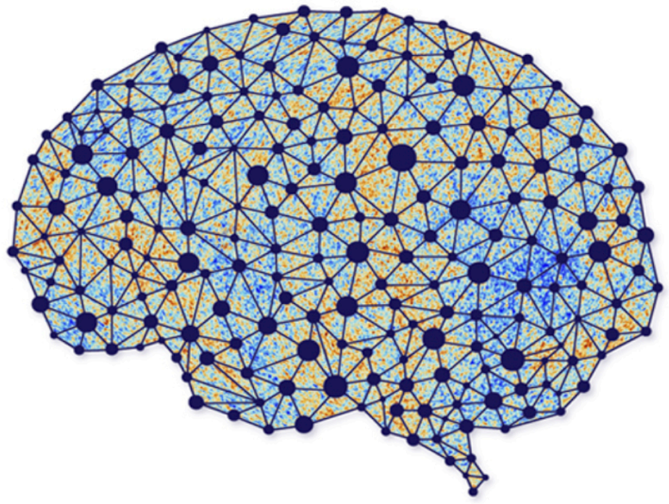
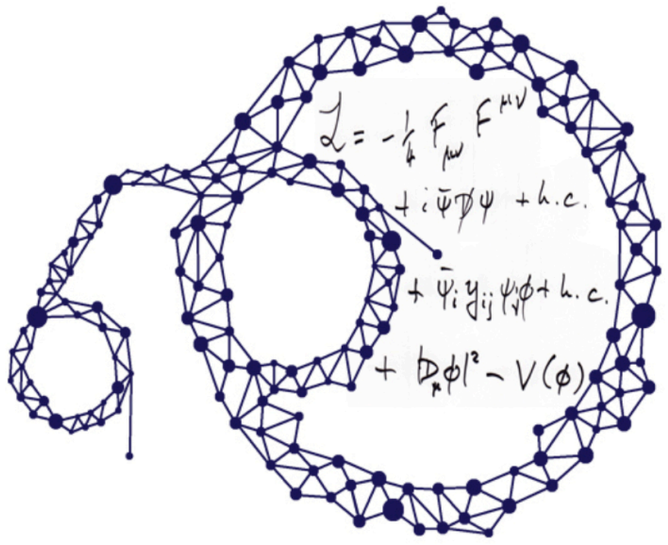
Physics 361 - Machine Learning in Physics

Lecture 5 – Basics of Machine Learning

Feb. 3rd 2025



AI
∩
Universe



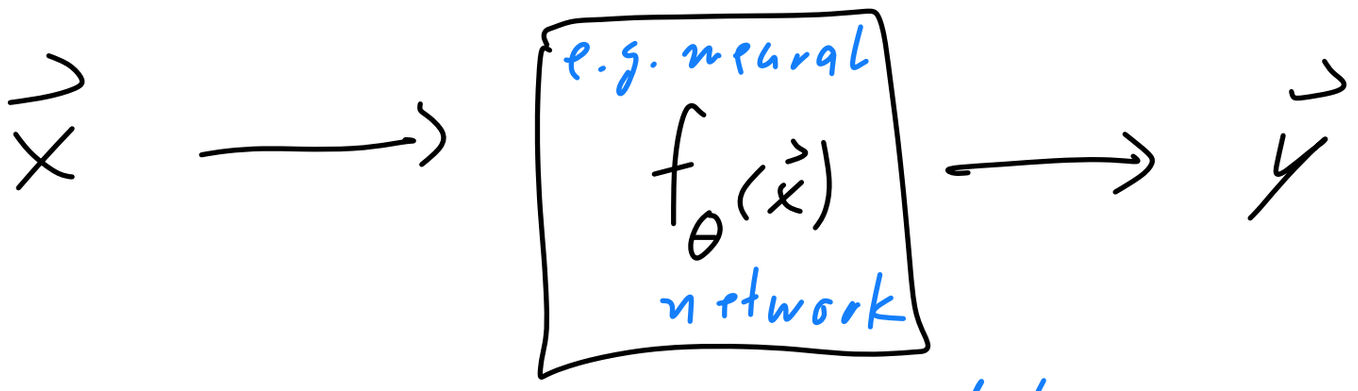
Moritz Münchmeyer

Unit 2: Machine Learning Basics

2.2 Neural Network Basics (cont.)

Supervised machine Learning

We want to Learn a complicated non-linear function to map input $\vec{x}$ to output $\vec{y}$.



