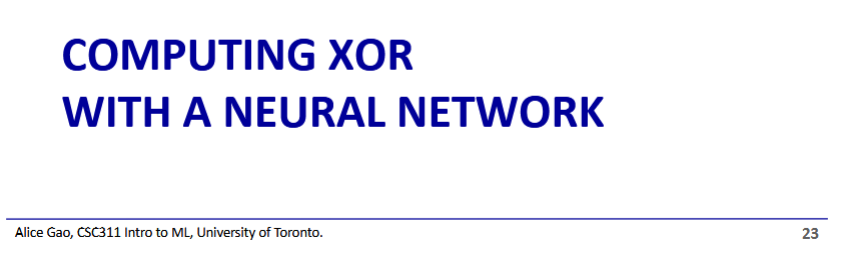
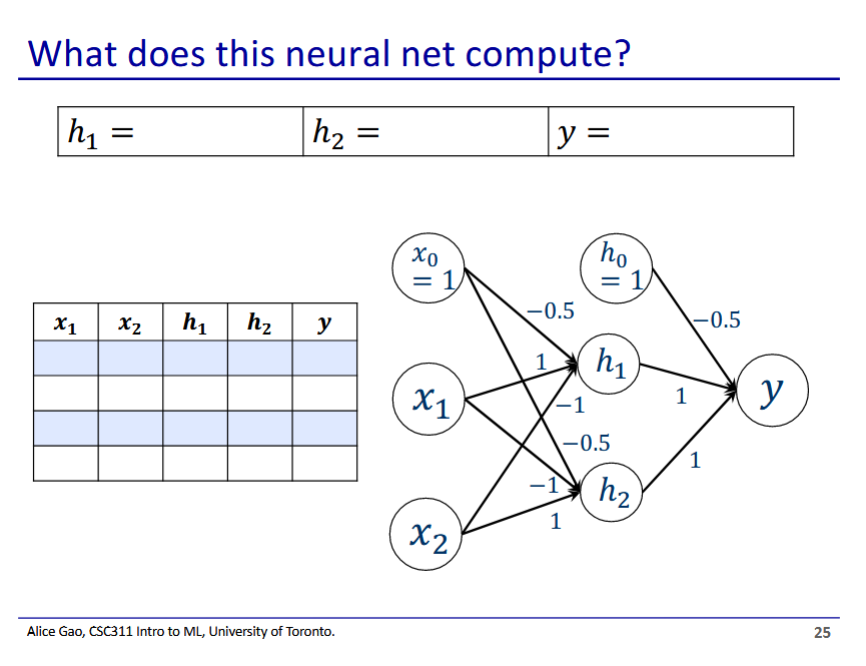
| **Designing a neural network by hand**   * Example on slide 23 (Computing XOR with a neural network) of neural networks lecture cont’d   + **There will be a question like this on the assignment and test**   **Neural networks as universal function approximators**   * Deep neural networks are universal function approximators   + Can approximate any function to arbitrarily well * The universality of neural networks comes from the non-linear activation function   + If the activation function was linear (deep linear network), the neural network can only represent linear functions     - The output would be   + Only with non-linear activation functions (threshold, logistic, tanh, ReLU) can non-linear functions be represented using a deep neural network * **Proof for universality for binary input/output functions**   + Proof for this subset of functions since general proof is too complicated   + Proof on slides 38-39   + Proof also demonstrates some limitations of universality     - May overfit     - Neural network may need to be very large   **Backpropagation**   * Gradient descent is possible for neural networks, we need an extra algorithm (backpropagation)   + The process is the same as other models (calculate gradients, apply gradients, repeat)   + However neural networks use a ton of weights     - This makes computing gradients take a while, and in high dimensions * Backpropagation is a more efficient way to calculate the gradients   + When using the chain rule, there is considerable overlap in the derivatives calculated     - and have an overlap of     - We could try to avoid making these repeated calculations   + **Computation graph**      - Graph that represents which values are dependent on which other values       * Tracing a path through the graph from weight to loss function gives the chain rule to calculate that part of the gradient     - If we move backwards through the graph, we can calculate the derivatives that are common for multiple paths       * We first calculate , then * **Process**   + We first run through the computation graph forwards, filling in the values of z, y, and L   + We then backpropagate through the graph to calculate the derivatives * **Notation (Error signals):**   + Unofficial notation, but we will use this for this course   + Let , this shows that is a physical value that we are trying to compute   **Multivariate chain rule**   * In more complicated neural networks, there will be multiple paths that lead from the weights to the loss function (in the computation graph)    + Multivariate chain rule will be needed to calculate these derivatives * We add together the chain rule for both paths to get the derivative |
| --- |

