Contents

What is deep learning

- Deep learning is the sub-filed of machine learning which uses algorithm
- It is inspired by the structure and function of brain's neural networks.
- The algorithm or models that do in this are based loosely on the structure and functions of brain's neural network
- The neural networks that we use in deep learning have some characteristics with biological neural networks that's why called it artificial neural network (ANN).

What does Deep mean in deep learning

For this we need to know how ANN is structured.

For now, here is what we need to know

- ANN are built using neuron
- Neuron are organized into layers
- Layers within an ANN are called hidden layers
- If an ANN has more than one hidden layer, ANN said to be deep ANN

Artificial Neural Net wok

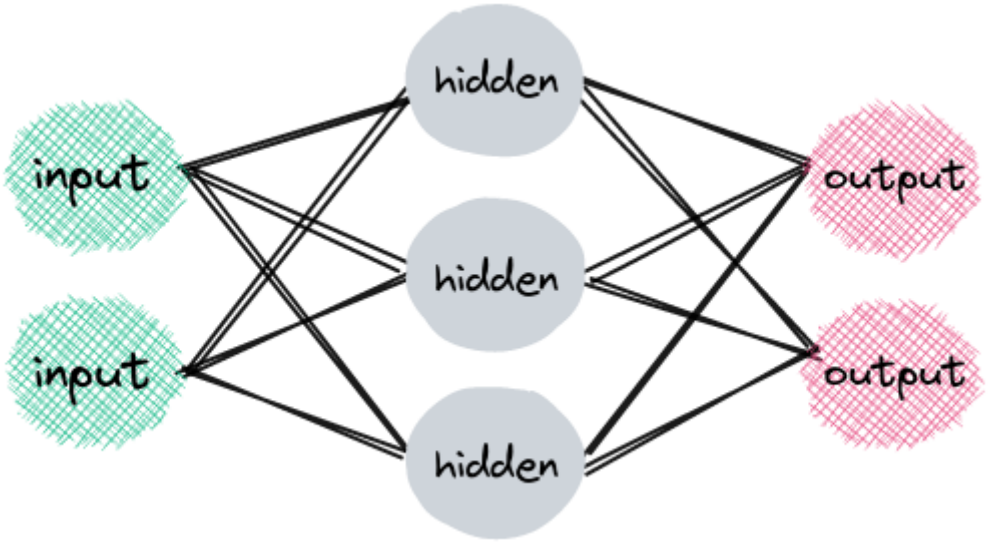
- Artificial neural network is computing system that is inspired by brain's neural networks.
- These networks are based on the collection of connected unit called neurons
- Each connection between neurons can transmit a signal from one neuron to another neuron
- The receiving neurons process the signals and signals downstream the neuron connected to it.
- Neurons are organized in layers

Input layers

hidden layers

output layers

- Different layers perform different kinds of transformations
- Data flows through the network starting at input moving through hidden layers until output layers is reached this is known as forward pass.



- Now here we will use **sequential model** of **Keras** to illustrate an example

Sequence models

sequence models are machine learning model that input or output sequence of data.

Sequence data includes text, audio, video time-series data etc. **Recurrent neural network** is a popular algorithm used in sequence models

- **Basic model example in Keras**

Dense is one type of layers

- first parameter tells how many neuron it should have
- second parameter tells how many neurons our input layer has
- and third parameter is activation function

```
from keras.models import Sequential
from keras.layers import Dense, Activation

layers = [
    Dense(units=3, input_shape=(2,), activation='relu'),
    Dense(units=2, activation='softmax')
]
model = Sequential(layers)
```

+ Code + Markdown

dense layers is also known as fully connected layers

Layers in neural network

- Dense or fully connected layers
- Convolutional layers
- Pooling layers
- Recurrent layers
- Normalization layers

Why have different types of layers

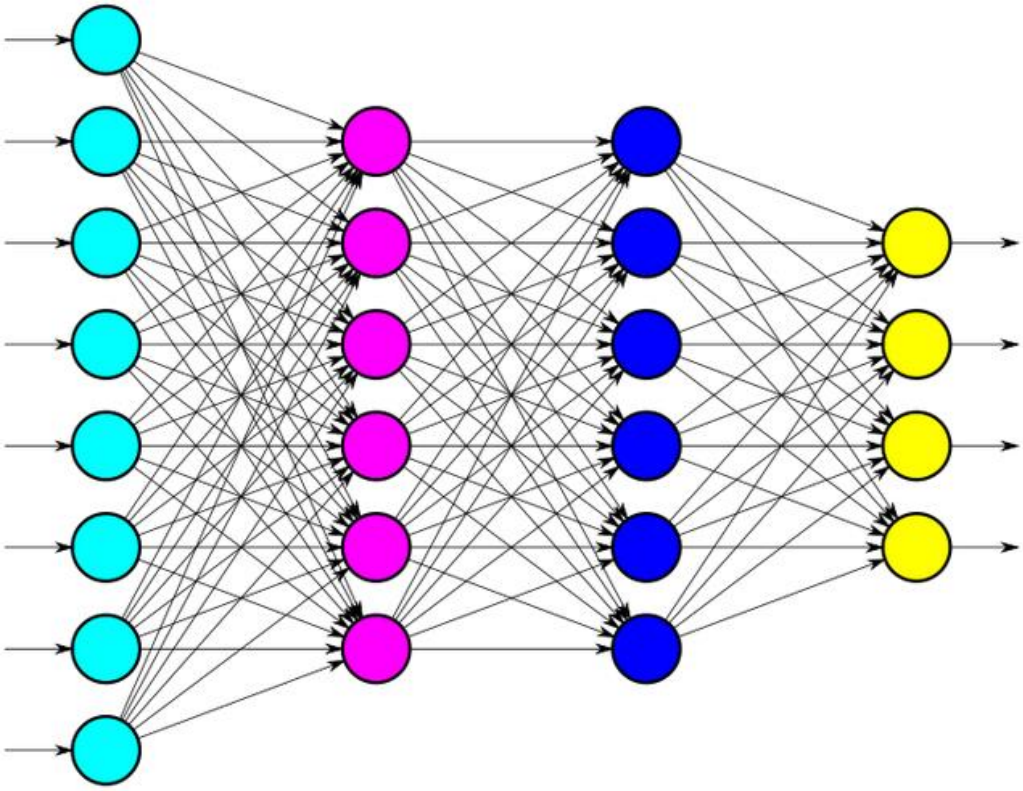
- different types of layers perform different transformations on their input, and some are better suited for some task than other.
- **For example**

convolutional layer is used in model that are doing work with image data

Recurrent layers are used in models that are doing work with time series data

fully connected layers fully connect each input to each output within its layer

Example of artificial neural network



- The above layer have eight nodes. Each 8 nodes represent an individual features given sample in our datasets. This tells us that single sample from our datasets consists eight dimensions
- We can see that each of the eight input nodes are connected to every node in the next layer (with hidden layer).