θ: model parameters = weights

We want to find the model parameters by minimizing some Loss function on our training data.

E.g. $L^{MSE}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f_\theta(\vec{x}_i))^2$

A very simple neural network

- We need some way to specify the function $f_\theta(\vec{x})$.
  It turns out that a very simple „architecture"
  works very well in practice:

  Linear transformations combined
  with element-wise non-linearities

- A linear transformation / linear layer* is
  given by $\vec{y} = \vec{f}(\vec{x}) = W\vec{x} + \vec{b}$

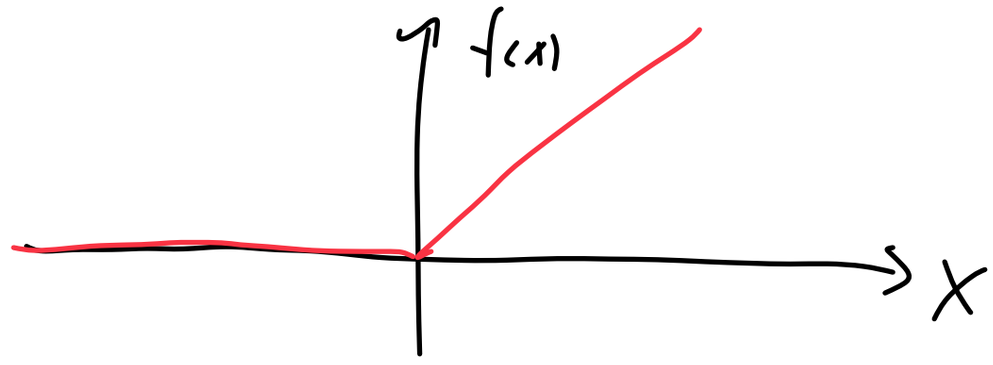
  $\vec{x}$: N_{dim}
  $\vec{y}$: M_{dim}

  W: matrix of weights
  $\vec{b}$: bias

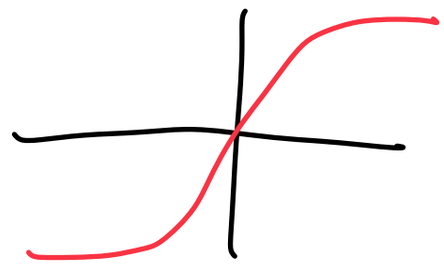
- The most common non-linearity / activation function
  is the ReLU „rectified linear unit"!

  * fully connected linear layer

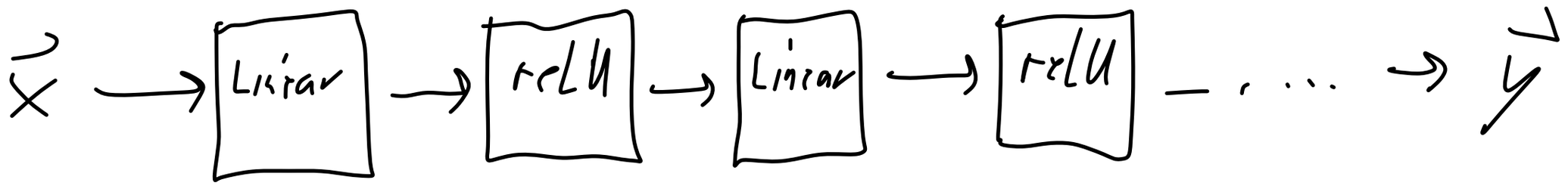
- ReLU is a 1-dimensional function



Piecewise linear function.
Alternatives: tanh, sigmoid



- A multilayer perceptron (MLP) is a "deep neural network" made out of a stack of the 2 building blocks,



The combination of ReLU and Linear Layer can be written as

$$f^i(x) = \max(0, W^i_j x^j + b^i)$$

ReLU

component notation

next Layer uses $f_{(x)}$ as x input.

