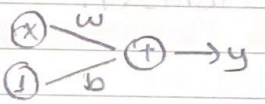
What is Deep Learning?

* Deep L. is an approach to machine learning characterized by deep stacks of computation.
* 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 single computation. The power of a neural network comes instead from the complexity of the connections these neurons can form

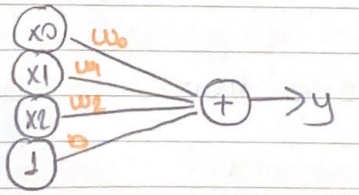
The Linear Unit



$y = wx + b$

* A neural network learns by modifying its weight.
* When we take an input first we multiply this with weight.
* b: Special kind of weight we call bias. The bias doesn't have any input data associated with it; instead we put a "1" in the diagram so the value that reaches the neuron is just b. The bias enables the neuron to modify the output independently of its inputs

Multiple Inputs



$\Rightarrow y = w_0x_0 + w_1x_1 + w_2x_2 + b$

Linear Units in Keras

* keras.Sequential = 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])])
```

* Units = define how many outputs we want
* Input_shape = dimension of input (number of features we will use)

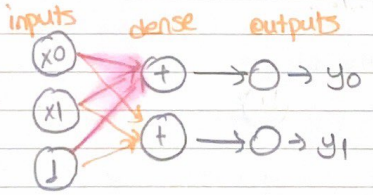
* Internally keras represents the weights of a neural network with tensors. Tensors are basically TensorFlow's version of a Numpy array

* We can take weight with;
  ⇒ w, b = model.weights (or model.get_weights())

DEEP NEURAL NETWORKS

Layers

* Neural networks typically organise their neurons into layers. When we collect together linear units having a common set of inputs we get a dense layer.



* In a well trained neural network each layer is a transformation getting us a little bit closer to solution

Activation Functions

* 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.

⇒ ImageNet Data ?

Stacking Dense Layers

* The layers before the output layer are sometimes called hidden since we never see their outputs directly.
* The final output layer is a linear unit (no activation function). That makes the network appropriate to a regression task. Other tasks might require an activation function on the output.

Building Sequential Models

```
⇒ model = keras.Sequential([
        # 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),
    ])
```

~~Stochastic Gradient Descent~~ TRAINING

* Training the network means adjusting its weights in such a way that it can transform the features into the target.
* In addition to train a data we need two thing:
    1) A "loss function" that measures how good the network's predictions are
    2) An "optimizer" that can tell the network how to change its weights

The Loss Function

* " " " measures the disparity between the target's true value and prediction.
* During training a model loss function is a guide for finding the correct values of its weights (lower loss better !!!)

The Optimizer - Stochastic Gradient Descent

* How to solve the problem → Optimizers.
* Virtually all of the optimization algorithms used in deep learning belong to family called Stochastic gradient descent. They are iterative alg. that train a network in steps. One step of traing:
    1) Sample some training data and run it through the network to make predictions.
    2) Measure the loss
    3) Adjust the weights in a direction that makes the loss smaller

* Each iteration's sample of training data is called a **minibatch** (or just batch) while a complete round of the training data is called an **epoch**. The number of epoch → how many time network see training data.

Learning Rate and Batch Size

* Smaller learning rate → network need to see more batches before it weights converge to their best values.

NOTE! Learning rate and batch size are two param have largest effect on SGD training.

* **Adam**: is an SGD alg. that has an adaptive learning rate that makes is suitable for most problems without any parameter tuning. It is great general-purpose optimiser.

Adding The Loss and Optimiser

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

Fitting

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

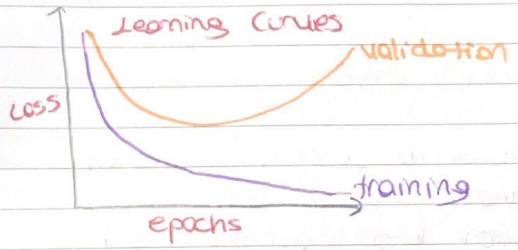
* With the learning rate and the batch size, we have some control over:

1) How long it takes the train a model.
2) How noisy the learning curves are
3) " small the loss becomes.

Interpreting Learning Curves

* Information in the training data as being of two kinds: signal and noise. The signal is the part that can help our model make predictions from new data. The noise is the part that is only true of the training data.

* To accurately assess a model's performance, we need to evaluate it on a new set of data, the validation data.



* The validation loss gives an estimate of the expected error on unseen data.
* Validation loss will go down only when the model learns signals.

Capacity

* A model's capacity refers to the size and complexity of the patterns it is able to learn. For neural networks, this determined by how many neurons it has and how they are connected. If there is underfitting, we should increase capacity
* we can increase capacity:
    1) making it wider (more units to existing layers)
    2) " " deeper (adding more layers)
* Wider networks have an easier time learning more linear relationships, while deeper networks prefer more nonlinear ones.

Early Stopping

* whenever validation loss not decreasing anymore, we interrupt with early stopping
* Training with early stopping also means we are in less danger of stopping training too early

Adding Early Stopping

* A callback ⇒ we want run while network trains. The early stopping callback will run after every epoch

⇒ from tensorflow.keras.callbacks import EarlyStopping

early_stop = EarlyStopping(min_delta = 0.001, # iyileştirme olarak sayılacak min değişiklik miktarı
patience = 20 #epoch
restore_best_weights = True,)

* These params says "If there is at least 0.001 improvement in validation loss after 20 epochs stop training, keep the best model"

Fitting with Early Stopping

⇒ history = model.fit(X_train, y_train,
batch_size = 256,
epochs = 900,
callbacks = [early_stop])

DROP OUT AND BATCH NORMALIZATION

DROP OUT LAYER

* Can help correct overfitting
* To break up conspiracies ([redacted]), 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 noise patterns in the training data
* We could also think dropout as creating a kind of ensemble networks

⇒ keras.Sequential([..... , layer.Dropout(rate=0.3),
layer.Dense(16)...])

* Rate ⇒ apply %30 dropout to the next layer

BATCH NORMALIZATION

* Helps correct training that is slow or unstable

NOTE! In neural networks scaling is important

* A batch normalization layer (batchnorm) looks at each batch it comes in, first normalizing the batch with its own mean and std dev, and then also putting the data on a new scale with two trainable rescaling parameters.
* Most often it is added as an aid to the optimization process.
* Models with batchnorm tend to need fewer epochs to complete training.

⇒
```
layers.Dense(16),
layers.BatchNormalization(),
layer.Activation("relu")
```

* If we add it as the first layer, it would be like standart scaler.