Layer weight

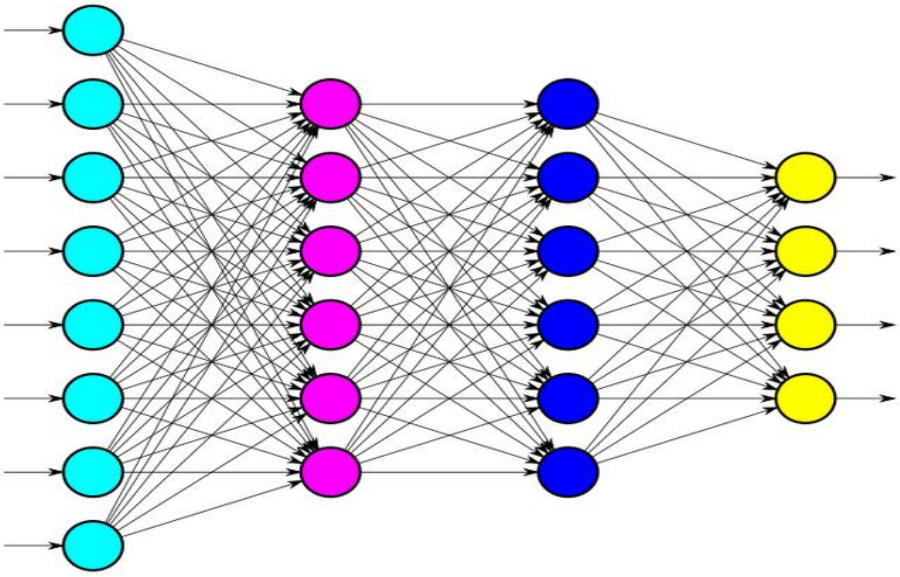
- Each connection between two nodes has an associated weight (which is just a number).
- Each weight represents the strength of connection between two nodes.
- When we first input receive input in input layer, this input is passed to the next node via connection, and *input will be multiplied by the weight assigned to that connection*.
- A weighted sum is then computed with each of the incoming connections. Then that sum is

passed to an activation function, which perform some type of transformation.
For example, activation function may transform the sum to be between zero and one.

node output = activation(weighted sum of inputs)

- Once we get the transformation result from the activation that result pass to the next neuron of next layer, and this process will occur over and over again until we reach output layer.

For this network



- for this we have

- Input shape is **8**, that is why shape is specified as input_shape = (8). first hidden layer (pink one) and second hidden layer (blue one) has **6** nodes, but output layers (yellow one) have **4** nodes.

```
layers = [
    Dense(units=6, input_shape=(8,), activation='relu'),
    Dense(units=6, activation='relu'),
    Dense(units=4, activation='softmax')
]
```

- We are using activation function called '**relu**' for both hidden layer, and '**softmax**' activation for output layer
- Notice input_shape is only require in first dense layer

Final product look like this

```
from keras.models import Sequential
from keras.layers import Dense, Activation

layers = [
    Dense(units=6, input_shape=(8,), activation='relu'),
    Dense(units=6, activation='relu'),
    Dense(units=4, activation='softmax')
]

model = Sequential(layers)
```

+ Code + Markdown

Activation function in neural network

- It maps a node's input to its corresponding output.
- We know that **node output = activation (weighted sum of inputs)**. Here activation function does some type of operation to transform the sum to a number. This transformation is often a non-linear transformation.

What do activation functions do?

what's up with this activation function transformation? What's the intuition? To explain this, let's first look at some example activation functions.

Sigmoid activation function

sigmoid takes input and does the following

For most negative input, sigmoid transform the input to a number close to **0**.

For most positive input, sigmoid will transform the input into number close to **1**.

For relatively close to **0**, sigmoid transform input between **0** and **1**.

For sigmoid **0** is the lower limit and **1** is the upper limit.

Activation function intuition

- Activation function is biologically inspired by the activity of our brains where different neurons are fire (or activated) by different stimuli.
- For example if we smell something pleasant, like fleshy baked cookies, certain neurons in our brain will fire and become activated. If we smell something unpleasant like spoiled milk, this will cause other neurons of our brain fire.
- Deep within our brain certain neurons are either firing or they are not. This can be represented by psuedocode.

// pseudocode

```
if (smell.isPleasant()) {
   neuron.fire();
}
```

- We have seen that neuron can be between 0 and 1. close to 1 means more activated neuron, while closer to 0 means less activated neuron.

Relu Activation function

- Our activation function not always going to do transformation on an input between **0** and **1**.
- Relu activation function doe's do this.
- Relu transform the maximum of **0** of either the input itself.
- So if the input is less than or equal to **0**, then Relu will ouput **0**. If input is greater than **0**, then Relu will give output given input,

// pseudocode

```
function relu(x) {
   if (x <= 0) {
      return 0;
   } else {
      return x;
   }
}
```

- The idea here is more positive neuron the more activate neuron it is.

Why do we use activation function?

- To understand why we use activation functions, we need to first understand linear functions.

Suppose that

F is a function on a set **X**.

Suppose that **a** and **b** are in **X**.

Suppose that **X** is a real number.

