A lot of the buzz about ML recently has come from recent advancements in **deep learning**.

When people talk about “deep learning” they are generally talking about a class of models called **neural networks** that are a loose approximation of how our brains work.
Recall: Linear Classifier

Remember the linear classifier based on score

\[ \text{Score}(x) = w_0 + w_1 x[1] + w_2 x[2] + ... + w_d x[d] \]

Score(x) > 0

Score(x) < 0
Perceptron

Graphical representation of this same classifier

This is called a perceptron
What can a perceptrons represent?

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
XOR

The perceptron can learn most boolean functions, but XOR always has to ruin the fun.

This data is not linearly separable, therefore can’t be learned with the perceptron.
Neural Network

Idea: Combine these perceptrons in layers to learn more complex functions.
Notice that we can represent
\[ x[1] \text{ XOR } x[2] = (x[1] \text{ AND } \neg x[2]) \text{ OR } (\neg x[1] \text{ AND } x[2]) \]
This is a 2-layer neural network

\[ y = x[1] \text{ XOR } x[2] = (x[1] \text{ AND ! } x[2]) \text{ OR (! } x[1] \text{ AND } x[2]) \]

\[ v[1] = (x[1] \text{ AND ! } x[2]) 
    = g(-0.5 + x[1] - x[2]) \]

\[ v[2] = (! x[1] \text{ AND } x[2]) 
    = g(-0.5 - x[1] + x[2]) \]

\[ y = v[1] \text{ OR } v[2] 
    = g(-0.5 + v[1] + v[2]) \]
Two layer neural network (alt. one hidden-layer neural network)

Single

\[ \text{out}(x) = g \left( w_0 + \sum_j w_j x[j] \right) \]

1-hidden layer

\[ \text{out}(x) = g \left( w_0 + \sum_k w_k g \left( w_0^{(k)} + \sum_j w_j^{(k)} x[j] \right) \right) \]
A surprising fact is that a 2-layer network can represent any function, if we allow enough nodes in hidden layer.

For this example, consider regression function with one input.

Class Session
Before, we were using the sign activation function.

- This is not generally used in practice.
  - Not differentiable
  - No notion of confidence

Generally, people use different sigmoid functions as activation functions. For example, the logistic function

\[
g(w_0 + \sum_j w_j x[j]) = \frac{1}{1 + e^{-(w_0 + \sum_j w_j x[j])}}
\]
Sigmoid Functions

• **Sigmoid**
  - Historically popular, but (mostly) fallen out of favor
  - Neuron’s activation saturates (weights get very large -> gradients get small)
  - Not zero-centered -> other issues in the gradient steps
    - When put on the output layer, called “softmax” because interpreted as class probability (soft assignment)

• **Hyperbolic tangent** $g(x) = \tanh(x)$
  - Saturates like sigmoid unit, but zero-centered

• **Rectified linear unit (ReLU)** $g(x) = x^+ = \max(0,x)$
  - Most popular choice these days
  - Fragile during training and neurons can “die off”… be careful about learning rates
    - ”Noisy” or “leaky” variants

• **Softplus** $g(x) = \log(1+\exp(x))$
  - Smooth approximation to rectifier activation
Neural Networks

Generally layers and layers of linear models and non-linearities (activation functions).

Have been around for about 50 years

- Fell in “disfavor” in the 90s when simpler models were doing well

In the last few years, have had a huge resurgence

- Impressive accuracy on several benchmark problems
- Have risen in popularity due to huge datasets, GPUs, and improvements to
Are NNs likely to overfit? **YES.**
- Consequence of being able to fit any function!

How to avoid overfitting?
- Get more training data
- Few hidden nodes / better topology
  - **Rule of thumb:** 3-layer NNs outperform 2-layer NNs, but going deeper only helps if you are very careful (different story next time with convolutional neural networks)
- Regularization
  - Dropout
- Early stopping
Application to Computer Vision
Features in computer vision are local detectors

- Combine features to make prediction

In reality, these features are much more low level (e.g. Corner?)
A popular approach to computer vision was to make hand-crafted features for object detection.