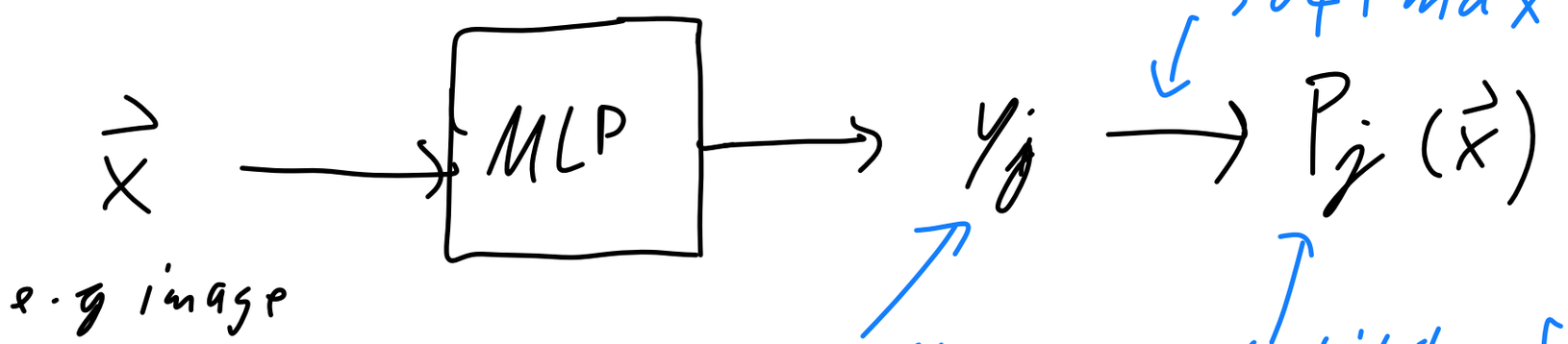
If we stack these layers:

$$\max\left(0, W^i_{n,j} \max\left(0, W^j_{n-1,k} \max\left(0, \cdots\right) + b^j_{n-1}\right) + b^i_n\right)$$

Now we have a function that we can „fit" to the training data.

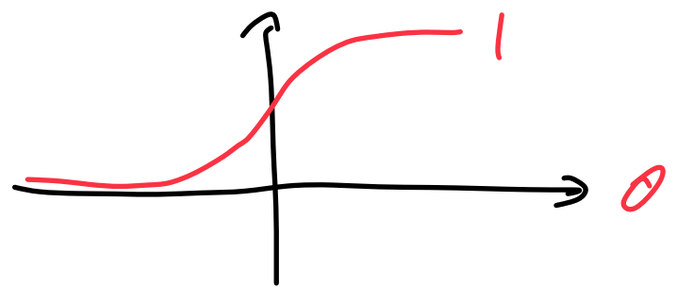
Loss for classification

For classification we want the NN output to be the probabilities of the various classes.



A probability distribution need to have only positive probabilities and they must sum to 1.
To achieve this we need the softmax function

$$P_j(y_j) = \frac{e^{y_i}}{\sum_i e^{y_i}}$$



The typical loss function for classification is the negative log likelihood = cross-entropy.

predicted by NN

$$L(\theta) = -\sum_{i=1}^{N_{examples}} \sum_{j=1}^{N_{classes}} y_j^i \log P_j(\vec{x}_i)$$

θ: parameters of NN: W, b

y^i: one-hot vector which is 1 for the right class.

$$y^i = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$$

Optimization of the parameters

- The last element we need is the „optimization procedure" to find good weights θ^* from minimizing the loss function

- Almost all algorithms used in practice to do this are versions of stochastic gradient descent.

$$\theta_{n+1} = \theta_n - \Sigma \nabla_\theta L(\theta_n; x)$$

new parameters at step n+1 — current parameters — learning rate — gradient of the Loss function