The function F is said to be a linear function if and only if:

f(a+b) = f(a) + f(b) and ***f(xa) = f(xb)***

- An important feature of linear functions is that the composition of two linear functions is also a linear function. This means that, even in very deep neural networks, if we only had linear transformations of our data values during a forward pass, the learned mapping in our network from input to output would also be linear.
- Typically, the types of mappings that we are aiming to learn with our deep neural networks are more complex than simple linear mappings.
- This is where activation functions come in. Most activation functions are non-linear, and they are chosen in this way on purpose. Having non-linear activation functions allows our neural networks to compute arbitrarily complex functions

Activation function in code with Keras

- Example:-

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(units=5, input_shape=(3,), activation='relu')
])

# Another way is by adding activation
model = Sequential()
model.add(Dense(units=5, input_shape=(3,)))
model.add(Activation('relu'))
```

+ Code + Markdown

- Remember that

node output = activation(weighted sum of inputs)

- So

node output = relu(weighted sum of inputs)

Training a model

- When we train the model we are basically trying to solve an optimization problem. We are trying to optimize the weights within the model.
- Our task is to find the weights that most accurately map our input data to the correct putout class

Mapping is what the network must learn

- During the training weights (weight of connection between node) are iteratively updated and moved forwards their optimal value

// pseudocode

```
def train(model):
    model.weights.update()
```

optimization algorithms

- The weights are optimize using optimization algorithm
- The optimization process depends upon the chosen optimization algorithm (most widely known optimizer is SGD (stochastic gradient decent).
- When we have any optimization problem, we must have an optimization objective (lets; consider we have SGD objective)
- Here the objective of SGD is to minimize given function what we called loss function. So SGD update the model's weight in such a way to make this loss function as close to its value as possible.

Loss function

- One common loss function is MSE (mean squared error). There are many other loss functions, but our job is to decide which loss function to use.
- For example let's say, we want to train to classify weather images are either images of car or dogs. We will supply our model with image of cats and dogs along with labels of these images that state weather each image is cate or of a dog.

In literal sense, the output will consists of probabilities of cat or dog. For example, it may assign a 75% probability to image being cat and a 25% probability to it being dog. In this case the model is assigning a higher likelihood to the image being of cat than a dog.

Here if we think, this is very similar to human make decisions. Everything is a prediction.

The loss is the difference of error between what the network is predicting for the images versus the true label of the image. So SGD will try to minimize this error to make out model as accurate as possible in predictions

- Passing data through model, we are going to continue passing the same data over and over again. This process is repeatedly sending the same data through the network is considered ***training***. During this training process is when the model will actually learn.

How neural network Learn

- We know that each data point used for training is passed through the network. This pass through the network from input to output is called ***forward pass***, and resiting output depends on the weights at each connection inside the network.
- Once all the data points in our datasets have been passed through the network, we say that an **epoch** is complete.

Epoch refers to a single pass of entire datasets to the network during training

So what exactly does it mean for the model to learn ?

- Well remember, when the model I initialize, the network weights are set to arbitrary values.

We have also seen that, at the end of the network, the model will provide the output for given input.

- Once the out[ut is obtained, the loss can be computed for that specific output by looking at what model predicted versus the true label.

Gradient of loss function

- After loss is calculated, the gradient of this loss function is computed with respect to each of the weights within the network.

Note gradient is just a word for the derivative of function of several variables.

At this point we have calculated the loss of single output, and we calculate the gradient of that loss with respect to our single chosen weight. This calculation is done using a technique called back-propagation.

Once we have the value of the gradient loss of function, we can use this value to update the model;s weight. The gradient tells us which direction will move the loss the minimum, and our task is to move in a direction that lowers the loss and step closer to minimum value.

Learning rate

- we multiply gradient value by something called learning rate. A learning rate is usually between 0.01 to 0.0001
- It tells us how large a step we should take in the direction of the minimize.

Updating the new weights

- so multiply the learning rate with gradient and subtract this from the weight, which give us the new update value of this weight.

$$\text{new weight} = \text{old weight} - (\text{learning rate} * \text{gradient})$$

- In this discuss, we will focus on the weight to explain the concept but this same process is going to happen with each of the weights in the model each time data passes through it.

The only difference is that when the gradient of the loss function is computed, the value for the gradient is going to be different for each weight because the gradient is being calculated with respect to each weight.

So now imagine all these weights being iteratively updated with each epoch. The weights are going to be incrementally getting closer and closer to their optimized values while SGD works to minimize the loss function

The model is training

This updating of the weights is essentially what we mean when we say that the **model is learning**. It's learning what values to assign to each weight based on how those incremental changes are affecting the loss function. As the weights change, the network is getting smarter in terms of accurately mapping inputs to the correct output.

```
# Import librabries
import keras
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense
from keras.optimizers import Adam
from keras.metrics import categorical_crossentropy

# Next define model
model = Sequential([
    Dense(units=16, input_shape=(1,), activation='relu'),
    Dense(units=32, activation='relu'),
    Dense(units=2, activation='sigmoid')
])
```

- Let's look now at how this training is done with code in Keras

```
# Before we can train our model, we must compile it like so:
model.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# we fit our model to the data
model.fit(
    x=scaled_train_samples,
    y=train_labels,
    batch_size=10,
    epochs=20,
    shuffle=True,
    verbose=2
)
```

To the **compile()** function, we are passing the **optimizer**, the **loss function**, and the **metrics** that we would like to see. Notice that the optimizer we have specified is called ***Adam***. Adam is just a variant of SGD. Inside the Adam constructor is where we specify the learning rate, and in this case Adam(***learning_rate=.0001***), we have chosen 0.0001.

Finally, we fit our model to the data. ***Fitting the model to the data means to train the model on the data***. We do this with the following code:

scaled_train_samples is a numpy array that contain the training samples.

train_labels is a numpy array that contain label sample data

batch_size=10 specifies how many training samples should be sent to the model at once.

epochs=20 means that the complete training set (all the samples) will be passed to the model a total of 20 times.

shuffle=True indicates that the data should first be shuffled before being passed to the model.

verbose=2 indicates how much logging we will see as the model trains

-

- output look like this