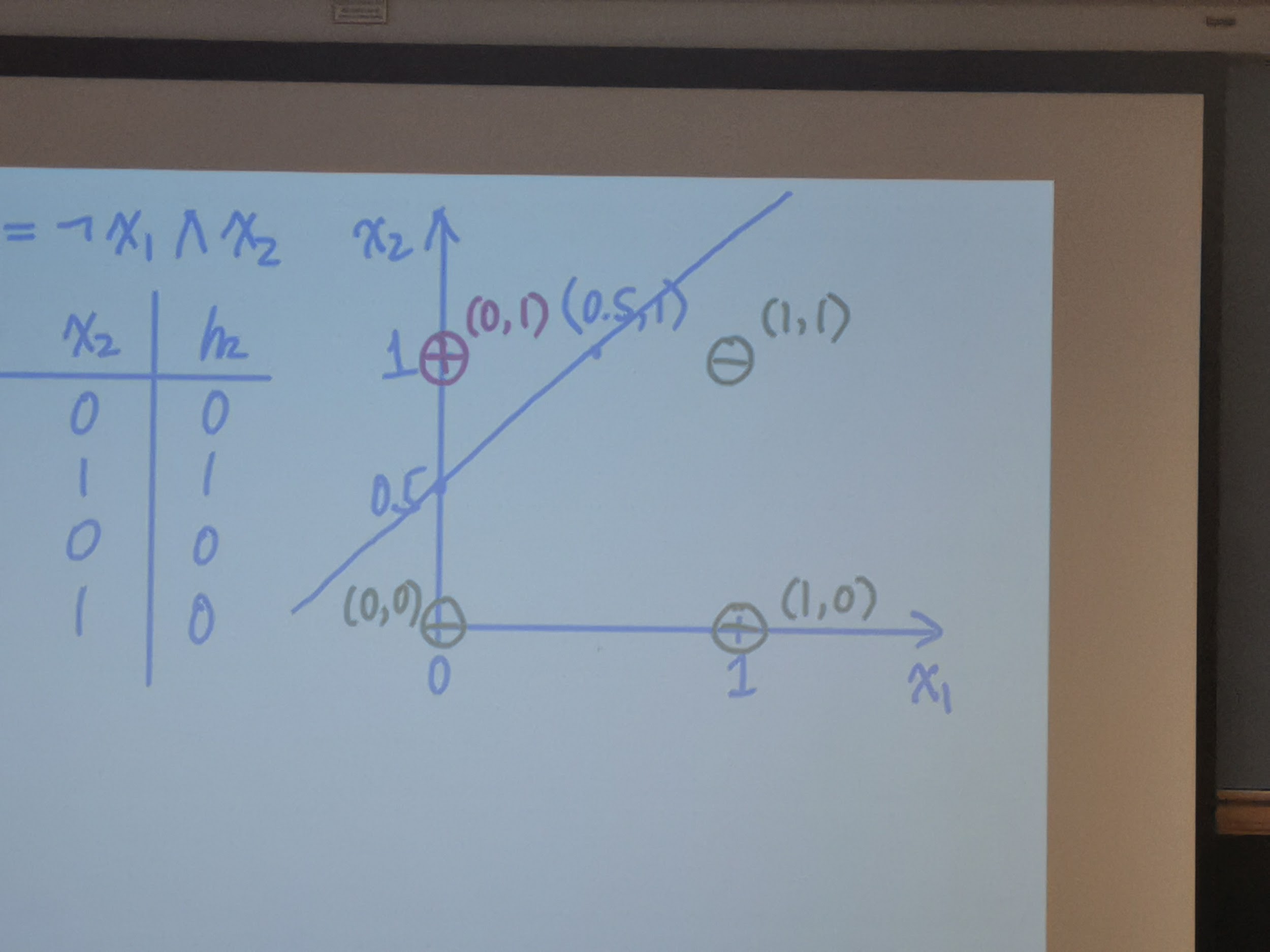
  


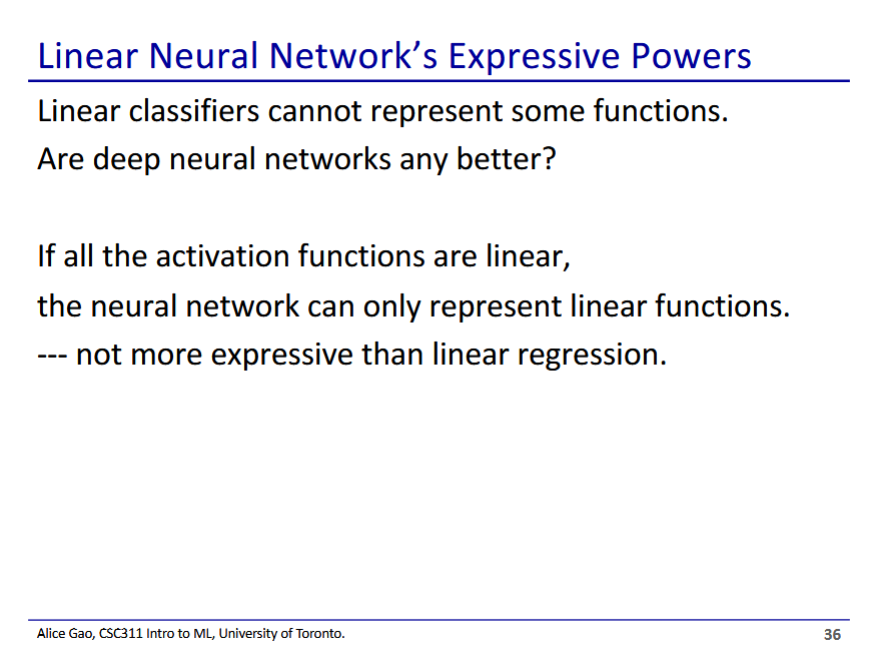
| x\_1 | x\_2 | h\_1 | h\_2 | y |
| --- | --- | --- | --- | --- |
| 0 | 0 |  |  | 0 |
| 0 | 1 |  |  | 1 |
| 1 | 0 |  |  | 1 |
| 1 | 1 |  |  | 0 |

* h\_1 calculates
* h\_2 calculates
* y calculates

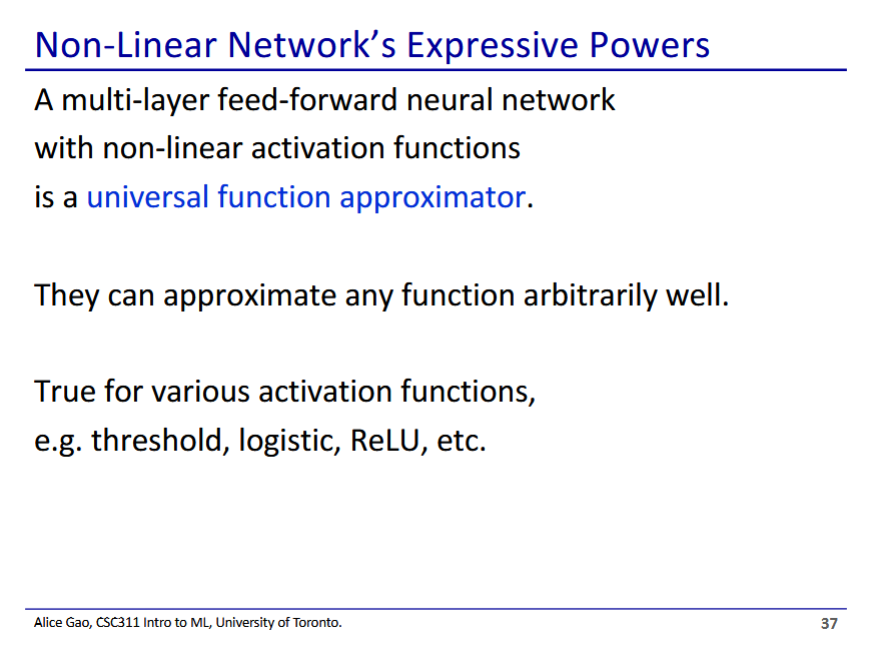
Now we need to design a logistic regression model for each of h1 and h2

Example for h2:

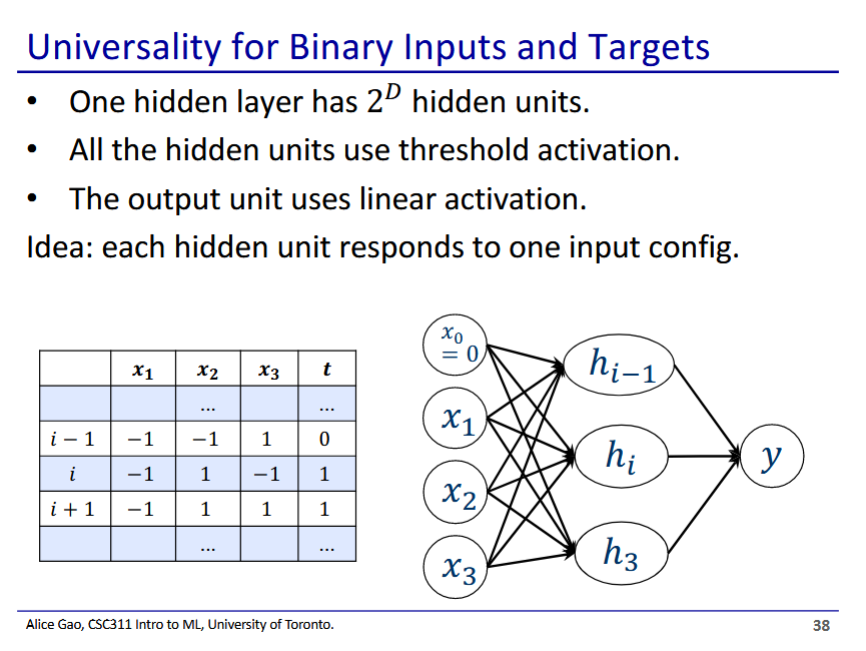
* First we draw out the points on a graph, and then separate using a line (can be any line that separates the items
  + 
* We then figure out the slope and bias that correspond to our line (this is the function for z)
* We then figure out the inequality that will give us the correct classification (threshold activation function for y)
  + We plug in a positive example , and we flip the weights and biases if it's less than 0
    - Thus:
* Weights:
* **On the assignment and test, there will be a question like this where we need to figure out the weights for a neural network**