- stochastic means we evaluate the gradient at each step using only a subset of the training data, called the „mini batch"

Algorithmic differentiation

For gradient descent we need the gradient.
Two approaches that generally don't work well:

- symbolic differentiation.
  gives you a (very long) symbolic expression. Not very scalable.
- Finite differences

$$f'(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

  With finite numerical precision this is very noisy, especially when f is very large or very small.
- Instead: Use *Autodifferentiation*

Auto-differentiation

- Split function $X \to Y$ into elementary building blocks. „Computation graph"
- Differentiate the building blocks analytically.
- Evaluate the building block derivatives numerically.
- Propagate through the function using the chain rule („backpropagation algorithm")

$$f(g_i(\theta_j)) \longrightarrow \nabla_{\theta_j} f = \sum_i \frac{\partial f}{\partial g_i} \times \frac{\partial g_i}{\partial \theta_j}$$

This is what frameworks like PyTorch implement. We'll skip the details for now since we don't immediately need them.

Unit 2: Machine Learning Basics

2.4 Training our first model

Train an MLP on SUSY data

- We will use the simulated collider data. **The goal will be to discriminate SUSY (supersymmetry) events from non-SUSY events.** The data is from the paper https://www.nature.com/articles/ncomms5308 **.**

- We use a modified version of the code from https://physics.bu.edu/~pankajm/MLnotebooks.html which was written for the review https://arxiv.org/abs/1803.08823 **A high-bias, low-variance introduction to Machine Learning for physicists.**

- We will use **python** with the **pytorch** framework (it does the auto-differentiation). Pytorch is the most widely used tool in ML research (followed probably by JAX now).

- We will use **Google Colab** to run the python code. Colab provides free computational resources, including GPUs, with a nice interface based on python's Jupyter.

Physics Background

- Using the dataset from the UC Irvine ML repository produced by MC simulations to contain events with 2 leptons (electrons or muons)

- These events with 2 leptons with large p_T can occur in SUSY models or within the SM.

- 18 kinematic variables ("features") are recorded for each event.

- We train a MLP classifier to classify the events into SUSY or SM background.