```
Epoch 1/20 0s - loss: 0.6400 - acc: 0.5576
Epoch 2/20 0s - loss: 0.6061 - acc: 0.6310
Epoch 3/20 0s - loss: 0.5748 - acc: 0.7010
Epoch 4/20 0s - loss: 0.5401 - acc: 0.7633
Epoch 5/20 0s - loss: 0.5050 - acc: 0.7990
Epoch 6/20 0s - loss: 0.4702 - acc: 0.8300
Epoch 7/20 0s - loss: 0.4366 - acc: 0.8495
Epoch 8/20 0s - loss: 0.4066 - acc: 0.8767
Epoch 9/20 0s - loss: 0.3808 - acc: 0.8814
Epoch 10/20 0s - loss: 0.3596 - acc: 0.8962
Epoch 11/20 0s - loss: 0.3420 - acc: 0.9043
Epoch 12/20 0s - loss: 0.3282 - acc: 0.9090
Epoch 13/20 0s - loss: 0.3170 - acc: 0.9129
Epoch 14/20 0s - loss: 0.3081 - acc: 0.9210
Epoch 15/20 0s - loss: 0.3014 - acc: 0.9190
Epoch 16/20 0s - loss: 0.2959 - acc: 0.9205
Epoch 17/20 0s - loss: 0.2916 - acc: 0.9238
Epoch 18/20 0s - loss: 0.2879 - acc: 0.9267
Epoch 19/20 0s - loss: 0.2848 - acc: 0.9252
Epoch 20/20 0s - loss: 0.2824 - acc: 0.9286
```

output give us

- Epoch number
- Loss
- Accuracy
- here you will notice that loss is going down where accuracy is going up as the epoch approach

.

Quiz

1. The ***batch size*** specifies the number of samples that are passed through an artificial neural network at one time.
2. The pass through a neural network that moves from input to output is called a *forward pass*, and the resulting output depends on the ***weights*** at each connection inside the network.
3. After all the data points in the training set have been passed through a neural network, we say that ***an epoch*** has completed.
4. In neural network programming, an *epoch* refers to a single pass of the entire training set through the network during training. ***True***
5. After the loss of a neural network has been calculated, the gradient of the loss function is computed with respect to each of the ***weights*** inside the network
6. The word *gradient* is a fancy word for the *derivative* of a function of several variables. ***True***
7. After we've calculated the loss of a single output, we calculate the gradient of that loss with respect to our single chosen weight. This calculation is done using a technique called ***back-propagation***.
8. In neural network programming, the gradient of the loss function allows us to determine which direction will move the loss towards the minimum. ***True***
9. In neural network programming, the learning rate tells us how large of a step we will take in the direction of the minimum when the network's weights are updated. ***True***
10. In neural network programming, updating of the weights is essentially what we mean when we say that the model is *learning*. ***True***

Loss in neural network's

- The loss function is what SGD is attempting to minimize by iterating updating the weights in the network.
- At the end of each epoch during the training process, the loss will be calculated using the network's predictions and the true labels for the respective input.

Suppose if we have to classify the image of cat and dog, and assume that the label for cat is **0** and dog is **1**.

Now suppose we pass an image of cat to the model, and the provided output is 0.25. In this case the difference the model's prediction and true label is 0.25 – 0.00 = 0.25

error = 0.25 – 0.00 = 0.25

This process is performed for every output. For each epoch, the error is accumulated across all the individual outputs.

Let's look at a loss function that is commonly used in practice called the *mean squared error* (MSE).

Mean squared error (MSE)

- For a single sample, with MSE, we first calculate the difference (the error) between the provided output prediction and the label. We then square this error. For a single input, this is all we do

MSE = output – label

- If we passed multiple samples to the model at once (a batch of samples), then we would take the mean of the squared errors over all of these samples.

Loss functions in code with Keras

```
# Before we can train our model, we must compile it like so:
model.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# we fit our model to the data
model.fit(
    x=scaled_train_samples,
    y=train_labels,
    batch_size=10,
    epochs=20,
    shuffle=True,
    verbose=2
)
```

- *Looking at the second parameter of the call to* compile (), *we can see the specified loss function* **loss='sparse_categorical_crossentropy**'.

In this example, we're using a loss function called sparse categorical crossentropy, but there are several others that we could choose, like MSE, for instance.

The currently available loss functions for Keras are as follows:

mean_squared_error

mean_absolute_error

mean_absolute_percentage_error

mean_squared_logarithmic_error

squared_hinge

hinge

categorical_hinge

logcosh

categorical_crossentropy

sparse_categorical_crossentropy

binary_crossentropy

- kullback_leibler_divergence
- poisson

- cosine_proximit

Quiz

1. *At the end of each epoch of training a neural network, the loss is calculated by comparing the model's **predictions** to the true labels*
2. *In neural network programming, the loss function is what SGD is attempting to minimize by iteratively updating the weights inside the network. **True***
3. *In neural network programming, the loss from a given sample is also referred to as the error. **True***
4. *In neural network programming, if we pass batches to our network during training, the loss will be calculated per batch. **True***

Learning rate and neural network

- *We know that the objective during training is for SGD to minimize the loss between the actual output and the predicted output from our training samples. The path towards this minimized loss is occurring over several steps.*

Recall that we start the training process with arbitrarily set weights, and then we incrementally update these weights as we move closer and closer to the minimized loss.

Now, the size of these ***steps*** we're taking to reach our minimized loss is going to depend on the learning rate. Conceptually, we can think of the learning rate of our model as the *step size*.

Before going further, let's first pause for a quick refresher. We know that during training, after the loss is calculated for our inputs, the gradient of that loss is then calculated with respect to each of the weights in our model.

Once we have the value of these gradients, this is where the idea of our learning rate comes in. The gradients will then get multiplied by the learning rate

gradients * learning rate

Updating the network's weights

new weight = old weight - (learning rate * gradient)

- *The value we choose for the learning rate is going to require some **testing**. The learning rate is another one of those hyperparameters that we have to test and tune with each model before we know exactly where we want to set it, but as mentioned earlier, a typical guideline is to set it somewhere between* 0.01 *and* 0.0001.

When setting the learning rate to a number on the higher side of this range, we risk the possibility of overshooting. This occurs when we take a step that's too large in the direction of the minimized loss function and shoot past this minimum and miss it.

To avoid this, we can set the learning rate to a number on the lower side of this range. With this option, since our steps will be really small, it will take us a lot longer to reach the point of minimized loss.

Overall, the act of choosing between a higher learning rate and a lower learning rate leaves us with this kind of trade-off idea.

Alright, so now we should have an idea about what the learning rate is and how it fits into the overall process of training.

