These lectures are based on two excellent, free online books (links on syllabus page):

- Neural Networks and Deep Learning by Michael Nielsen
- Deep Learning by Goodfellow, Bengio and Courville
Data is sent in as mini-batches, after each mini-batch Z is adjusted. After all the mini-batches have been "learned", the final machine Z_{final} is run on test data to see how often it yields correct answers.
- Many possible structures for MNIST, etc.
- We first cover the big math picture for a push forward, "fully connected net"

- Each layer is represented by a function $F_i$. It inputs $\tilde{x} \in \mathbb{R}^{n_i}$ and outputs $\tilde{z} \in \mathbb{R}^{n_{i+1}}$

$$
\begin{array}{ccc}
\vdots & \vdots & \vdots \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\vdots & \vdots & \vdots \\
A_{l} & A_{l} & A_{l} \\
\text{neurons or nodes} & n_{l} & n_{l+1}
\end{array}
$$

- $F_i(x, A_i, \tilde{b}_l)$ depends on the parameters

  - $A_i$ an $(n_{l+1} \times n_{l})$ matrix of weights so $A: \mathbb{R}^{n_{l}} \rightarrow \mathbb{R}^{n_{l+1}}$
  - $\tilde{b}_l$ an $(n_{l+1} \times 1)$ vector = $\mathbb{R}$ bias
  - an activation function $\tau$
\[ F(x, A_l, b_L) = \sigma(A_l x + b_L) \]

The activation function has various forms:

- **Step function**

- **Sigmoid**
  \[ \sigma(z) = \frac{1}{1 + e^{-z}} \]

- **Ramp or ReLU**
  \[ \sigma(z) = \max(z, 0) \]
The activation function is vectorized, i.e., it acts on each component of a vector.

So \( \tau(z_1, \ldots, z_n) = (\tau(z_1), \ldots, \tau(z_n)) \)

and \( F_i : \mathbb{R}^n \rightarrow \mathbb{R}^{n+1} \)

Putting the pieces together from many layers

\[
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 0 & 0 \\
\vdots & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{array}
\]

\( G_m = F_k \circ F_{k-1} \circ \ldots \circ F_1 \), \( z = (a_1, b_1, a_2, b_2, \ldots, a_k, b_k) \)

all the weights and bias together (lots of parameters!)
Now we train the machine with training data \( \tilde{x}_1, \ldots, \tilde{x}_p \) which are correctly characterized as \( \tilde{y}_1, \ldots, \tilde{y}_n \).

Now throw in all the training data and construct the cost or objective or error function

\[
\ell_{\tilde{y}} (\tilde{x}_1, \ldots, \tilde{x}_n) = \frac{1}{N} \sum_{i=1}^{N} \left( \frac{1}{2} \left( C_{\tilde{y}_i} (\tilde{x}_i) - \tilde{y}_i \right) \right)^2
\]

(This is simplest, least squares version. More sophisticated versions later).

We treat this as a function of \( \tilde{y} \) and use an optimization routine to diminish \( \ell_{\tilde{y}} \) to \( \ell_{\tilde{y}} \).

Repeat with \( \tilde{y} \to \tilde{y}' \).
In practice, a random subset of training data is thrown in, $\mathcal{M}$ is adjusted, then another minibatch, etc.

- The goal is to get a $\mathcal{M}_2$ given by $\mathcal{G}_2$ that generalizes, i.e., works well on test data that is not in the training set.

- A big issue is how much to optimize $\mathcal{M}_2$ for just the training set. Don't want the machine to memorize the training data and not generalize to other test data.

- This is called overfitting.

- Now the question is why $\mathcal{M}_2$ takes this form?

- This is connected to why it is called a neural net.
It will be easier to understand if you watch the video on YouTube by Marc Dymony.

https://www.youtube.com/watch?v=6qS8wD29Py

Lecture MLB

Artificial neurons and a single layer net.
- We first describe a simple artificial neuron called "the perceptron."

- The output is zero or one (fire or don't fire)
- The neuron weights the input using weights $w_1$, $w_2$, and $w_3$
- A threshold $-b$ is set (minus sign explained later)
- Also called the bias
- Rule: output is zero if
  \[ w_1 x_1 + w_2 x_2 + w_3 x_3 \leq -b \]

  output is one if
  \[ w_1 x_1 + w_2 x_2 + w_3 x_3 > -b \]

- Example: you are trying to decide whether to do your math HW tonight
  - \( x_1 \): how close is the due date
  - \( x_2 \): how long is the HW
  - \( x_3 \): what your friends are doing tonight

- You weigh up these various factors and make a decision \( 0 = \text{no}, 1 = \text{yes} \).

- We want to express the decision process more succinctly.
- Let \( \sigma(z) = 0 \) if \( z \leq 0 \)
  \( = 1 \) if \( z > 0 \)
  activation function

- Let \( F(x) = \sigma(\mathbf{w}^T \mathbf{x} + b) \) with
  \[ \mathbf{w}^T = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \]
  \[ \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \]
  then \( F(x) = 0 \Rightarrow \text{No} \)
  \( = 1 \Rightarrow \text{Yes} \)

- Now we want to model a more complicated decision or classification problem using multiple perceptrons
The k-th perceptron has weights $\vec{w}_k$ and threshold or bias $b_k$.

We get for each $k$

$$z_k = \sum \left( \vec{w}_k \vec{x} + b_k \right) \quad k=1,...,m \quad m=\#d \text{ neurons}$$

We want to combine all these into a matrix form.

Let $W = \begin{bmatrix} \vec{w}_1 & \ldots & \vec{w}_m \end{bmatrix}$, $b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$.

