The Perceptron (1958)

- A single "neuron" (unit) aka Threshold Logic Unit (TLU)

- **Transfer Function**
  - T is the Threshold value (assume $T = 0$)
  
  \[
  I = \sum_{i=1}^{i=n} w_i x_i \\
  y = \begin{cases} 
  +1, & \text{if } I \geq T \\
  -1, & \text{if } I < T 
  \end{cases}
  \]
Learning in Neural Networks (so far)

- A **labelled dataset**
  e.g., Iris Dataset, MNIST Numbers Dataset

- A **model** for the network
  e.g., the Perceptron (single TLU)
  **Parameters** refer to the number of weights

- A **forward pass/prediction algorithm**
  Compute net input. Compute activation.

- A **Learning Rule/Algorithm**
  Learning occurs by changing weights.
  Specifies change in the weights using the Error/Loss.
  In Perceptrons, weights are changed after every prediction.

- Several **epochs** of training are needed (how many???)
  This is a **hyperparameter**.

### Perceptron Learning Rule

\[ y = \begin{cases} 
  +1, & \text{if } I \geq T \\
  -1, & \text{if } I < T 
\end{cases} \]

\[ I = \sum_{i=1}^{n} w_i x_i \]

Introducing Bias

- Instead of using an arbitrary Threshold value, we can turn it into an input (=1)

- The weight on the bias, \( w_0 \) can then be learned using the same algorithm.

\[
\vec{w} = [w_0, w_1, w_2] \\
\vec{x} = [x_0, x_1, x_2]
\]

- More often, in other networks, the net input is determined using the following (and no bias is used for output layer):

\[ I = \sum_{i=1}^{n} w_i x_i + \bar{b} \]
Multi-Layer Perceptron Network

- **Example**: This could be a network that can recognize all three categories of irises from the Iris dataset.

  4 inputs, 3 outputs (Hyperparameters)
  4x4 (input to hidden) + 4x3 (hidden to output) weights + 7 bias inputs

  \#Parameters = 16+12+7 = 35

- Since all units are linear TLUs this network can only learn linear functions.

- We need to make each unit non-linear.

---

Backpropagation Network (Classic Version)

- **Net Input**

  \[
  I = \sum_{i=1}^{n} w_i x_i + b
  \]

- **Activation Function** (Sigmoid)

  \[
  f(I) = \frac{1}{1+e^{-I}}
  \]

- **Learning Rule**

  \[
  \Delta w_{ij} = \beta \cdot E \cdot f(I_j)
  \]

- **Error/Loss**

  \[
  E_{j}^{\text{output}} = y_j^{\text{desired}} - y_j^{\text{actual}}
  \]

  \[
  E_{l}^{\text{hidden}} = \frac{df(I_l^{\text{hidden}})}{dI} \sum_{j=1}^{n} (w_{lj}E_{j}^{\text{output}}) 
  \]
Backpropagation Network (Classic Version)

- **Net Input**
  \[ I = \sum_{i=1}^{n} w_i x_i + b \]

- **Activation Function (Sigmoid)**
  \[ f(I) = \frac{1}{1 + e^{-I}} \]

- **Learning Rule**
  \[ w_{ij}^{new} = w_{ij}^{old} + \beta \times E \times f'(I_j) \]
  where \( \beta \) is the Learning Constant (0 < \( \beta \) < 1), \( E \) is the error/loss

- **Error/Loss (E)**
  \[ E_{output}^{j} = y_{j}^{desired} - y_{j}^{actual} \]
  \[ E_{hidden}^{i} = \frac{df^{hidden}}{dI} \sum_{j=1}^{n} (w_{ij}E_{output}^{j}) \]

Blob Diagram:  
- **N inputs**
- **Hidden Layer**
- **Output Layer**
- \( Y_j^{actual} \) is the actual output
- \( w_{ij} \) is the weight between layer \( i \) and \( j \)

From: https://en.wikipedia.org/wiki/Sigmoid_function

- Sigmoid Function
  - Output is always between 0 and 1
  - Its derivative (slope) is always positive
  - \( \frac{df}{dI} = f(I)(1 - f(I)) \)
Backpropagation Network (Classic Version)

- **Net Input**
  \[ I = \sum_{i=1}^{i=n} w_i x_i + b \]

- **Activation Function (Sigmoid)**
  \[ f(I) = \frac{1}{1 + e^I} \]

- **Learning Rule**
  \[ w_{ij}^{new} = w_{ij}^{old} + \beta \cdot E \cdot f(I_j) \]
  where \( \beta \) is the Learning Constant \((0 < \beta < 1)\), \( E \) is the error/loss