```
# Before we can train our model, we must compile it like so:
model.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# we fit our model to the data
model.fit(
    x=scaled_train_samples,
    y=train_labels,
    batch_size=10,
    epochs=20,
    shuffle=True,
    verbose=2
)
```

In the above the learning_rate is set as 0.0001 inside the Adam optimizer as a parameter, this is optional. If we don't set it then Keras will automatically set it by default.

Another way to set learning_rate

model.optimizer.learning_rate = 0.01

Quiz

1. During the training process of a neural network, the learning rate determines, in part, the size of the **adjustments made to the weights**.

2. In neural network programming, the size of the steps we take to reach our minimized loss depends on the **learning rate**.
3. To obtain a particular updated weight value, we **subtract** the product of the gradient and the learning rate.
4. The learning rate is a hyperparameter. **True**
5. In a code implementation, the object that typically needs to have the learning rate is the **optimizer**.

Train, Test, & Validation Sets explained

- *For training and testing purposes for our model, we should have our data broken down into three distinct datasets. These datasets will consist of the following:*

Training set

Validation set

Test set

Training set

- *It's the set of data used to train the model.*
- *During each epoch, our model will be trained over and over again on this same data in our training set, and it will continue to learn about the features of this data*
- *The hope with this is that later we can deploy our model and have it accurately predict on new data that it's never seen before*
- *It will be making these predictions based on what it's learned about the training data*

Validation set

- ***The validation set allows us to see how well the model is generalizing during training.***
- *Validation dataset is separate from the training set. It is used to validate our model during training. This validation process help give give information that may assist us with adjusting our hyperparameters*

With the each epoch during training, model will be trained data of training set. Well it will also simultaneously be validated using data of validation set.

Then model will classify the output for each input of training datasets. After this loss will be calculated and weight will be adjusted in the model. Then during the next epoch it will classify the same input again.

- *Now, also during the training model will be classifying each input from the validation set as*

*well. It will be doing this classification based only on what it's learned about the data it's being trained on in the training set. The weights will not be updated in the model based on the loss calculated from our validation data (**Remember, the data in the validation set is separate from the data in the training set. So when the model is validating on this data, this data does not consist of samples that the model already is familiar with from training**.)*

- One of the major reasons we need a validation set is to ensure that our model is not over-fitting to the data in the training set.

- During training, if we're also validating the model on the validation set and see that the results it's giving for the validation data are just as good as the results it's giving for the training data, then we can be more confident that our model is not overfitting.

On the other hand, if the results on the training data are really good, but the results on the validation data are lagging behind, then our model is overfitting. Now let's move on to the test set.

Training and validation datasets are labeled

Test Set

- *Test set is unlabeled datasets.*
- *The test set is a set of data that is used to test the model after the model has already been trained.*

The test set is separate from both the training set and validation set.

- After our model has been trained and validated using ***training and validation sets***, we will then use our model to predict the output of the unlabeled data in the test set.
- When model predict on unlabeled data in our test set, this would be the same type of process that would be used if we were to deploy our model out into the field
- ***The test set provides a final check that the model is generalizing well before deploying the model to production***
- *The ultimate goal of machine learning and deep learning is to build models that are able to*

generalize well

Quiz

1. *The test set differs from the training and validation sets by* ***being passed to the model after training with no labels****.*
2. *During the training process of neural network, the model will be classifying each input from the training and validation sets. The classification will be based only on what the network has learned about the data from* ***training set.***
3. *One of the major reasons we need a validation set when training a neural network is to ensure that our model is not* ***over-fitting*** *to the data in the training set.*

Predicting with neural network

- *After the model training is completed, the next step is o predict on the data in our test.*
- *When we pass the test data to the model, we do not pass the corresponding labels, So model is not aware of the labels for the test set all.*

Passing samples with no labels

- *For prediction we pass our unlabeled data to the model , and model will think about each sample in our test data. Those predictions are occurring based on what model learned during training.*
- *Predictions are based on what the model learned during training.*
- **Example**,

suppose we trained a model to classify different breeds of dogs based on dog images. For each sample image, the model outputs which breed it thinks is most likely.

Now, suppose our test set contains images of dogs our model hasn't seen before. We pass these samples to our model, and ask it to predict the output for each image. Remember, the model does not have access to the labels for these images. This process will tell us how well our model performs on data it hasn't seen before based on how well its predictions match the true labels for the data. This process will also help to give some insight on what our model has or hasn't learned.

For example, suppose we trained our model only on images of large dogs, but our test set has some images of small dogs. When we pass a small dog to our model, it likely isn't going to do well at predicting what breed the dog is, since it's not been trained very well on smaller dogs in general.

This means that we need to make sure that our training and validation sets are representative of the actual data we want our model to be predicting on.,

Deploying the model in the real world (production)

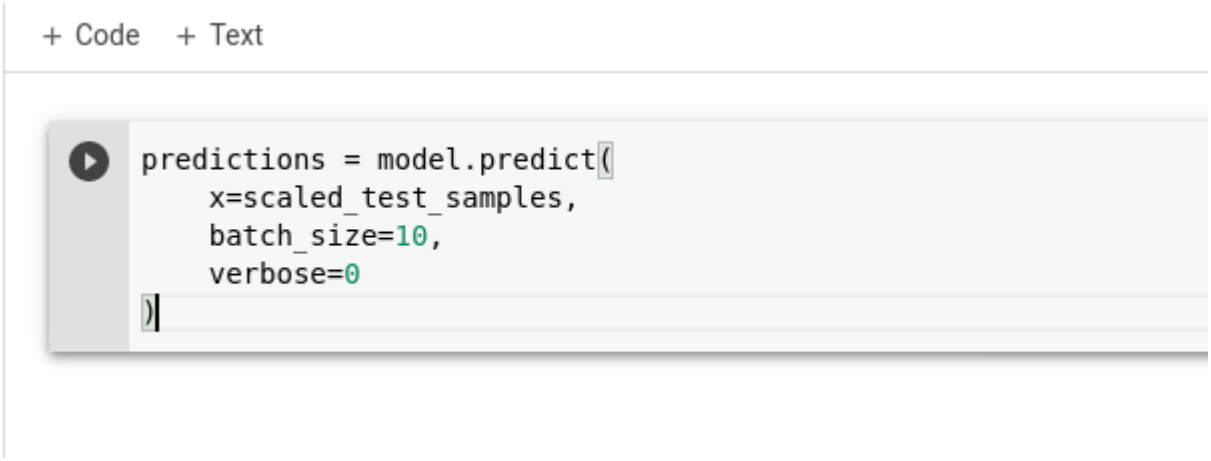
- *Aside from prediction on test data, we can have our model predict on real world data once it's deployed to serve its actual purpose.*

- *For example:-* if we deployed this neural network for classifying dog breeds to a website that anyone could visit and upload an image of their dog, it would be predicting the breed of the dog based on the image.

This image would likely not have been one that was included in our training, validation, or test sets, so this prediction would be occurring with true data from out in the field.

Using a Keras model to get a prediction

- *Suppose we have the following code.*



*Here at first we have variable called **predictions**. We are assuming that we already have model built and trained. Our model in this example is the object called **model**. Then are assigning the value of **model.predict()** to **predictions** variable.*

*This **predict()** function is called to actually make the predictions. To the **predict()** function, we are passing the variable called **scaled_test_sample**. This is the variable that is holding our test data.*

*We set **batch_size()** here arbitrarily to **10**. we have the verbosity, which is how much we want to see printed to the screen when we run these predictions, to **0** here to show nothing.*

Output

