of each epoch.
    - We can use Pyplot to visualize the model's performance using the history object that the fit method returns.
    - The accuracy of the training and validation sets should increase during training while the loss should decrease.
- Once we are satisfied with the model's accuracy, we use the **evaluate()** method to evaluate the model on the test set.
- Lastly, we can make new predictions using the model's **predict()** method.

Building a Regression MLP using the Sequential API

- The process for making a regression MLP is essentially the same as for making a classification MLP.
- The main differences are the output layer has a single neuron to predict a value, it uses no activation function, the loss function is the mean squared error, the metric is the RMSE, and it uses the Adam optimizer.
    - Instead of using a Flatten() layer for the first layer, we use a Normalization() layer, which is similar to the StandardScaler().
    - In this case, we must use the model's **adapt()** method before calling the fit() method.

Building Complex Models using the Functional API

- An example of a nonsequential neural network is a <u>Wide & Deep</u> neural network.
    - This neural network directly connects all or part of the inputs to the output layer.
    - This allows the neural network to learn deep patterns (using the deep path) and simple rules (using the wide path).
    - 

    - First, we create all the layers shown above (depending on the task, we may need more hidden layers). Then, we create the model using those layers.
    - In some cases, we may want to pass different subsets of features through each path. We can create two inputs and pass them into the model.
    - Alternatively, we may need a neural network with more than one output.
        - We could train one neural network per task, but generally, a single neural network with one output per task will yield better results.
        - Each output will need its loss function, so when we compile the model, we must pass a list of losses.
- We can use the compile() , evaluate(), and predict() methods as we did before.
    - We can access the Adam optimizer through **keras.optimizers.Adam()** and providing it with a learning rate.

Using the Subclass API to Build Dynamic Models

- Both the sequential API and the functional API are declarative: we declare which layers we want to use and how they should be connected, and we feed the model some data for training or inference.
- We may want to create a custom model.
    - We can create a class for our model and pass in **tensorflow.keras.Model** as a class parameter for our class to inherit Keras's Model's parameters.
    - Here, we have more flexibility and use loops, conditional branches, etc.

Saving and Restoring a Model

- We can use the model's **save()** method to store the model.
- To load the model, we can use its **load_model()** method.
- We can also use the **save_weights()** and **load_weights()** methods to store the parameter values, which is faster and more memory efficient.
    - We can save checkpoints during training in case an error occurs.

Using Callbacks

- The fit() method has a *callbacks* argument to save checkpoints during training.
    - The callbacks are available through the **callbacks.ModelCheckpoint()** method.
- We can also use an early-stopping checkpoint, interrupting training when the algorithm measures no progress on the validation set for several epochs.
    - We can access early-stopping checkpoints with the **callbacks.EarlyStopping()** method.

Fine-Tuning Neural Network Hyperparameters

- Neural networks have abundant hyperparameters, making tweaking them difficult. We need to find an optimal combination of hyperparameters to tweak.
- One option is to use the GridSearchCV() or RandomizerSearchCV() method to fine-tune the hyperparameters.
    - In this case, we must use the **KerasRegressor()** and **KerasClassifier()** wrapper classes from the SciKeras library.
- An easier option is to use the **keras_tuner** library.
    - We can write a function that builds, compiles, and returns a Keras model.
    - This function must take a **keras_tuner.HyperParameters** object as an argument. These hyperparameters can be integers, floats, strings, etc.
    - We must also define the number of layers and neurons, the learning rate, and the optimizer.
    - Then, we call the library's **RandomSearch()** function, providing it with the function's name and calling the **search()** method.
        - We can access the best models and hyperparameters using the **get_best_models()** and **get_best_hyperparameters()** methods.
    - In some cases, we may want to change how the model uses its fit() method. We can create a custom model.

        - We create a class for this custom model and pass in **keras_tuner.HyperModel** as an argument.
        - Then, we can build a **HyperBand()** tuner using this custom model.
    - Another type of tuner is the **BayesianOptimization()** tuner.

The Number of Hidden Layers

- Deep networks have a much higher parameter efficiency than shallow ones: they can model complex functions using exponentially fewer neurons than shallow nets.
    - Lower hidden layers model low-level structures, intermediate layers combine the low-level structures to model intermediate-level structures, and higher layers combine the intermediate structures to model high-level structures.
    - In image classification, the low-level structures may be lines, the intermediate structures may be squares and circles, and the higher-level structures may be the complete image.
- This way, the network will not have to learn all the low-level structures from scratch; it will only have to learn the higher-level structures (e.g., hairstyles). This is called <u>transfer learning</u>.

The Number of Neurons per Hidden Layer

- Before, hidden layers would be structures with fewer neurons at each layer (like a pyramid).
    - The logic was that many low-level features can transform into far fewer high-level features.
- It seems that using the same number of neurons in all hidden layers performs as well or even better in most cases.
- Just like the number of layers, we can try gradually increasing the number of neurons until the network starts overfitting.
- Alternatively, we can try building a model with slightly more layers and neurons than we need, then use early stopping and other regularization techniques to prevent it from overfitting too much.

Other Hyperparameters

- Learning rate: One way to find a good learning rate is to train the model for a few hundred iterations, starting with a very low learning rate and gradually increasing it up to a very large value.
- Optimizer: we should choose a good optimizer and tune its hyperparameters.
- Batch size: we can try a large batch size and switch to a smaller batch if the performance suffers.
- Activation function: ReLU is a good default function, but we can try others.
- Number of iterations: the number of training iterations does not need to be tweaked; we can simply use early stopping.