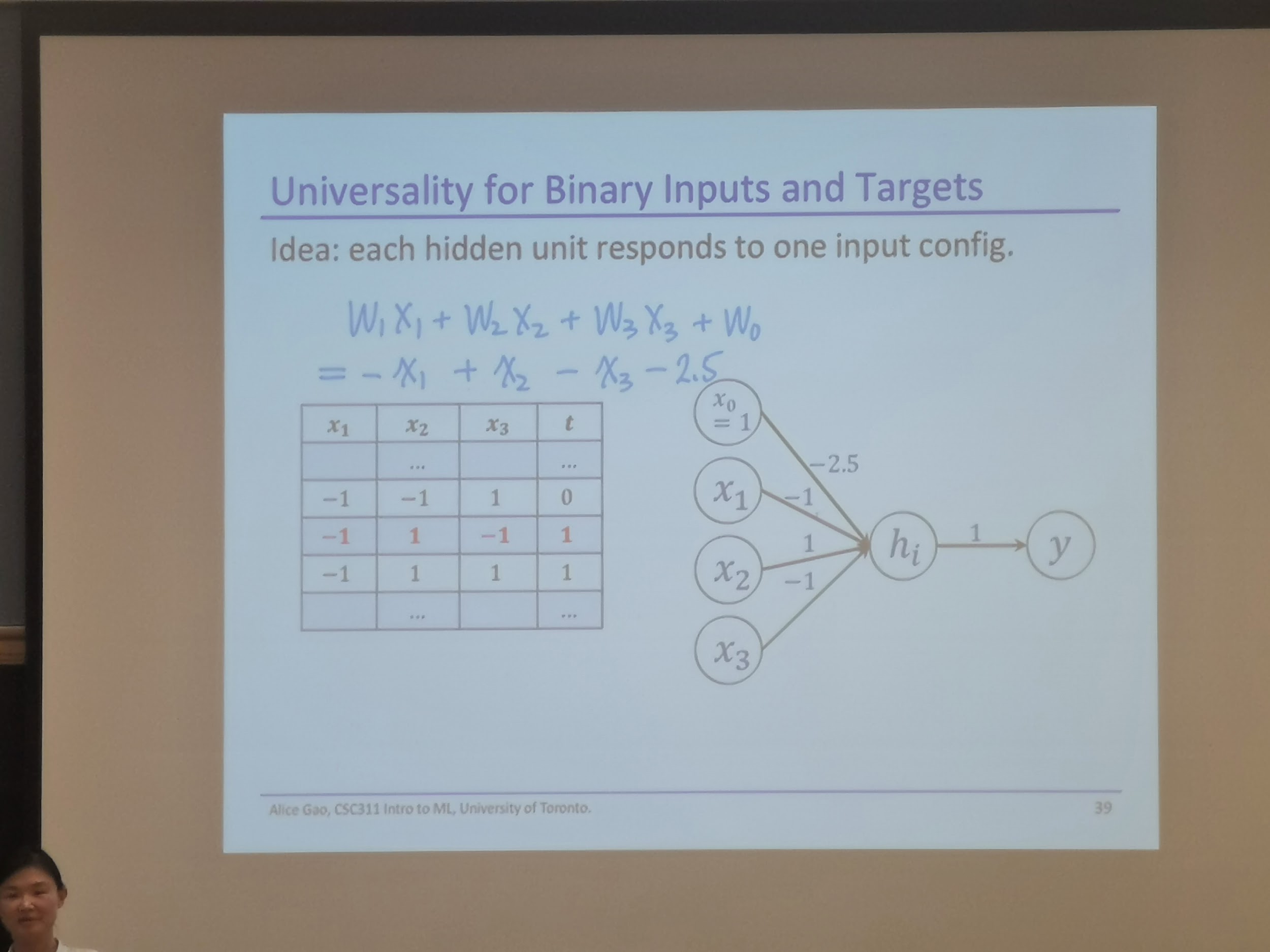
* Assume our activation functions are linear (just returns z), our output would be
  + This is still a linear model, and cannot represent nonlinear functions
  + This we can call a deep linear network
* Thus the power of neural networks comes from the non-linear component (activation functions)



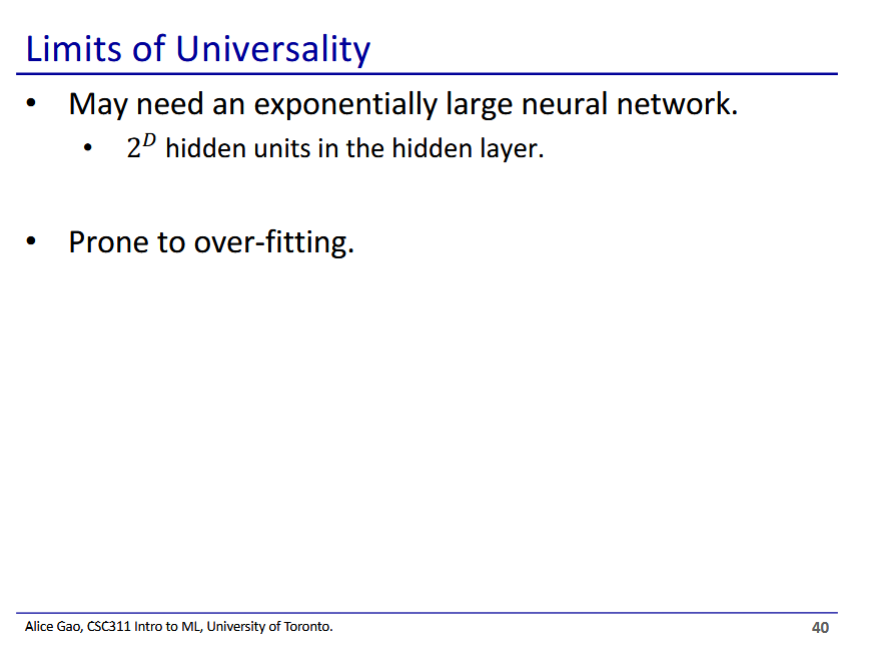
* With a non-linear activation function, a neural network is a universal function classifier
  + Any function can be approximated to any threshold using a neural network

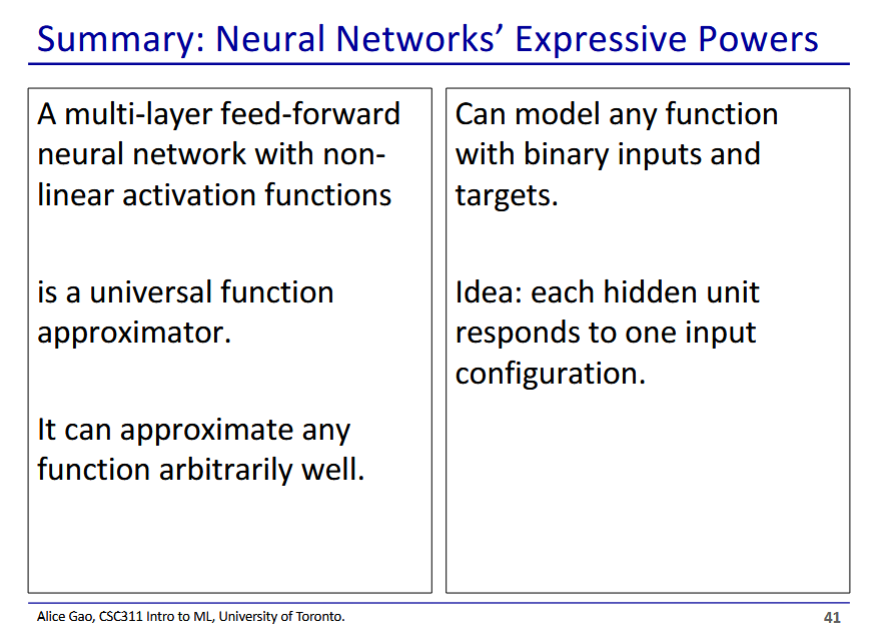


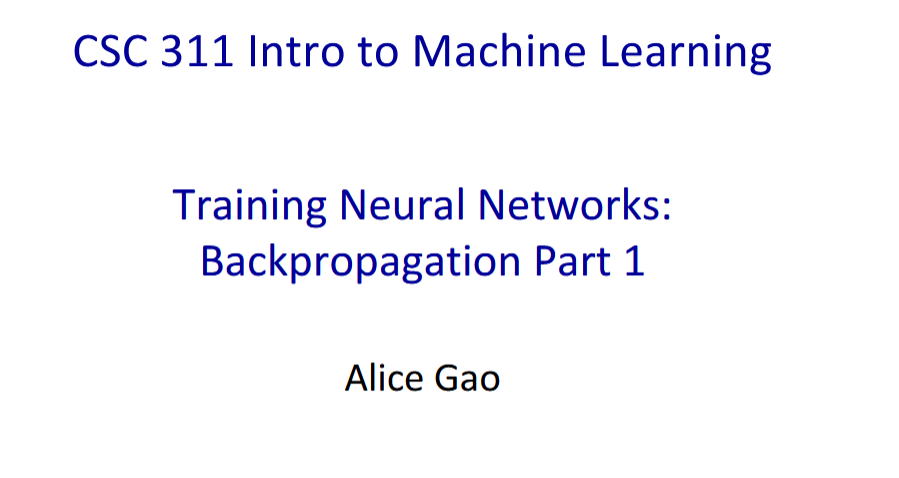
* Thought exercises to understand the universality of neural networks
  + We will not do the general proof, just a small subset where we have binary inputs and binary output
* For all functions with D binary inputs and a binary output, we can have a neural network that represents it perfectly
* We will make a neural network with a hidden layer with D neurons to perfectly model this function