Relies on coming up with these features by hand (yuck!)

Input → Extract features → Use simple classifier (e.g., logistic regression, SVMs)
Neural Networks implicitly find these low level features for us!

Each layer learns more and more complex features

<table>
<thead>
<tr>
<th>Example detectors learned</th>
<th>Example interest points detected</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Layer 1 example" /></td>
<td><img src="image2.png" alt="Layer 1 interest points" /></td>
</tr>
<tr>
<td><img src="image3.png" alt="Layer 2 example" /></td>
<td><img src="image4.png" alt="Layer 2 interest points" /></td>
</tr>
<tr>
<td><img src="image5.png" alt="Layer 3 example" /></td>
<td><img src="image6.png" alt="Layer 3 interest points" /></td>
</tr>
</tbody>
</table>

[Zeiler & Fergus ‘13]
Brain Break
You can use neural networks for classification and regression!

**Regression**
The output layer will generally have one node that is the output (outputs a single number)

**Classification**
The output layer will have one node per class. Usually take the node with the highest score as the prediction for an example. Can also use the logistic function to turn scores into probabilities!
Learning Coefficients

So the idea of neural networks might make sense, but how do we actually go about learning the coefficients in the layers?

First we need to define a quality metric or cost function

- For regression, generally use RSS or RMSE
- For classification, generally use something call the Cross Entropy loss.

Can we use gradient descent here? Actually yes!

- How do we take the derivative of a network?
- Are there convergence guarantees?
What does gradient descent do in general? Have the model make predictions and update the model in a special way such that the new weights have lower error.

To do gradient descent with neural networks, we generally use the **backpropagation algorithm**.

1. Do a forward pass of the data through the network to get predictions
2. Compare predictions to true values
3. Backpropagate errors so the weights make better predictions
Training NN

It’s pretty expensive to do this update for the entire dataset at once, so it’s common to break it up into small batches to process individually.

However, processing each batch only once isn’t enough. You generally have to repeatedly update the model parameters. We call an iteration that goes over every batch once an **epoch**.

```python
for i in range(num_epochs):
    for batch in batches(training_data):
        preds = model.predict(batch.data) # Forward pass
        diffs = compare(preds, batch.labels) # Compare
        model.backprop(diffs) # Backpropagation
```
Brain Break
Neural Networks have MANY hyperparameters

- How many hidden layers and hidden neurons?
- What activation function?
- What is the learning rate for gradient descent?
- What is the batch size?
- How many epochs to train?
- And much much more!

How do you decide these values should be? ¯\_(ツ)_/¯

The most frustrating thing is that we don’t have a great grasp on how these things impact performance, so you generally have to try them all.
Hyperparameter Optimization

How do we choose hyperparameters to train and evaluate?

Grid search:

Hyperparameters on 2d uniform grid
Hyperparameter Optimization

How do we choose hyperparameters to train and evaluate?

Grid search:

Hyperparameters on 2d uniform grid

Random search:

Hyperparameters randomly chosen
Hyperparameter Optimization

How do we choose hyperparameters to train and evaluate?

Grid search:

Hyperparameters on 2d uniform grid

Random search:

Hyperparameters randomly chosen

Bayesian Optimization:

Hyperparameters adaptively chosen
Recent work attempts to speed up hyperparameter evaluation by stopping poor performing settings before they are fully trained.

In general, hyperparameter optimization is a non-convex optimization problem where we know very little about how the function behaves.

Your time is valuable and compute time is cheap. Write your code to be modular so you can use compute time to try a range of values.

Tools for different purposes

- Very few evaluations: use random search (and pray)
- Few evaluations and long-run computations: See last slide
- Moderate number of evaluations: Bayesian optimization
- Many evaluations possible: Use random search. Why overthink it?
In general, loss functions with neural networks are not convex.

This means the backprop algorithm for gradient descent will only converge to a local optima.

Like with k-means, this means that how you initialize the weights is really important and can impact the final result.

How should you initialize weights? 🙃
- Usually people do random initialization

All the same rules apply from gradient descent with a learning rate, you might miss the mark of the local optima if the step size is too large.