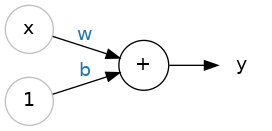
# The Linear Unit

So let's begin with the fundamental component of a neural network: the individual neuron. As a diagram, a **neuron** (or **unit**) with one input looks like:



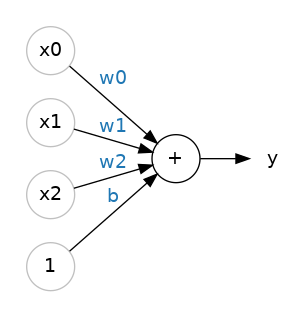
The Linear Unit: y=wx+b

The input is x. Its connection to the neuron has a **weight** which is w. Whenever a value flows through a connection, you multiply the value by the connection's weight. For the input x, what reaches the neuron is w \* x. A neural network "learns" by modifying its weights.

The b is a special kind of weight we call the **bias**. The bias doesn't have any input data associated with it; instead, we put a 1 in the diagram so that the value that reaches the neuron is just b (since 1 \* b = b). The bias enables the neuron to modify the output independently of its inputs.

The y is the value the neuron ultimately outputs. To get the output, the neuron sums up all the values it receives through its connections. This neuron's activation is y = w \* x + b, or as a formula y=wx+b

# Multiple Inputs:



# Linear Units in Keras:

<https://www.tensorflow.org/api_docs/python/tf/keras/layers>

from tensorflow import keras

from tensorflow.keras import layers

*# Create a network with 1 linear unit*

model = keras.Sequential([

layers.Dense(units=1, input\_shape=[3])

])

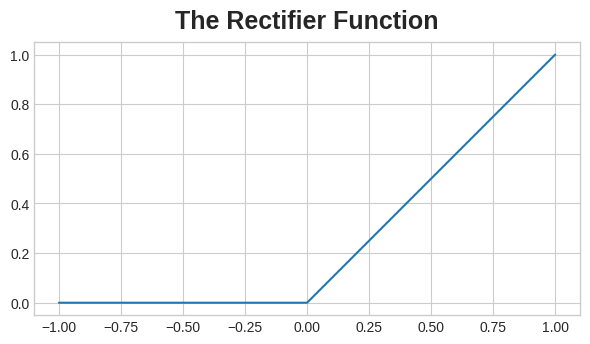
## Get Weights:

W, b = model.weights

## The Activation Function

*Without activation functions, neural networks can only learn linear relationships. In order to fit curves, we'll need to use activation functions.*

An **activation function** is simply some function we apply to each of a layer's outputs (its *activations*). The most common is the *rectifier* function max(0,x)max(0,x).

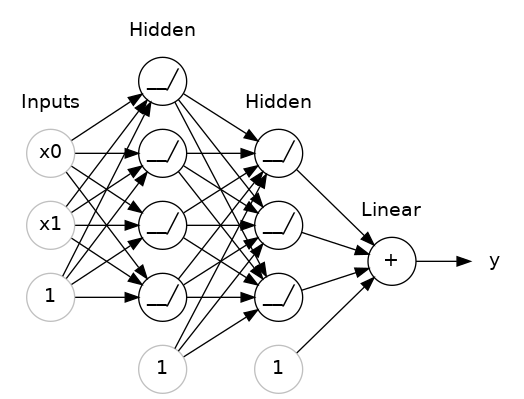


**ReLU (rectified linear unit ):**

=> max(0, w \* x + b)

### Stacking Dense Layers:

Now that we have some nonlinearity, let's see how we can stack layers to get complex data transformations.



from tensorflow import keras

from tensorflow.keras import layers

model = keras.Sequential([

*# the hidden ReLU layers*

layers.Dense(units=4, activation='relu', input\_shape=[2]),

layers.Dense(units=3, activation='relu'),

*# the linear output layer*

layers.Dense(units=1),

])

### Activation Functions:

<https://www.tensorflow.org/api_docs/python/tf/keras/activations>

## The Loss Function

A common loss function for regression problems is the **mean absolute error** or **MAE**. For each prediction y\_pred, MAE measures the disparity from the true target y\_true by an absolute difference abs(y\_true - y\_pred).