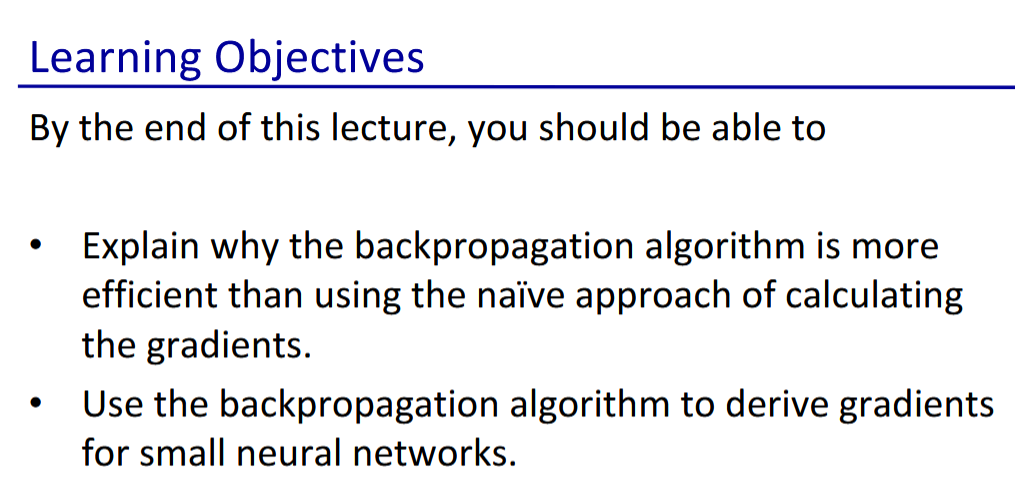


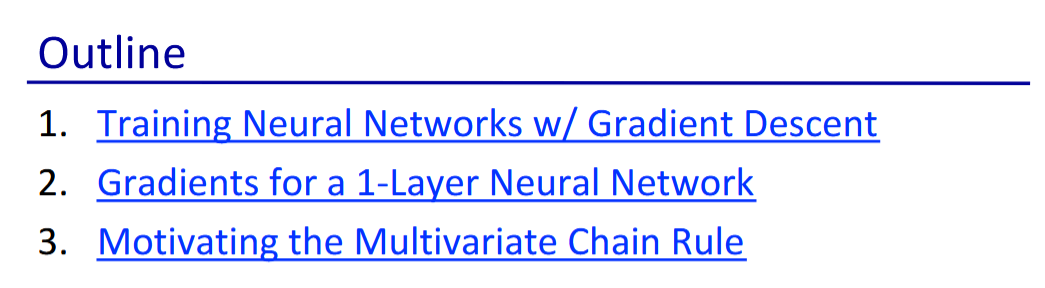
* Hidden neuron hi corresponds to the i’th row of the truth table (is 1 when inputs correspond to that row, is 0 otherwise)
  + We set the weights to the value of the feature in that row (makes it such that the max sum is when input values match the row)
  + We then set bias to 2.5 as the input to this node maxxes at 3
* In this case, hi corresponds to a row where the target output is 1
  + Exercise to decide what to do with a target output is 0

