Deep learning is an approach to machine learning characterized by deep stacks of computations. Through their power and scalability **neural networks** have become the defining model of deep learning. Neural networks are composed of neurons, where each neuron individually performs only a simple computation.

The Linear Unit

⇒ 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.

⇒ A neural network "learns" by modifying its weights.

⇒ The weights are set to random numbers (and the bias to 0.0). A neural network learns by finding better values for its weights.
model.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. 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.

⇒ We can just add more input connections to the neuron, one for each additional feature.

⇒ It's often useful to start with a single neuron model as a baseline. **Single neuron models are linear models.**

Linear Units in Keras¶

The easiest way to create a model in Keras is through keras.Sequential, which creates a neural network as a stack of 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])
])
```

⇒ We've seen how a linear unit computes a linear function -- now we'll see how to combine and modify these single units to model more complex relationships.

Neural networks typically organize their neurons into layers.

⇒ You could think of each layer in a neural network as performing some kind of relatively simple transformation. In a well-trained neural network, each layer is a transformation getting us a little bit closer to a solution.

⇒ What we need is something nonlinear. What we need are activation functions.

An **activation function** is simply some function we apply to each of a layer's outputs (its activations). -> ReLU

⇒ The layers before the output layer are sometimes called **hidden** since we never see their outputs directly.

⇒ Now, notice that the final (output) layer is a linear unit (meaning, no activation function). That makes this network appropriate to a regression task, where we are trying to predict some arbitrary numeric value.

⇒ Other tasks (like classification) might require an activation function on the output.

```
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),
])
```

There is a whole family of variants of the 'relu' activation -- 'elu', 'selu', and ‘swish’. The ReLU activation tends to do well on most problems, so it's a good one to start with.

In addition to the training data, we need two more things:

- A "loss function" that measures how good the network's predictions are.
- An "optimizer" that can tell the network how to change its weights.

The **loss function** measures the disparity between the the target's true value and the value the model predicts.

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)

Besides MAE, other loss functions you might see for regression problems are the mean-squared error (MSE) or the Huber loss (both available in Keras).

In other words, the loss function tells the network its objective.

We've described the problem we want the network to solve, but now we need to say how to solve it. This is the job of the **optimizer**. The optimizer is an algorithm that adjusts the weights to minimize the loss.

Virtually all of the optimization algorithms used in deep learning belong to a family called **stochastic gradient descent**.

⇒ Sample some training data and run it through the network to make predictions.

⇒ Measure the loss between the predictions and the true values.

⇒ 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

⇒ A smaller learning rate means the network needs to see more minibatches before its weights converge to their best values.

⇒ The learning rate and the size of the minibatches are the two parameters that have the largest effect on how the SGD training proceeds.

Fortunately, for most work it won't be necessary to do an extensive hyperparameter search to get satisfactory results. **Adam is an SGD algorithm that has an adaptive learning rate that makes it suitable for most problems without any parameter tuning.**

With the learning rate and the batch size, you have some control over:

- How long it takes to train a model
- How noisy the learning curves are
- How small the loss becomes

⇒ Smaller batch sizes gave noisier weight updates and loss curves. This is because each batch is a small *sample* of data and smaller samples tend to give noisier estimates.

⇒ Smaller learning rates make the updates smaller and the training takes longer to converge. Large learning rates can speed up training, but don't "settle in" to a minimum as well. When the learning rate is too large, the training can fail completely.

```
model.compile(
optimizer="adam",
loss="mae",
)
```

We've told Keras to feed the optimizer 256 rows of the training data at a time (the batch_size) and to do that 10 times all the way through the dataset (the epochs).