Then

$$W^T \vec{x} + \vec{b} = \begin{bmatrix} \vec{w}_1^T \\ \vdots \\ \vec{w}_m^T \end{bmatrix} \vec{x} + \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

$$= \begin{bmatrix} \vec{w}_1^T \vec{x} + b_1 \\ \vdots \\ \vec{w}_m^T \vec{x} + b_m \end{bmatrix} = \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix}$$
The last step is to vectorize the activation $f$.

So our one layer machine is described by $A = W^T \Delta \theta$. The weight matrix

$$f(x) = \sigma(Ax + b)$$

We now change our point of view on the one layer perceptron and treat it as a learning machine.
Lecture MLC
Learning and Multiple Layers
So we return to the characterization problem. Let $x_1, \ldots, x_p$ be inputs with correct outputs $y_1, \ldots, y_p$. We fix a value of the parameters (the weights and bias) $A, b$. Then the machine outputs $\hat{y}_1, \ldots, \hat{y}_p$ and for each $i$, this has least squares error $F(x_i) = \sqrt{(Ax_i + b - \hat{y}_i)^2}$. For each $i$, this has least squares error $E_i = \sqrt{(Ax_i + b - \hat{y}_i)^2}$. For the total error over all inputs, we average these:

$$E = \frac{1}{N} \sum_{i=1}^{N} (Ax_i + b - \hat{y}_i)^2$$
• Now we optimize, i.e., find \( A_f \) and \( b_f \) which minimize \( \Phi(A, b) \).

• We now declare our final machine to be \( \bar{y} = A_f \bar{x} + b_f \).

• Note the similarity to least squares and polynomial fitting.

• Looking more closely, how do we optimize?

• Usual thing is to differentiate \( \Phi \) with respect to \( A \) and \( b \), etc.

• But \( J \) is not differentiable.
- Another issue with T is that it is restrictive, binary output.

- Back to the HW example:

  - How close is the due date?
  - How long is the HW?
  - What are my friends doing tonight?

  How much time should I spend on HW?

The output is not just 0 or 1, but a number.
So we probably want $T$ at least continuous or maybe differentiable. We want small changes in the output when parameter changes. The sigmoid $T(z) = \frac{1}{1 + e^{-z}}$ is nice and differentiable but computationally expensive. The ramp $T(z) = \max(z, 0)$ is continuous, its "derivative" is $\text{sign}(z)$ which is not so bad. It is computationally cheap.

For now, let $T$ be the sigmoid for theoretical ease.
Let's study optimization or learning for one level.

\[ F(x, A, b) = \nabla (A x + b) \]

Let's have three inputs and 2 neurons:

\[ x_1 \rightarrow z_1 \]
\[ x_2 \rightarrow z_2 \]
\[ x_3 \]

So \( z_1 = \nabla (w_1 x_1 + w_2 x_2 + w_3 x_3 + b_1) \)

\( z_2 = \nabla (w_{21} x_1 + w_{22} x_2 + w_{23} x_3 + b_2) \)

If the true values are \( y_1 \) and \( y_2 \) we have

\[ \Phi = \frac{1}{2} \left( (\nabla (w_{11} x_1 + w_{12} x_2 + w_{13} x_3 + b_1) - y_1)^2 + (\nabla (w_{21} x_1 + w_{22} x_2 + w_{23} x_3 + b_2) - y_2)^2 \right) / 2 \]
we want to minimize the error $\Phi$ as a function of the parameters $\text{the w's and b's}$.

So we treat $\Phi$ as a function of these and compute $\nabla \Phi$ and find critical points
and see if they are local maximum or saddle.

For example,

$$\nabla \Phi = \begin{bmatrix} \frac{\partial \Phi}{\partial w_1} & \cdots & \frac{\partial \Phi}{\partial w_k} \\ \frac{\partial \Phi}{\partial b_1} & \cdots & \frac{\partial \Phi}{\partial b_k} \end{bmatrix}$$

\(\text{(same argument)} - y_{-1}\)

with

$$\frac{\partial \Phi}{\partial w_i} = \left( \text{same argument} \right) \\ \frac{\partial \Phi}{\partial b_i} = \left( \text{same argument} \right) - 1$$

by the Chain Rule.

This is complicated for just this simple one layer but we need many layers with many neurons and maybe thousands of parameters.
So we need new ideas

(1) A better optimization scheme namely Gradient Descent

(2) A clever way of computing $\nabla \Phi$ when there are many layers

- We will cover each of these in more detail in later lectures

- Now to finish the introduction we describe multiple layers - this is the "Deep" in deep learning

- One way to think of this is decision making in stages
For example, first you decide how much time to allot to your math HW tonight, then you decide what order to fit it in with your other HW.

Each layer is given by a function

\[ F_i (\hat{x}, A_i, \vec{b}_L) = \nabla (A_i x + \vec{b}_L) \]
- The layers act sequentially:

\[ F_f \text{ then } F_2 \text{ then } F_3 \ldots F_L \]

- Mathematically, this is a composition (recall it is written in the reverse order):

\[ F = F_L \circ F_{L-1} \circ \ldots \circ F_2 \circ F_1 \]

- The least squares error is:

\[ \Phi = \sum_{i=1}^{N} ||F(x_i) - y_i||^2 \quad \text{and} \]

\[ \nabla \Phi \]

- It depends on all the \( A_i \) and \( b_i \), so \( \nabla \Phi \) is a chore to compute.

- This net is called "Feed Forward" since information just flows in one direction:

\[ \text{Input} \rightarrow \text{Output} \]