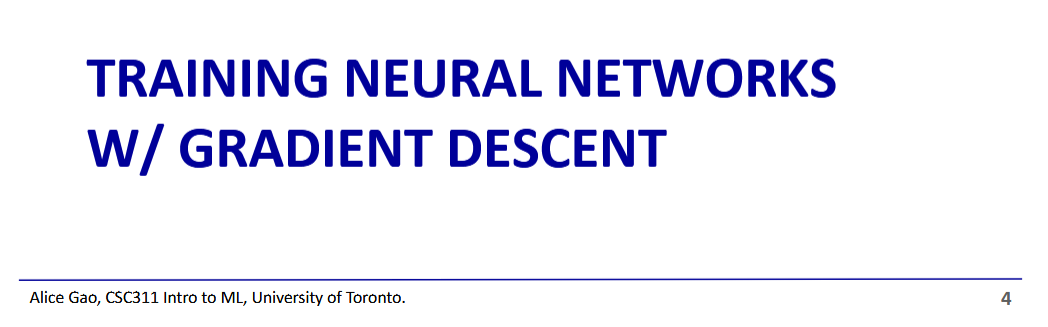


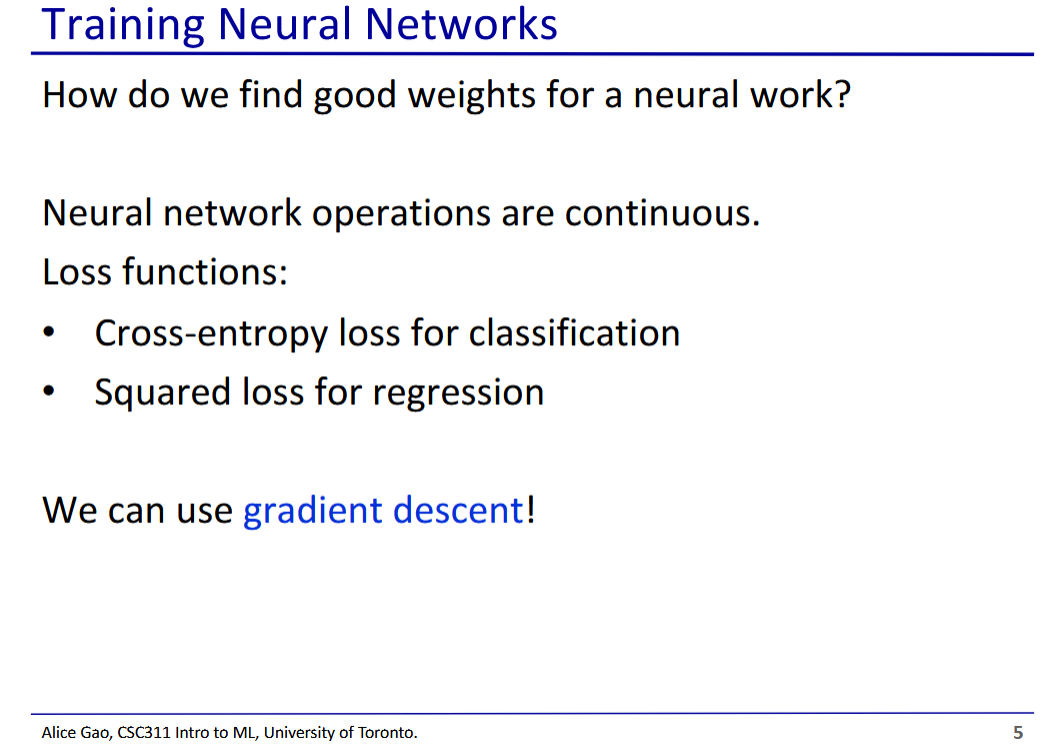


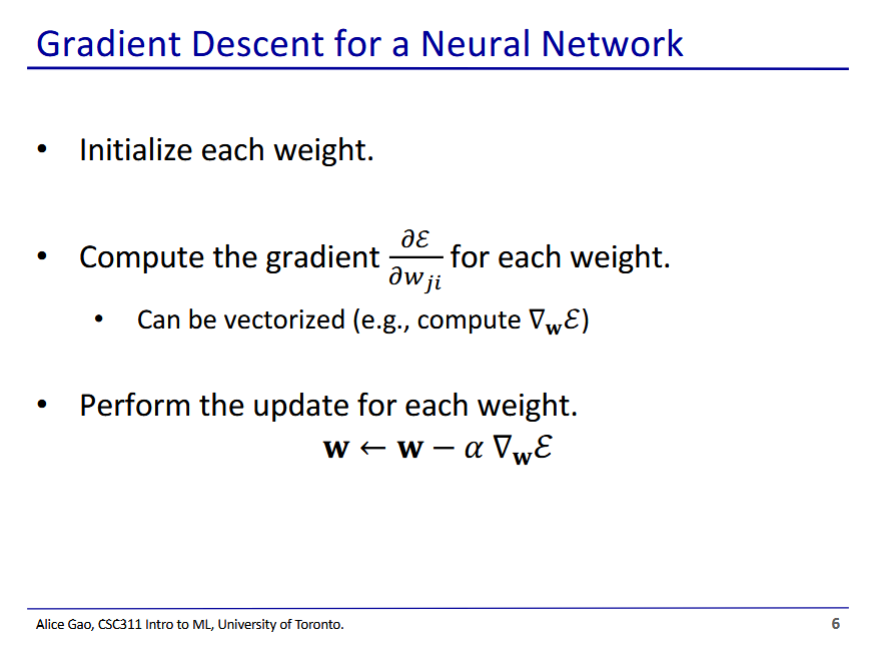


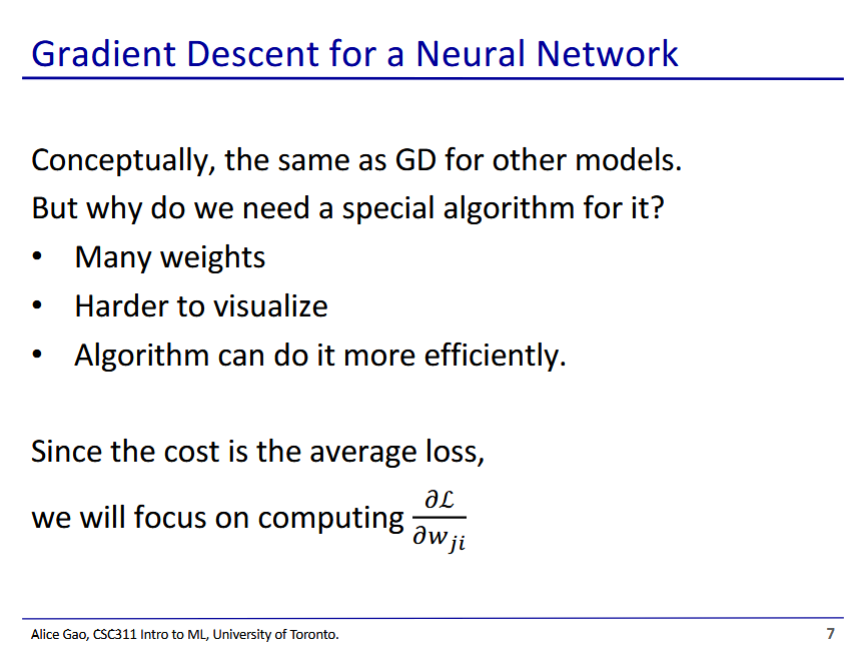




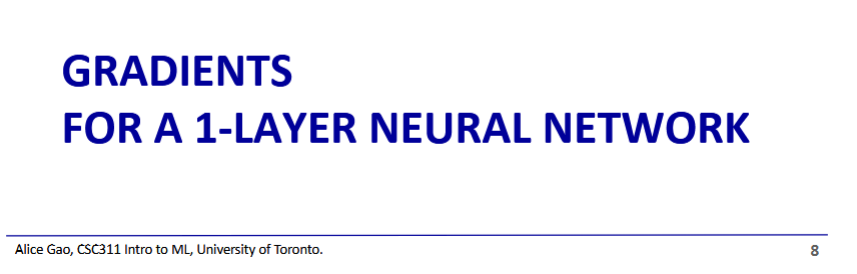


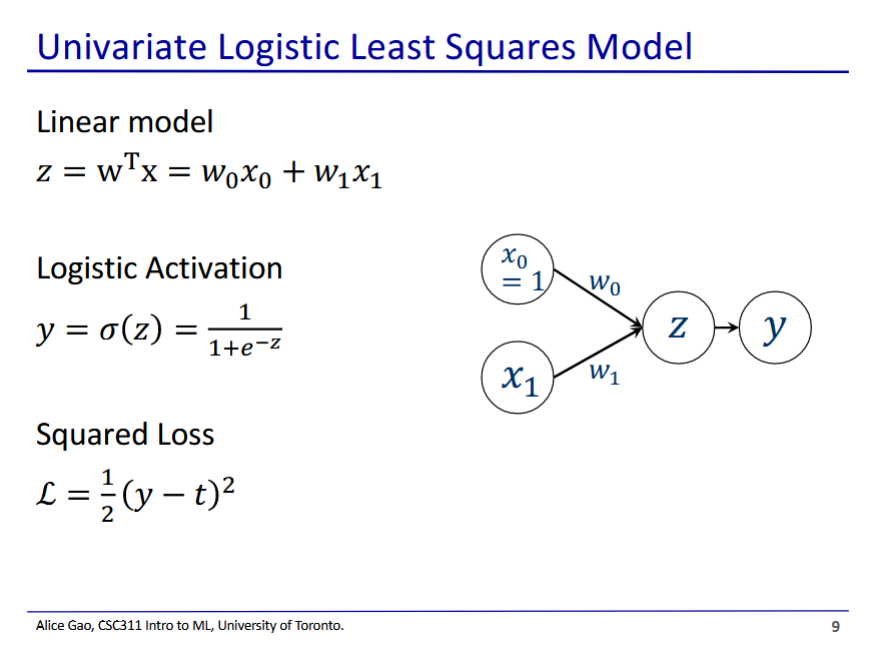




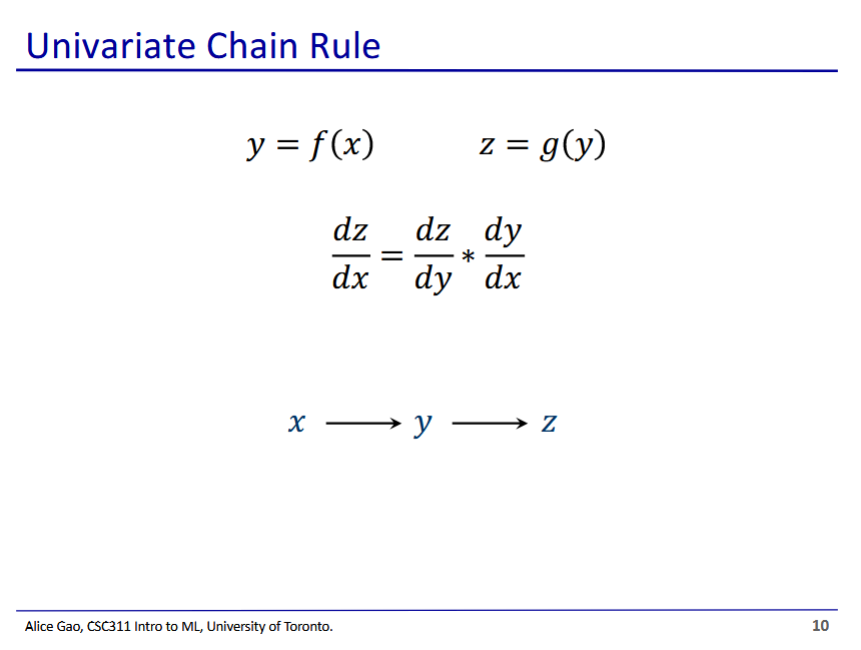


* Gradient descent is hard to do for neural networks since we end up with a ton of weights
  + Each layer needs weights
  + High number of weights results in a high dimension, making things harder to visualise
* Backpropagation can make this more efficient