## The Optimizer - Stochastic Gradient Descent

1. Sample some training data and run it through the network to make predictions.
2. Measure the loss between the predictions and the true values.
3. Finally, adjust the weights in a direction that makes the loss smaller.

Each iteration's sample of training data is called a **minibatch** (or often just "batch"), while a complete round of the training data is called an **epoch**. The number of epochs you train for is how many times the network will see each training example.

# Learning Rate and Batch Size:

Notice that the line only makes a small shift in the direction of each batch (instead of moving all the way). The size of these shifts is determined by the **learning rate**. A smaller learning rate means the network needs to see more minibatches before its weights converge to their best values

## Adding the Loss and Optimizer

After defining a model, you can add a loss function and optimizer with the model's compile method:

from tensorflow import keras

from tensorflow.keras import layers

model = keras.Sequential([

layers.Dense(512, activation='relu', input\_shape=[11]),

layers.Dense(512, activation='relu'),

layers.Dense(512, activation='relu'),

layers.Dense(1),

])

model.compile(

optimizer="adam",

loss="mae",

)

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_valid, y\_valid),

batch\_size=256,

epochs=10,

)

Often, a better way to view the loss though is to plot it. The fit method in fact keeps a record of the loss produced during training in a History object. We'll convert the data to a Pandas dataframe, which makes the plotting easy.

import pandas as pd

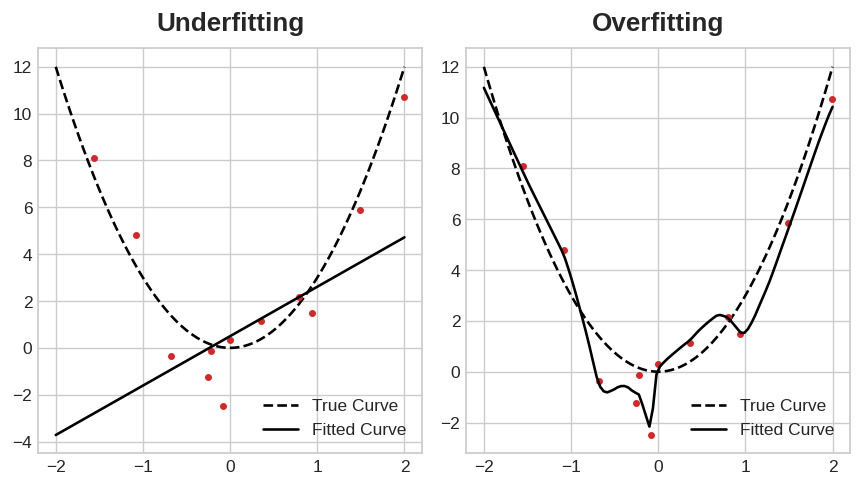
*# convert the training history to a dataframe*

history\_df = pd.DataFrame(history.history)

*# use Pandas native plot method*

history\_df['loss'].plot();

# Interpreting the Learning Curves



## Capacity:

A model's **capacity** refers to the size and complexity of the patterns it is able to learn

You can increase the capacity of a network either by making it *wider* (more units to existing layers) or by making it *deeper* (adding more layers).

model = keras.Sequential([

layers.Dense(16, activation='relu'),

layers.Dense(1),

])

wider = keras.Sequential([

layers.Dense(32, activation='relu'),

layers.Dense(1),

])