- **Error/Loss (E)**
  \[ E_{output}^{j} = y_{desired}^{j} - y_{actual}^{j} \]
  \[ E_{hidden}^{i} = \frac{dE_{output}^{j}}{dl} \sum_{j=1}^{n} (w_{ij}E_{output}^{j}) \]
**Backpropagation (Classic) Training Algorithm**

set minimum acceptable error and #epochs to train
set #epochs = 0

repeat
  total error = 0
  for each pattern in training set do
    do a forward pass
      for each unit in the hidden layer do
        compute net input I, and activation f(I)
        save f(I) for backpropagation
      for each unit in the output layer do
        compute net input I, and activation f(I)
        output y = f(I)
    do backward pass
      for each unit in the output layer do
        compute error = desired – actual output
        total error = total error + error
      for each unit in the middle layer do
        compute incoming error = weighted sum of output layer errors
        compute final error = incoming error * f(I) * (1 – I) (derivative)
      for each unit in the output layer do
        for each weight from a hidden layer to unit do
          compute weight change β * error * f(I) and update weight
      for each unit in the middle layer do
        for each weight from an input layer unit do
          compute weight change β * final error * f(I) and update weight
    #epochs = #epochs + 1
  until total error < maximum acceptable error or #epochs reaches limit

**Example: Recognizing Handwritten Digits**

- **MNIST Dataset**
  70,000 images (28x28 pixels), grayscale values (in range 0 (white) to 255 (black)).

- **Training set**: 60,000 images
- **Testing set**: 10,000 images

- **Task**: Given an image, classify it as [0,1,...,9]
MNIST Digit Recognition: The Design

• Assume we will use a 3-layer network: input, hidden, and output
• Input will be 28x28=784 units
• 10 outputs, one for each digit. Say the network is shown a 6, we would then expect the output of 6 to be high (closer to 1.0) compared to others.
• How many hidden units???

No known science to this. Lets say we have 512 units in hidden layer. (Why??)
MNIST Digit Recognition: The Design

- A 3-layer network: input, hidden, and output layers
- Input will be 28x28=784 units
- 10 outputs, one for each digit. 512 units in hidden layer. (Why??)
- All units will have a Sigmoid activation function.
- How to determine error/loss at output? (use our formulation?)
- What should be the value of $\beta$?

- How many parameters are there?
MNIST Digit Recognition: The Design

- A 3-layer network: input, hidden, and output layers
- Input will be 28x28=784 units
- 10 outputs, one for each digit. 512 units in hidden layer. (Why??)
- All units will have a Sigmoid activation function.
- How to determine error/loss at output? (use our formulation?)
- What should be the value of $\beta$?
- How many parameters are there?
  $784 \times 512 + 512 \text{ (bias)} + 512 \times 10 + 10 \text{ (Bias)} = 407,050$

What are the hyperparameters?
MNIST Digit Recognition: The Design

- A 3-layer network: input, hidden, and output layers
- Input will be 28x28 = 784 units
- 10 outputs, one for each digit. 512 units in hidden layer. (Why??)
- All units will have a Sigmoid activation function.
- How to determine error/loss at output? (use our formulation?)
- What should be the value of $\beta$?
- How many parameters are there?
  \[784 \times 512 + 512 (\text{bias}) + 512 \times 10 + 10 (\text{bias}) = 407,050\]
- What are the hyperparameters?
  
  # Layers, # Units in each layer, Activation Function, $\beta$, 
  # epochs, Minimum acceptable error, etc.

Backpropagation: Gradient Descent

- Learning in a neural network using Backpropagation is essentially a Gradient Descent process.
- Each change in the weights is an attempt to reduce error and descend into the lowest possible position in the “error bowl” (as shown in a 2-D weight vector case)
- In higher dimensional weight vectors (typical ML situations), the error surface can be quite complex.

From: https://builtin.com/data-science/gradient-descent
Backpropagation: Gradient Descent

• In higher dimensional weight vectors (typical ML situations), the error surface can be quite complex.

![Error surface diagram](https://poissonisfish.com/2023/04/11/gradient-descent/)

![Learning Rate Schedule](https://builtin.com/data-science/gradient-descent)

Backpropagation: Learning rate

• Learning Rate, $\beta$ ($0 < 1$)

• The value of $\beta$ determines how fast or slow the gradient descent takes place.

• Typically, one starts with a higher value (say 0.5 or 0.6) and then decrease it as the learning/epochs progresses. This is called a Learning Rate Schedule.