```
for p in predictions:
    print(p)

[ 0.7410683  0.2589317]
[ 0.14958295  0.85041702]
...
[ 0.87152088  0.12847912]
[ 0.04943148  0.95056852]
```

Quiz

1. To make predictions, the model relies on what is learned during training.
2. Suppose we trained a model *only* on images of large dogs, but our test set has some images of small dogs. When we pass a small dog to our model, we should expect the model to perform poorly.
3. In neural network programming, the training and validation sets should be representative of the *actual data* the model will be predicting on.
4. It is possible to get a prediction from a neural network model *before* the network has been trained.

Over-fitting in neural network's

- Overfitting occurs when our model becomes really good at being able to classify or predict on data that was included in the training set, but is not good at classifying data that it wasn't trained on. So essentially, the model has overfitt the data in the training set.

How to stop Overfitting ?

- When specify a validation set during training, we get metrics for the validation accuracy and loss, as well as the training accuracy and loss.
- If the validation metrics are considerably worse than the training metrics, then that is indication that our model is overfitting.
- We can also get an idea that our model is overfitting if during training, the model metrics were good, but when we use the model to predict on test it does not accurately classify the data in the test set.
- It has learned the features of the training set extremely well, but if we give the model any data that slightly deviates from the exact data used during training, it is unable to generalize and accurately predict the output.
-

How to Reduce overfitting

- Overfitting is common issue, so how can we reduce this?
- **Adding more data to the training set**
- **Data augmentation**

The general idea of data augmentation allows us to add more data to our training set that is similar to the data that we already have, but is just reasonably modified to some degree so that it's not the exact same.

For example, if most of our dog images were dogs facing to the left, then it would be a reasonable modification to add augmented flipped images so that our training set would also have dogs that faced to the right

Reduce the complexity of the model

Something else we can do to reduce overfitting is to reduce the complexity of our model. We could reduce complexity by making simple changes, like removing some layers from the model, or reducing the number of neurons in the layers. This may help our model generalize better to data it hasn't seen before.

Dropout

The last tip we'll cover for reducing overfitting is to use something called *dropout*. The general idea behind dropout is that, if you add it to a model, it will randomly ignore some subset of nodes in a given layer during training, i.e., it *drops out* the nodes from the layer. Hence, the name *dropout*. This will prevent these dropped out nodes from participating in producing a prediction on the data.

This technique may also help our model to generalize better to data it hasn't seen before.

Quiz

1. In neural network programming, overfitting occurs when a model becomes really good at being able to classify or predict on data that is included in the training set but is *not* as good at classifying data that it wasn't trained on. **True**
2. In neural network programming, if the validation metrics are considerably worse than the training metrics, then this is indication that our model is **overfitting**
3. In the realm of neural networks, the concept of *overfitting* boils down to the fact that the model is unable to **generalize well**.
4. *Data augmentation* is a technique that can be used to reduce overfitting. **True**
5. The general idea of data augmentation involves adding more data to the training set that is similar to the data that we already have but is just reasonably modified to some degree so that it's *not the exact same*. **True**

Underfitting in neural network

- A model is said to be *underfitting* when it's not able to classify the data it was trained on
- we can also said underfitting if the that training accuracy of the model is low and/or the training loss is high.

Reducing underfitting

- **Increase the complexity of model**

One thing we can do is increase the complexity of our model. This is the exact opposite of a technique we gave to reduce overfitting. If our data is more complex, and we have a relatively simple model, then the model may not be sophisticated enough to be able to accurately classify or predict on our complex data.

We can increase the complexity of our model by doing things such as:

Increasing the number of layers in the model.
Increasing the number of neurons in each layer.
Changing what type of layers we're using and where.

- **Add more features to the input samples**

Another technique we can use to reduce underfitting is to add more features to the input samples in our training set if we can. These additional features may help our model classify the data better.

For example, say we have a model that is attempting to predict the price of a stock based on the last three closing prices of this stock. So our input would consist of three features:

day 1 close
day 2 close
day 3 close

If we added additional features to this data like, maybe the opening prices for these days, or the volume of the stock for these days, then perhaps this may help our model learn more about the data and improve its accuracy

- ***Reduce dropout***

The last tip we'll discuss about reducing underfitting is to reduce dropout. Again, this is exactly opposite of a technique we gave in a previous post for reducing overfitting.

As mentioned in that post, dropout, which we'll cover in more detail at a later time, is a regularization technique that randomly ignores a subset of nodes in a given layer. It essentially prevents these dropped out nodes from participating in producing a prediction on the data.

Quiz

1. Underfitting occurs when a model is **unable to classify data in the training set**.

Supervised Learning

- Labels are used to supervise or guide the learning process.
- Training, validation, and testing sets, we explained that both the training data and validation data are labeled when passed to the model. *This is the case for supervised learning*.
- With supervised learning, each piece of data passed to the model during training is a pair that consists of the input object, or sample, along with the corresponding label or output value.
- Essentially, with supervised learning, the model is learning how to create a mapping from given inputs to particular outputs based on what it's learning from the labeled training data.

Labels are numeric

- To do this, the labels need to be encoded into something numeric. In this case, for example if we have to classify the lizard and turtle from images then label of *lizard* may be encoded as 0, whereas the label of *turtle* may be encoded as 1
- After this, we go through this process of determining the error or loss for all of the data in our training set for as many epochs as we specify.

Remember, during this training, the objective of the model is to minimize the loss, so when we deploy our model and use it to predict on data it wasn't trained on, it will be making these predictions based on the labeled data that it did see during training

Working with labeled data in Keras