deeper = keras.Sequential([

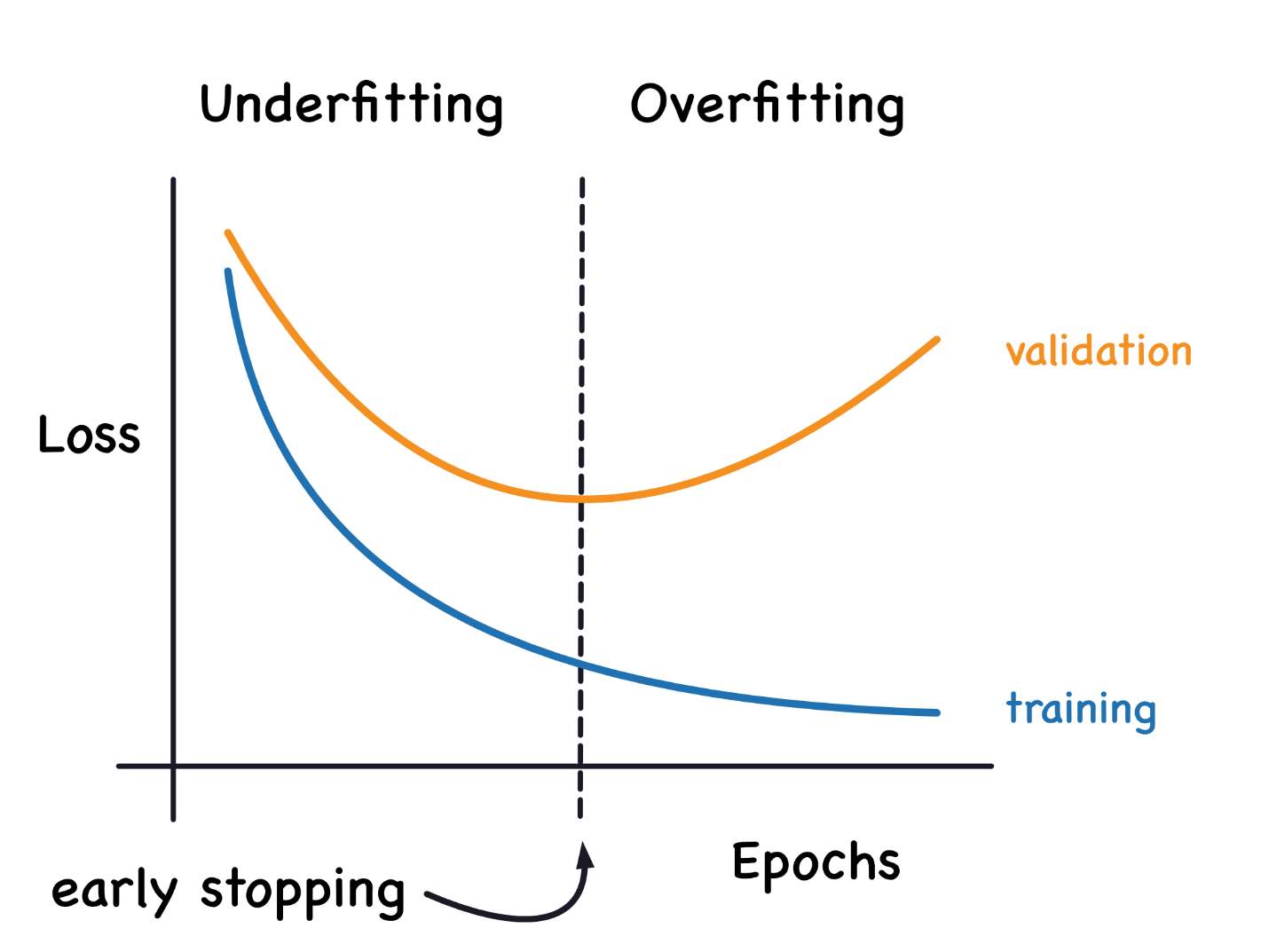
layers.Dense(16, activation='relu'),

layers.Dense(16, activation='relu'),

layers.Dense(1),

])

## Early Stopping:



from tensorflow.keras.callbacks import EarlyStopping

early\_stopping = EarlyStopping(

min\_delta=0.001, *# minimium amount of change to count as an improvement*

patience=20, *# how many epochs to wait before stopping*

restore\_best\_weights=True,

)