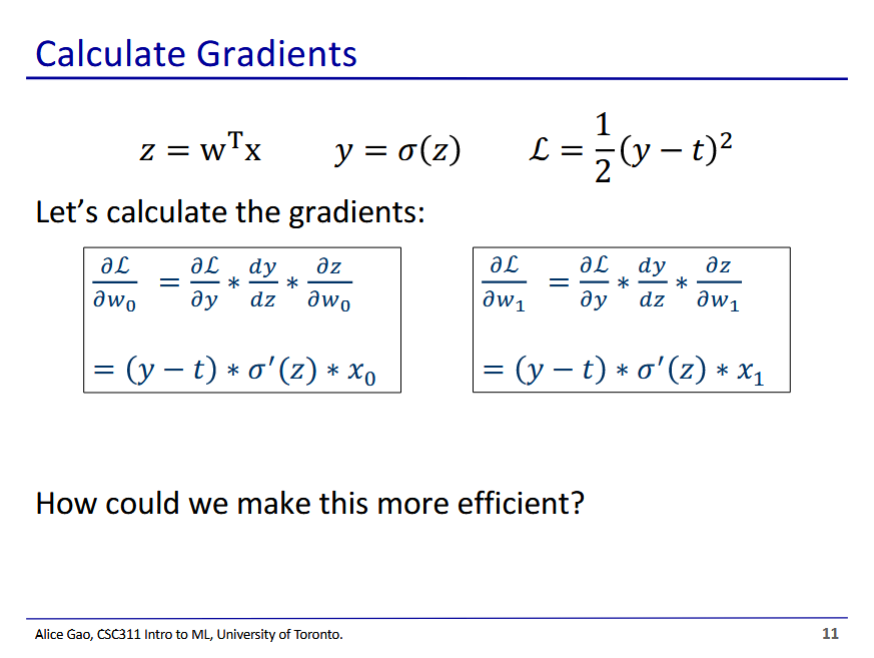




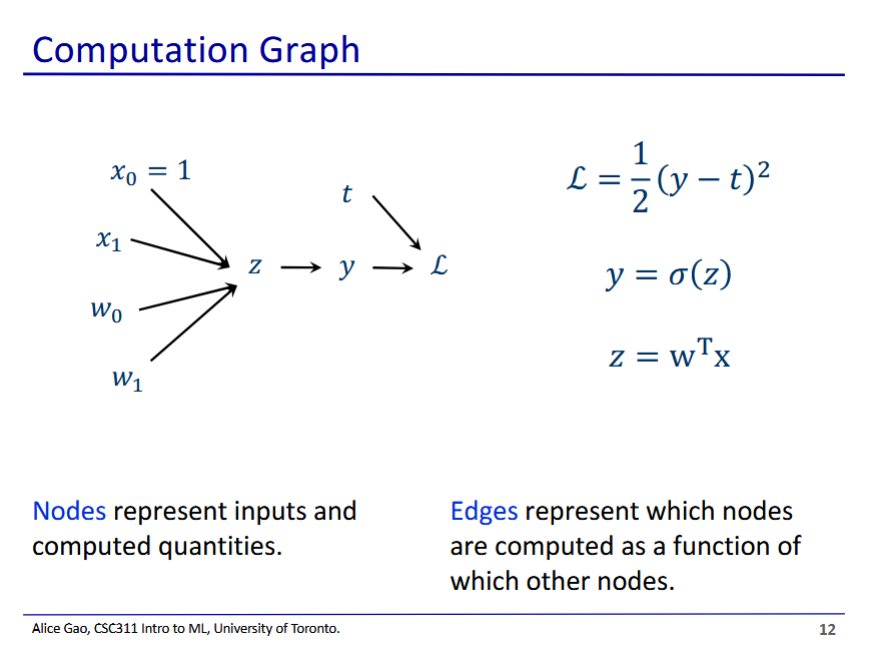
* This is the easiest neural network choice for calculation of derivatives



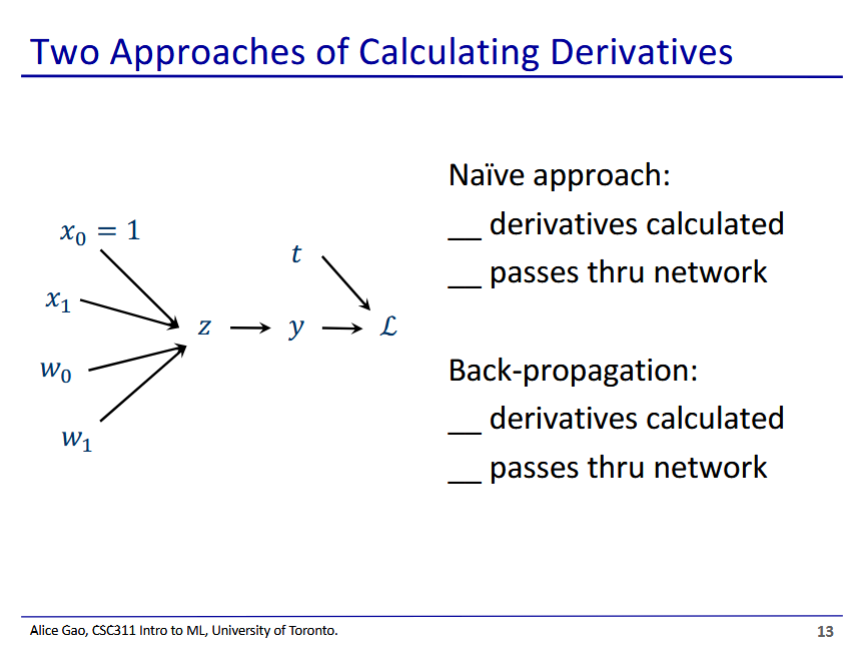
* Univariate chain rule (should be very familiar)
  + Say if y is a function of x, and z is a function of y, we can go backwards from z to y to x when calculating the derivative



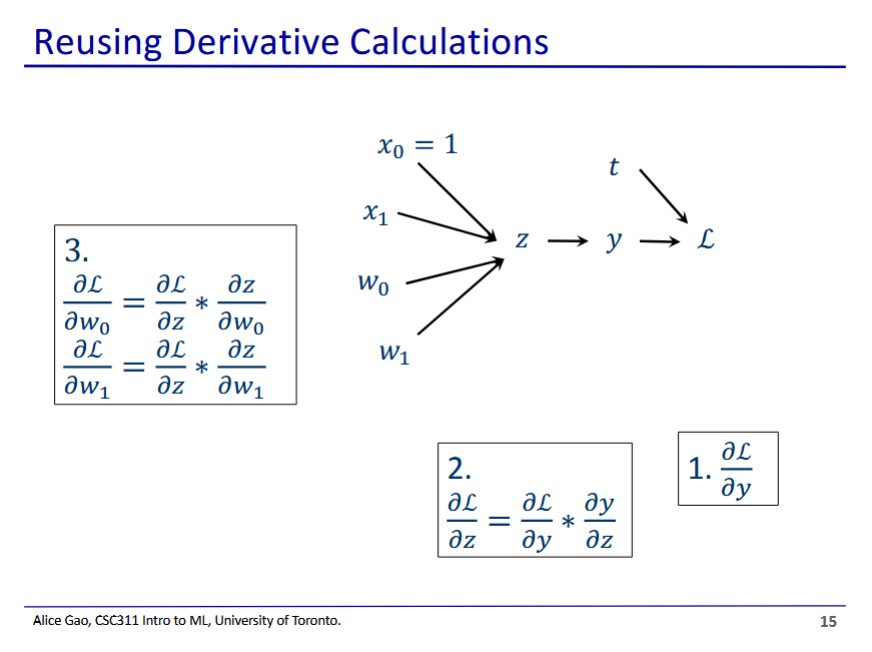
* How can we make this more efficient?
  + We are repeating derivative calculations multiple times ( and )
  + We could try to avoid making these repeated calculations