```
# weight, height
train_samples = np.array([
    [150, 67],
    [130, 60],
    [200, 65],
    [125, 52],
    [230, 72],
    [181, 70]
])

# 0: male
# 1: female
train_labels = np.array([1, 1, 0, 1, 0, 0])

model.fit(
    x=train_samples,
    y=train_labels,
    batch_size=3,
    epochs=10,
    shuffle=True,
    verbose=2
)
```

```
# import libraries
import keras
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense
from keras.optimizers import Adam
import numpy as np

# suppose we have sequential model with two hideen layers and 2 ouput
model = Sequential([
    Dense(units=16, input_shape=(2,), activation='relu'),
    Dense(units=32, activation='relu'),
    Dense(units=2, activation='sigmoid')
])

model.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

- At first, we have imported some libraries
- then we are using sequential model with two hidden layers and expect 2 output categories
- We're assuming the task of this model is to classify whether an individual is ***male*** or ***female*** based on his or her ***height*** and ***weight***.
- In **training_sample** variables we have stored the the training datasets. Here, we have a list of pairs, and each of these pairs is an individual sample, and a sample is the weight and height of a person. The first element in each pair is the weight measured in ***pounds***, and the second element is the height measured in *inches*

- Next, we have our **labels** data stored in this **train_labels** variable. Here, a **0** represents a male, and a **1** represents a female.
- The position of each of these labels corresponds to the positions of each sample in our **train_samples** variable. For example, this first **1** here, which represents a **female**, is the label for the first element in the **train_samples** array. This second **1** in **train_labels** correspond to the second sample in **train_samples**, and so on.
- Now, when we go to train our model, we call **model.fit()** as we've discussed in previous posts, and the first parameter here specified by **x** is going to be our **train_samples** variable, and the second parameter, specified by y, is going to be the corresponding **train_labels**

Unsupervised Learning

- Unsupervised learning occurs with unlabeled data.
- each piece of data passed to our model during training is solely an unlabeled input object, or sample. There is no corresponding label that's paired with the sample.

If the data isn't labeled, then how is the model learning? How is it evaluating itself to understand if it's performing well or not?

Since the model is unaware of the labels for the training data, there is no way to measure accuracy. Accuracy is not typically a metric that we use to analyze an unsupervised learning process

- Here the model is going to be given an unlabeled dataset, and it's going to attempt to learn some type of ***structure*** from the data and will ***extract the useful information or features from this data***

Clustering algorithms

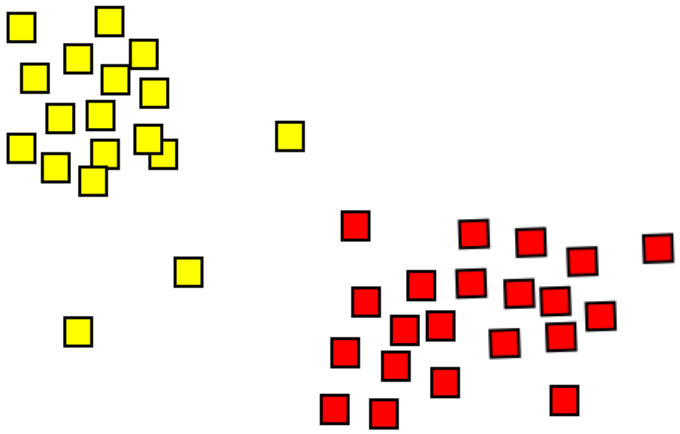
- One of the most popular applications of unsupervised learning is through the use of *clustering algorithms*. Sticking with our example from our previous post on supervised learning, let's suppose we have the *height* and *weight* data for a particular age group of *males* and *females*.

This time, we don't have the labels for this data, so any given sample from this data set would just be a pair consisting of one person's height and weight. There is no associated label telling us whether this person was a male or female.

Now, a clustering algorithm could analyze this data and start to learn the structure of it even though it's not labeled. Through learning the structure, it can start to cluster the data into groups.

We could imagine that if we were to plot this height and weight data on a chart, then maybe it would look something like this with weight on the x-axis and height on the y-axis.

There's nothing explicitly telling us the labels for this data, but we can see that there are two pretty distinct clusters here,



and so we could infer that perhaps this clustering is occurring based on whether these individuals are male or female.

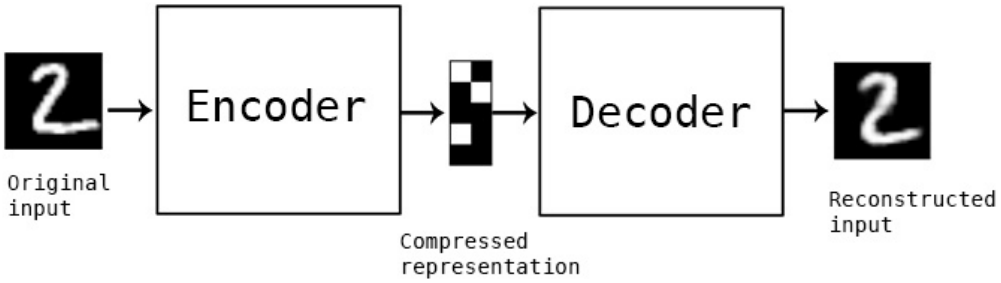
One of these clusters may be made up predominately of females, while the other is predominately male, so clustering is one area that makes use of unsupervised learning. Let's look at another

Autoencoders

Unsupervised learning is also used by *autoencoders*.

In the most basic terms, an autoencoders is an artificial neural network that takes in input, and then outputs a reconstruction of this input.