**These parameters say**: "If there hasn't been at least an improvement of 0.001 in the validation loss over the previous 20 epochs, then stop the training and keep the best model you found."

from tensorflow import keras

from tensorflow.keras import layers, callbacks

early\_stopping = callbacks.EarlyStopping(

min\_delta=0.001, *# minimium amount of change to count as an improvement*

patience=20, *# how many epochs to wait before stopping*

restore\_best\_weights=True,

)

model = keras.Sequential([

layers.Dense(512, activation='relu', input\_shape=[11]),

layers.Dense(512, activation='relu'),

layers.Dense(512, activation='relu'),

layers.Dense(1),

])

model.compile(

optimizer='adam',

loss='mae',

)

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_valid, y\_valid),

batch\_size=256,

epochs=500,

callbacks=[early\_stopping], *# put your callbacks in a list*

verbose=0, *# turn off training log*

)

history\_df = pd.DataFrame(history.history)

history\_df.loc[:, ['loss', 'val\_loss']].plot();

print("Minimum validation loss: **{}**".format(history\_df['val\_loss'].min()))

# Dropout and Batch Normalization:

## Dropout:

we randomly *drop out* some fraction of a layer's input units every step of training, making it much harder for the network to learn those spurious patterns in the training data

rate defines what percentage of the input units to shut off

keras.Sequential([

# ...

layers.Dropout(rate=0.3), # apply 30% dropout to the next layer

layers.Dense(16),

# ...

])

## Batch Normalization:

A batch normalization layer looks at each batch as it comes in, first normalizing the batch with its own mean and standard deviation, and then also putting the data on a new scale with two trainable rescaling parameters. Batchnorm, in effect, performs a kind of coordinated rescaling of its inputs.

layers.Dense(16, activation='relu'),

layers.BatchNormalization(),

... or between a layer and its activation function

layers.Dense(16),

layers.BatchNormalization(),

layers.Activation('relu'),

When adding dropout, you may need to increase the number of units in your Dense layers.

from tensorflow import keras

from tensorflow.keras import layers

model = keras.Sequential([

layers.Dense(1024, activation='relu', input\_shape=[11]),

layers.Dropout(0.3),

layers.BatchNormalization(),

layers.Dense(1024, activation='relu'),

layers.Dropout(0.3),

layers.BatchNormalization(),

layers.Dense(1024, activation='relu'),

layers.Dropout(0.3),

layers.BatchNormalization(),

layers.Dense(1),

])

# Binary Classification

 You might want to predict whether a customer is likely to make a purchase, whether or not a credit card transaction was fraudulent.

In your raw data, the classes might be represented by strings like "Yes" and "No", or "Dog" and "Cat". Before using this data, we'll assign a **class label**: one class will be 0 and the other will be 1. Assigning numeric labels puts the data in a form a neural network can use

## Accuracy and Cross-Entropy

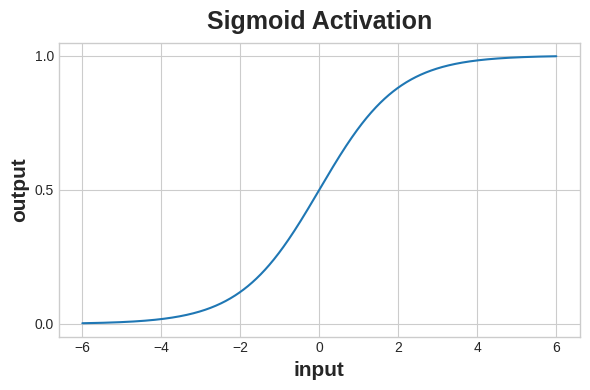
**accuracy** = number\_correct / total

ratio of correct predictions to total predictions is called accuracy

**Cross-entropy** is a sort of measure for the distance from one probability distribution to another.

### Sigmoid Function:

To covert the real-valued outputs produced by a dense layer into probabilities i.e, from range 0 to 1, we attach a new kind of activation function, the **sigmoid activation.**



**Example**:

In the final layer include a 'sigmoid' activation so that the model will produce class probabilities