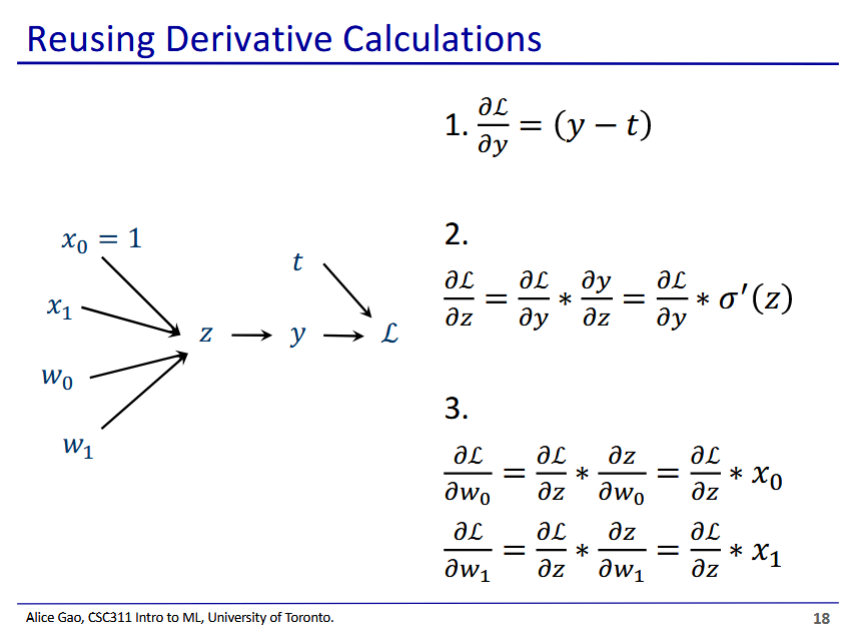
* This looks kind of like a neural network, but has some differences
  + Weights and features are all fed into z (since all 4 things are used to calculate z)
  + We have the loss, which takes the target and output as input



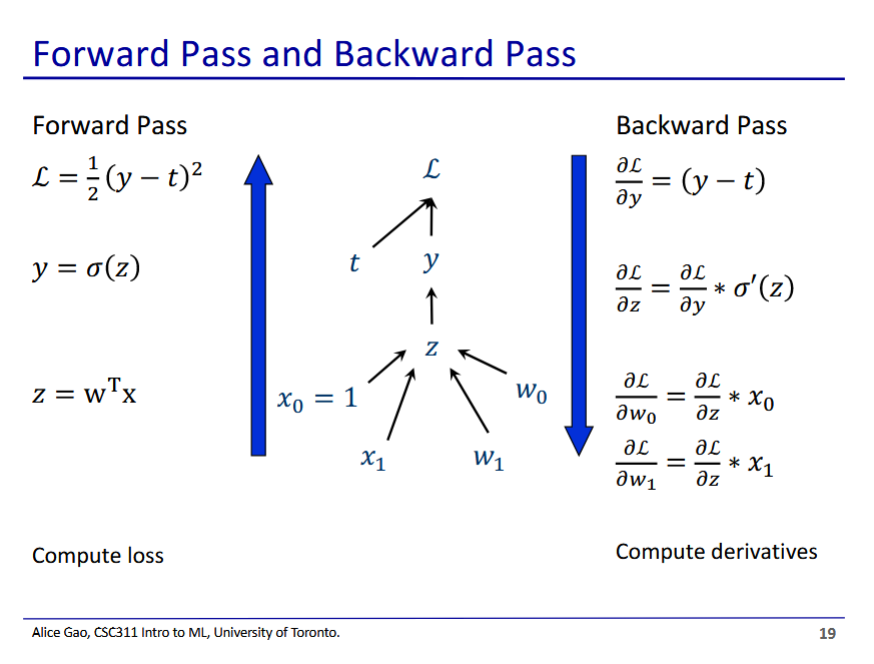
* Naive approach
  + 6 derivatives calculated ()
  + Going through the network twice (once for each weight)
* Backpropagation (see next slide)
  + 4 derivatives calculated
  + Goes through the network once



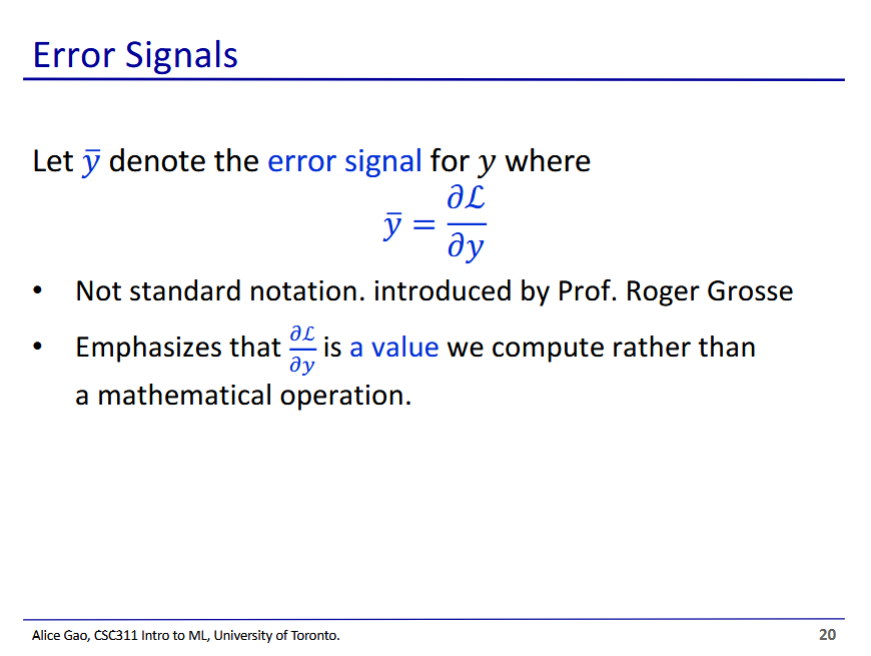
* We go backwards through the graph, reusing the previous steps
* We first calculate , then
  + These are common to both paths
* We then finally calculate and
* We only calculated 4 derivatives and passed through the network once
  + Saves us time (imagine if we had 1000 weights)



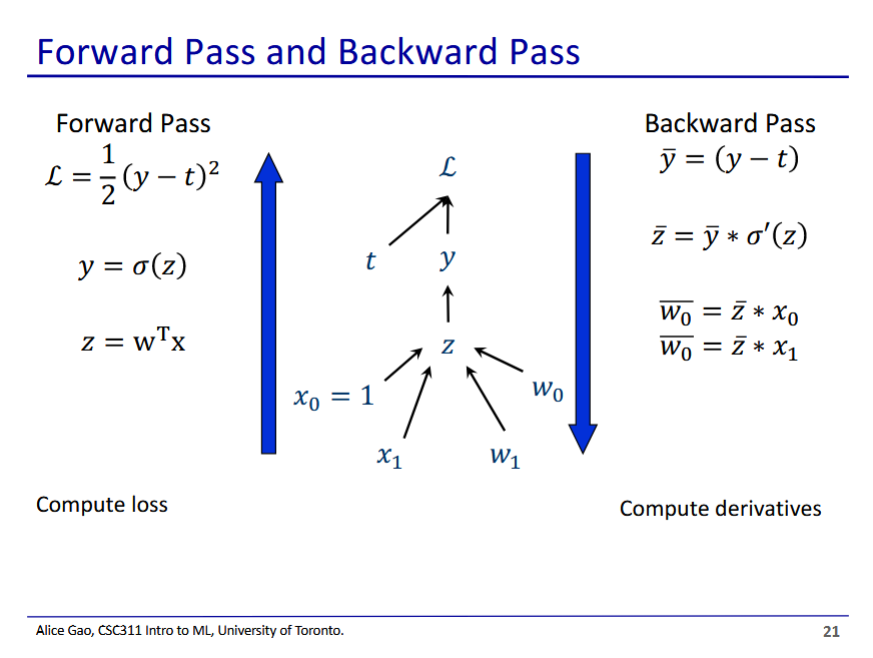
* What if we want to calculate numbers?
  + We need to know
  + comes from the data, but the others (intermediate values) need to be first calculated