```
history = model.fit(
X_train, y_train,
validation_data=(X_valid, y_valid),
batch_size=256,
epochs=10,
)
```

Overfitting and Underfitting

You might think about the information in the training data as being of two kinds: signal and noise. The signal is the part that generalizes, the part that can help our model make predictions from new data. The noise is all of the random fluctuation that comes from data in the real-world or all of the incidental, non-informative patterns.

The training loss will go down either when the model learns signal or when it learns noise. But the validation loss will go down only when the model learns signal.

Ideally, we would create models that learn all of the signal and none of the noise.

There can be two problems that occur when training a model: not enough signal or too much noise.

⇒ **Underfitting** the training set is when the loss is not as low as it could be because the model hasn't learned enough signal.

⇒ **Overfitting** the training set is when the loss is not as low as it could be because the model learned too much noise.

The trick to training deep learning models is finding the best balance between the two.

A model's **capacity** refers to the size and complexity of the patterns it is able to learn. For neural networks, this will largely be determined by how many neurons it has and how they are connected together.

If it appears that your network is underfitting the data, you should try increasing its capacity.

We mentioned that when a model is too eagerly learning noise, the validation loss may start to increase during training. To prevent this, we can simply stop the training whenever it seems the validation loss isn't decreasing anymore. Interrupting the training this way is called **early stopping**.

In Keras, we include early stopping in our training through a callback. A **callback** is just a function you want run every so often while the network trains.

```
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.”

```
history = model.fit(

callbacks=[early_stopping] )
```

Dropout and Batch Normalization

There are two kinds of special layers, not containing any neurons themselves, but that add some functionality that can sometimes benefit a model in various ways.

Dropout¶

The first of these is the "dropout layer", which can help correct overfitting.

To recognize these spurious patterns a network will often rely on very a specific combinations of weight, a kind of "conspiracy" of weights. Being so specific, they tend to be fragile.

This is the idea behind **dropout**. To break up these conspiracies, 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.

In Keras, the dropout rate argument rate defines what percentage of the input units to shut off. Put the Dropout layer just before the layer you want the dropout applied to.

```
keras.Sequential([
# ...
layers.Dropout(rate=0.3), # apply 30% dropout to the next layer
layers.Dense(16) ])
```

Moreover, by making it harder for the network to fit spurious patterns, dropout may have encouraged the network to seek out more of the true patterns, possibly improving the validation loss some as well).

The next special layer we'll look at performs **"batch normalization"** which can help correct training that is slow or unstable.

With neural networks, it's generally a good idea to put all of your data on a common scale, perhaps with something like scikit-learn's StandardScaler or MinMaxScaler. The reason is that SGD will shift the network weights in proportion to how large an activation the data produces. Features that tend to produce activations of very different sizes can make for unstable training behavior.

Now, if it's good to normalize the data before it goes into the network, maybe also normalizing inside the network would be better! In fact, we have a special kind of layer that can do this, the **batch normalization layer**.

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.

Models with batchnorm tend to need fewer epochs to complete training.

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

Binary Classification

So far in this course, we've learned about how neural networks can solve regression problems. Now we're going to apply neural networks to another common machine learning problem: classification.

Before using this data we'll assign a **class label**: one class will be 0 and the other will be 1.

⇒ **Accuracy and Cross-Entropy¶**

Accuracy is one of the many metrics in use for measuring success on a classification problem. Accuracy is the ratio of correct predictions to total predictions: accuracy = number_correct / total. A model that always predicted correctly would have an accuracy score of 1.0.

The problem with accuracy is that it can't be used as a loss function. So, we have to choose a substitute to act as the loss function. This substitute is the cross-entropy function.

Now, recall that the loss function defines the objective of the network during training. With regression, our goal was to minimize the distance between the expected outcome and the predicted outcome. We chose MAE.

For classification, what we want instead is a distance between probabilities, and this is what cross-entropy provides. **Cross-entropy** is a sort of measure for the distance from one probability distribution to another.

The idea is that we want our network to predict the correct class with probability 1.0. The further away the predicted probability is from 1.0, the greater will be the cross-entropy loss.

The cross-entropy and accuracy functions both require probabilities as inputs, meaning, numbers from 0 to 1. We attach a new kind of activation function, the sigmoid activation.

To get the final class prediction, we define a threshold probability. Typically this will be 0.5, so that rounding will give us the correct class: below 0.5 means the class with label 0 and 0.5 or above means the class with label 1.

A 0.5 threshold is what Keras uses by default.

```
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'),
])

model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['binary_accuracy'],
)
```

MORE THINGS TO DO

- Classify images with TPUs in Petals to the Metal
- Create art with GANs in I'm Something of a Painter Myself
- Classify Tweets in Real or Not? NLP with Disaster Tweets
- Detect contradiction and entailment in Contradictory, My Dear Watson