**Code:**

from tensorflow import keras

from tensorflow.keras import layers

model = keras.Sequential([

layers.Dense(4, activation='relu', input\_shape=[33]),

layers.Dense(4, activation='relu'),

layers.Dense(1, activation='sigmoid'),

])

Add the **cross-entropy loss and accuracy metric** to the model with its **compile method**. For two-class problems, be sure to use **'binary'** versions.

model.compile(

optimizer='adam',

loss='binary\_crossentropy',

metrics=['binary\_accuracy'],

)

The model in this particular problem can take quite a few epochs to complete training, so we'll include an early stopping callback for convenience.

early\_stopping = keras.callbacks.EarlyStopping(

patience=10,

min\_delta=0.001,

restore\_best\_weights=True,

)

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_valid, y\_valid),

batch\_size=512,

epochs=1000,

callbacks=[early\_stopping],

verbose=0, *# hide the output because we have so many epochs*

)

**Note:**

Remember that early stopping will restore the weights to those that got these values.

**Plotting:**

history\_df = pd.DataFrame(history.history)

*# Start the plot at epoch 5*

history\_df.loc[5:, ['loss', 'val\_loss']].plot()

history\_df.loc[5:, ['binary\_accuracy', 'val\_binary\_accuracy']].plot()

print(("Best Validation Loss: **{:0.4f}**" +\

"**\n**Best Validation Accuracy: **{:0.4f}**")\

.format(history\_df['val\_loss'].min(),

history\_df['val\_binary\_accuracy'].max()))

## Example Preprocessing:

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.impute import SimpleImputer

from sklearn.pipeline import make\_pipeline

from sklearn.compose import make\_column\_transformer

hotel = pd.read\_csv('../input/dl-course-data/hotel.csv')

X = hotel.copy()

y = X.pop('is\_canceled')

X['arrival\_date\_month'] = \

X['arrival\_date\_month'].map(

{'January':1, 'February': 2, 'March':3,

'April':4, 'May':5, 'June':6, 'July':7,

'August':8, 'September':9, 'October':10,

'November':11, 'December':12}

)

features\_num = [

"lead\_time", "arrival\_date\_week\_number",

"arrival\_date\_day\_of\_month", "stays\_in\_weekend\_nights",

"stays\_in\_week\_nights", "adults", "children", "babies",

"is\_repeated\_guest", "previous\_cancellations",

"previous\_bookings\_not\_canceled", "required\_car\_parking\_spaces",

"total\_of\_special\_requests", "adr",

]

features\_cat = [

"hotel", "arrival\_date\_month", "meal",

"market\_segment", "distribution\_channel",

"reserved\_room\_type", "deposit\_type", "customer\_type",

]

transformer\_num = make\_pipeline(

SimpleImputer(strategy="constant"), # there are a few missing values

StandardScaler(),

)

transformer\_cat = make\_pipeline(

SimpleImputer(strategy="constant", fill\_value="NA"),

OneHotEncoder(handle\_unknown='ignore'),

)

preprocessor = make\_column\_transformer(

(transformer\_num, features\_num),

(transformer\_cat, features\_cat),

)

# stratify - make sure classes are evenlly represented across splits

X\_train, X\_valid, y\_train, y\_valid = \

train\_test\_split(X, y, stratify=y, train\_size=0.75)

X\_train = preprocessor.fit\_transform(X\_train)

X\_valid = preprocessor.transform(X\_valid)

input\_shape = [X\_train.shape[1]]

**Competition**:

* Classify images with TPUs in [**Petals to the Metal**](https://www.kaggle.com/c/tpu-getting-started)
* Create art with GANs in [**I'm Something of a Painter Myself**](https://www.kaggle.com/c/gan-getting-started)
* Classify Tweets in [**Real or Not? NLP with Disaster Tweets**](https://www.kaggle.com/c/nlp-getting-started)
* Detect contradiction and entailment in [**Contradictory, My Dear Watson**](https://www.kaggle.com/c/contradictory-my-dear-watson)