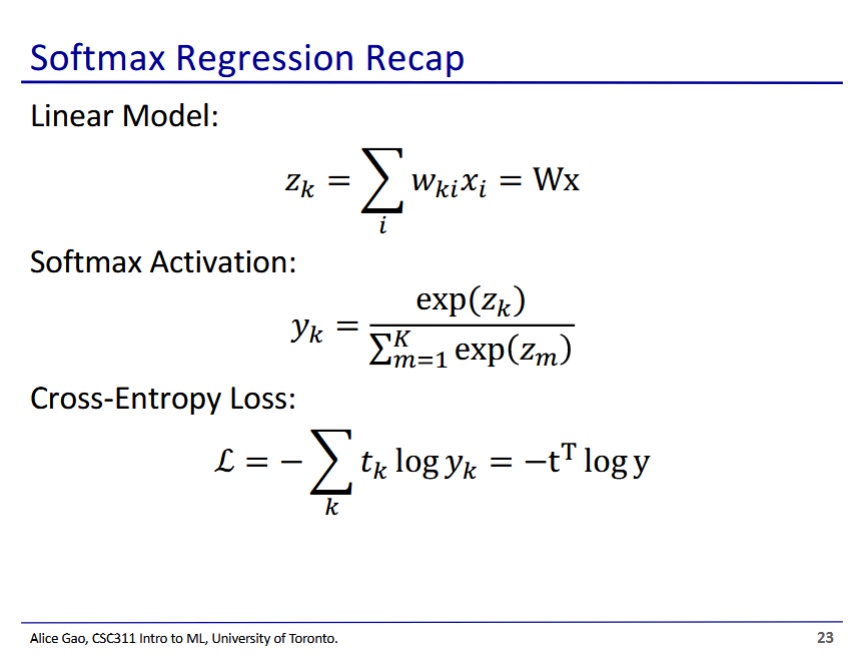
* We first need to pass through the network forwards, calculating the loss and filling in the intermediate values
* We can then go through backwards and calculate the derivatives

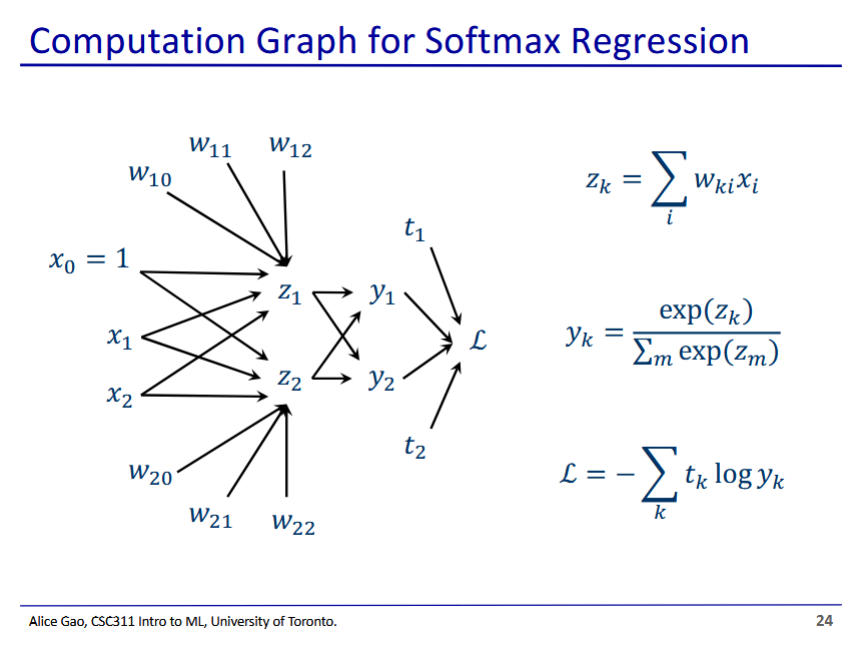


* We will use this notation in homework, etc.

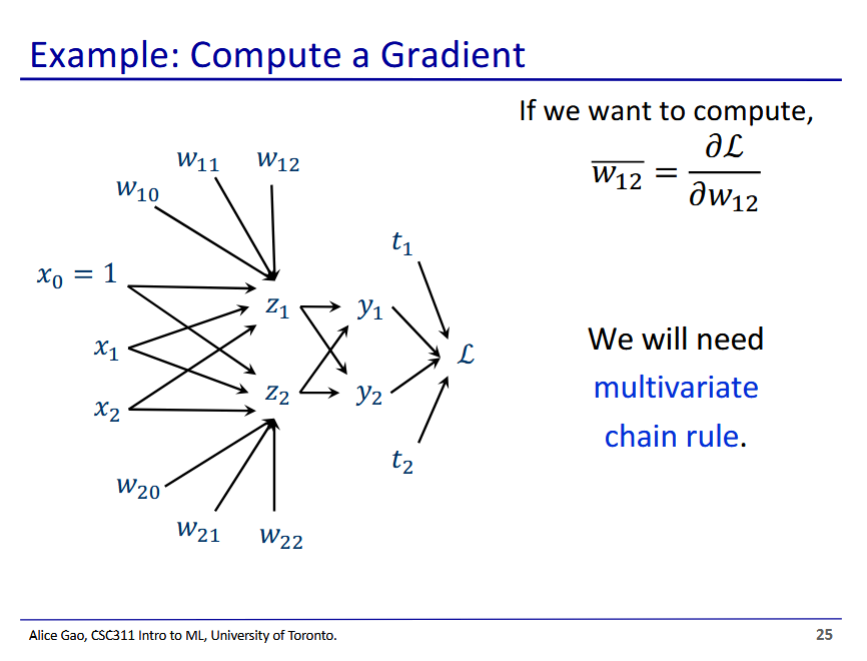


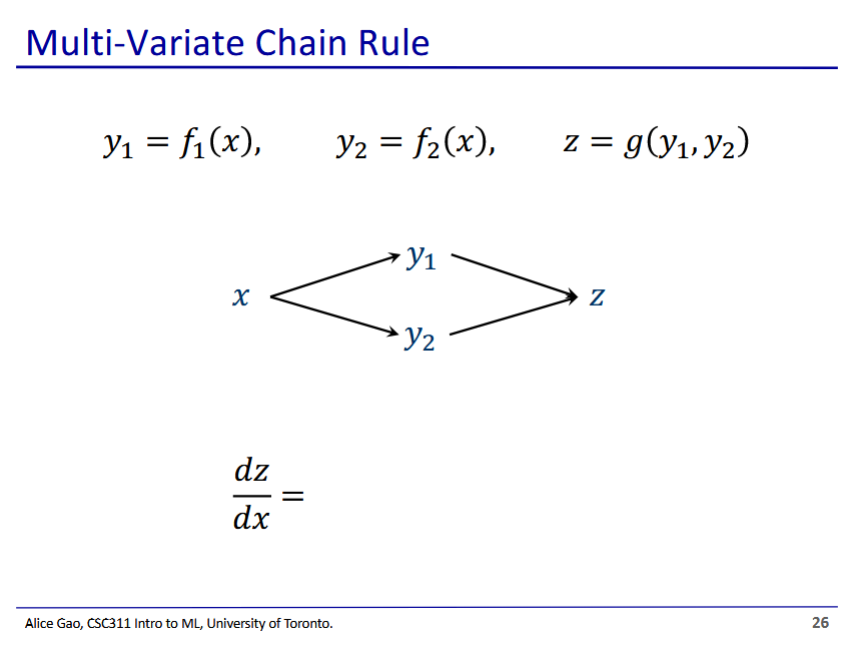






* Calculating the derivative for softmax is very hard
  + We have multiple paths to go from L to w1,2





* We add together the chain rule for both paths