Based on everything we've learned so far on neural networks, this seems pretty strange, but let's explain this idea further using an example



Suppose we have a set of images of handwritten digits, and we want to pass them through an autoencoder. Remember, an autoencoder is just a neural network.

This neural network will take in this image of a digit, and it will then *encode* the image. Then, at the end of the network, it will *decode* the image and output the decoded reconstructed version of the original image.

The goal here is for the reconstructed image to be as close as possible to the original image

Applications of autoencoders

- one application for this could be to denoise images.

Once the model has been trained, then it can accept other similar images that may have a lot of noise surrounding them, and it will be able to extract the underlying meaningful features and *reconstruct* the image without the noise

- Autoencoders output a ***reconstruction of the input***

Semi-supervised learning

- Semi-supervised learning kind of takes a middle ground between supervised learning and unsupervised learning.
- *Semi-supervised learning* uses a combination of supervised and unsupervised learning techniques, and that's because, in a scenario where we'd make use of semi-supervised learning, we would have a combination of both *labeled* and *unlabeled* data.
- If we have large datasets, then we could go through and manually label some portion of this large data set ourselves and use that portion to train our model.

However, if we have access to large amounts of data, and we've only labeled some small portion of this data, then what a waste it would be to just leave all the other unlabeled data on the table.

Pseudo-labeling

- As just mentioned, we've already labeled some portion of our data set. Now, we're going to use this labeled data as the training set for our model. We're then going to train our model, just as we would with any other labeled data set

Just through the regular training process, we get our model performing pretty well, and so everything we've done up to this point has been regular old, supervised learning in practice.

Now here's where the unsupervised learning piece comes into play. After we've trained our model on the labeled portion of the data set, we then use our model to predict on the remaining unlabeled portion of data, and we then take these predictions and label each piece of unlabeled data with the individual outputs that were predicted for them

Quiz

- Semi-supervised learning employs ***pseudo-labeling*** to create labels for the remaining unlabeled data
- Using pseudo labeling, the unlabeled portion of the data gets labeled via ***predictions from a model that was trained on the labeled portion of data***

Data Augmentation

- Data augmentation occurs when we create new data by modifications of our existing data.

For example, we could augment image data by flipping the images, either horizontally or vertically. We could rotate the images, zoom in or out, crop, or even vary the color of the images. All of these are common data augmentation techniques

why use data augmentation?

- *For example*, say we have a relatively small number of samples to include in our training set, and it's difficult to get more. Then we could create new data from our existing data set using data augmentation to create more samples.

- To reduce overfitting

Think about if we had a data set full of images of dogs, but most of the dogs were facing to the right.

If a model was trained on these images, it's reasonable to think that the model would believe that only these right-facing dogs were actually dogs. It may very well not classify left-facing dogs as actually being dogs when we deploy this model in the field or use it to predict on test images.

With this, producing new right-facing images of dogs by augmenting the original images of left-facing dogs would be a reasonable modification. We would do this by horizontally flipping the original images to produce new ones.

One hot encoding

Labels

- We know that when we're training a neural network via supervised learning, we pass labeled input to our model, and the model gives us a predicted output.
- If our model is an image classifier, for example, we may be passing labeled images of animals as input. When we do this, the model is usually not interpreting these labels as words, like *dog* or *cat*. Additionally, the output that our model gives us in regards to its predictions aren't typically words like *dog* or *cat* either. Instead, most of the time our labels become encoded, so they can take on the form of an integer or of a vector of integers

Hot and cold values

One type of *encoding* that is widely used for encoding categorical data with numerical values is called *one-hot encoding*.

One-hot encoding transform our categorical labels into vectors of **0s** and **1s**. The length of these vectors is the number of classes or categories that our model is expected to classify.

0 => Cold

1 => Hot

Vectors of 0s and 1s

- If we were classifying whether images were either of a dog or of a cat, then our one-hot encoded vectors that corresponded to these classes would each be of length 2 reflecting the two categories.
- If we added another category, like lizard, so that we could then classify whether images were of dogs, cats, or lizards, then our corresponding one-hot encoded vectors would each be of length 3 since we now have three categories.
- Alright, so we know the labels are transformed or *encoded* into vectors. We know that each of these vectors has a length that is equal to the number of output categories, and we briefly mentioned that the vectors contain 0s and 1s. Let's go into further detail on this last piece.

One-hot encoding for multiple categories

- Let's stick with the example of classifying images as being either of a *cat*, *dog*, or *lizard*. With each of the corresponding vectors for these categories being of length 3, we can think of each index or each element within the vector corresponding to one of the three categories.
- Let's say for this example that the cat label corresponds to the first element, dog corresponds to the second element, and lizard corresponds to the third element.
- With each of these categories having their own *place* in the corresponding vectors, we can now discuss the intuition behind the name *one-hot*.
- With each one-hot encoded vector, every element will be a zero EXCEPT for the element that corresponds to the actual category of the given input. This element will be a *hot one*
- Sticking with our same example, recall we said that a cat corresponded to the first element, dog to the second, and lizard to the third, so the corresponding one-hot encoded vectors for

Label	Index-0	Index-1	Index-2
Cat	1	0	0
Dog	0	1	0
Lizard	0	0	1

each of these categories would look like this.

One vector for each category

- Similarly, for dog, we see that the second element is a one, while the first and third elements are zeros. Lastly, for lizard, the third element is a one, while the first and second elements are zeros.

We can see that each time the model receives input that is a cat, it's not interpreting the label as the word *cat*, but instead is interpreting the label as this vector **[1,0,0]**.

For images labeled as dog, the model is interpreting the dog label as the vector **[0,1,0]**, and for images labeled as lizard, the model is interpreting the label as the vector **[0,0,1]**.

Label	Vector
Cat	[1,0,0]
Dog	[0,1,0]
Lizard	[0,0,1]

- Just for clarity purposes, say we add another category, llama, to the mix. Now, we have four categories total, and so this will cause each one-hot encoded vector corresponding to each of these categories to be of length **4** now.

The vectors will now look like this

Label	Vector
Cat	[1,0,0,0]
Dog	[0,1,0,0]
Lizard	[0,0,1,0]
Llama	[0,0,0,1]

Quiz

1. In supervised learning, the index of the *"hot one"* corresponds to a(n) ***label***.