![Learning Rate Graphs](https://www.ibm.com/topics/gradient-descent)
Vocabulary

- In what order do we present the patterns? As they are in the training set? Or, randomly? If patterns are chosen at random (without replacement), we call it Stochastic Gradient Descent (SGD).

Choices:
- Do a backpropagation pass after every input. True SGD
- Do the backward pass after all the inputs have been seen, and errors recorded. Full batch SGD
- Do a small batch of data and the do a backward pass. Mini Batch SGD (each batch is a power of 2)
- Is there a better way to assess error/loss? Loss Functions
- Are there any other weight update mechanisms? Optimizers (also manage Learning rate schedules)

Most of the period from 1986 until now has been spent on studying these questions.

MNIST Digit Recognition: More Decisions

- In what order do we present the patterns? As they are in the training set? Or, randomly?
- Do we do a backpropagation pass after every input?

Choices:
- Do a backpropagation pass after every input.
- Do the backward pass after all the inputs have been seen, and errors recorded.
- Do a small batch of data and the do a backward pass.
- Is there a better way to assess error/loss?
- Are there any other weight update mechanisms?

Most of the period from 1986 until now has been spent on studying these questions.
Advances in NN

• **Better hardware**
  Laptops became 5000 times faster between 1990 and 2010
  Use of GPUs for faster processing (NVIDIA, AMD). Took off in 2011
  Google’s Tensor Processing Units (TPUs), 2016

• **Wide availability of datasets and benchmarks**
  MNIST, ImageNet, etc.

• **Better Algorithms**
  Better activation functions
  Better weight initialization schemes
  Better optimization schemes (RMSprop, Adam)

• **Widely available toolsets for creating and training NNs**
  Theano, TensorFlow, Scikit Learn, Keras

---

Backpropagation Network (Updated)

• **Net Input**
  \[ I = \sum_{i=1}^{n} w_i x_i + b \]

• **Activation Functions**
  Sigmoid, Relu, Softmax, tanh, exponential, etc.

• **Loss Functions**
  Binary Cross Entropy, Categorical Cross Entropy, Poisson, KL
  Divergence, Mean Squared Error, Mean Absolute Error, Cosine
  Similarity, etc.

• **Optimizer (Learning Rule)**
  SGD, RMSprop, Adam, Adadelta, Adamax, Namad, etc.
Popular Activation Functions

**Relu** – Rectified Linear Unit: \( f(I) = \max(0, I) \)

**Sigmoid**:

\[
f(I) = \frac{1}{1 + e^I}
\]

Vocabulary

- In what order do we present the patterns? As they are in the training set? Or, randomly? If patterns are chosen at random (without replacement), we call it **Stochastic Gradient Descent (SGD)**

**Choices**:

- Do a backpropagation pass after every input. **True SGD**
- Do the backward pass after all the inputs have been seen, and errors recorded. **Full batch SGD**
- Do a small batch of data and the do a backward pass. **Mini Batch SGD** (each batch is a power of 2)

- Is there a better way to assess error/loss? **Loss Functions**
- Are there any other weight update mechanisms? **Optimizers** (also manage Learning rate schedules)

Most of the period from 1986 until now has been spent on studying these questions.
The Learning Paradigm

How NN Learning Works

From: Chollet, 2021.
Introducing Keras

• Deep Learning API for Python (2016-17)
• Built on top of TensorFlow (2015)
• Can run on a typical CPU, or can be accelerated with specialized hardware, if available. GPUs (Graphics Processing Units), TPUs (Tensor Processing Units)
• Makes Neural Network design, implementation, and exploration akin to building with LEGOs!

Typical Keras Workflow

• Acquire, prepare, and load the dataset
  Keras has several predefined datasets available: MNIST Digits, CIFAR10, CIFAR100, IMDB Reviews for sentiment classification, Reuters Newswire classification, Fashion MNIST, Boston Housing price regression (see https://keras.io/api/datasets/)

• Design and Build the Model
  How many layers to use? How many units in each layer? What activation function to use? (see https://keras.io/api/layers/activations/)

• Compile the Model
  Decide which optimizer to use, loss function, accuracy metric
  https://keras.io/api/optimizers/, https://keras.io/api/losses/, https://keras.io/api/metrics/

• Train/Fit the Model
  Provide the training data and its labels, number of epochs to train, batch size

• Test/Validate the Model
  Use the test data to test how well the trained model performs
Over to Colab...

• See Lab for Recognizing Handwritten Digits

References