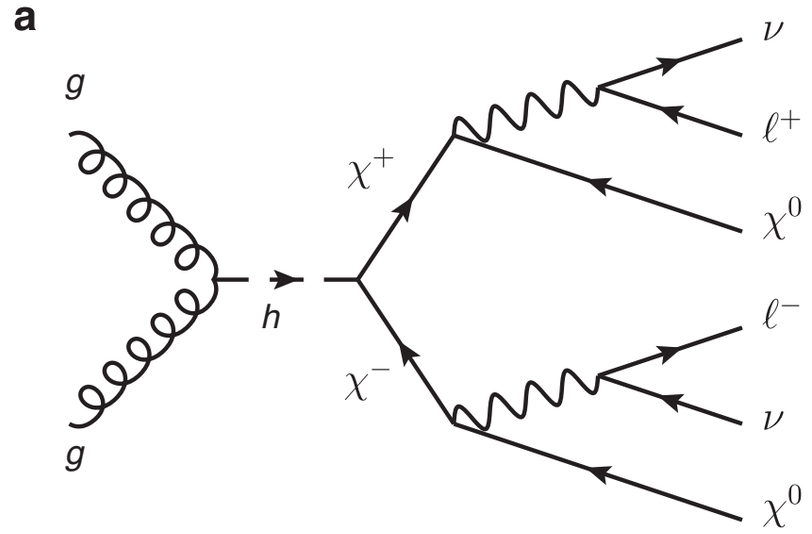
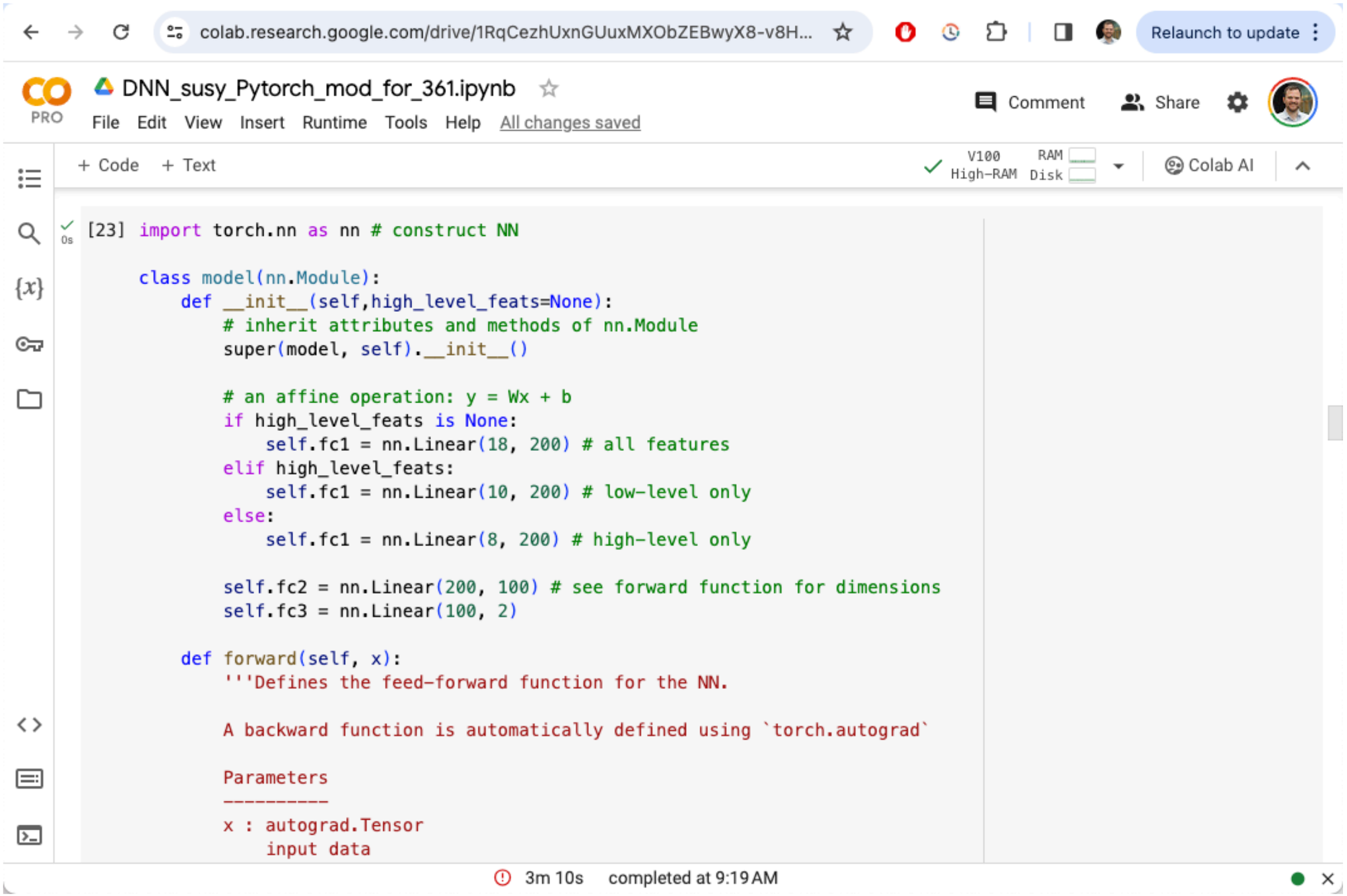


Figure 4 | Diagrams for SUSY benchmark. Example diagrams describing the signal process involving hypothetical supersymmetric particles $\chi^{\pm}$ and χ^0 along with charged leptons $\ell^{\pm}$ and neutrinos ν (**a**) and the background process involving W bosons (**b**). In both cases, the resulting observed particles are two charged leptons, as neutrinos and χ^0 escape undetected.

Baldi et al, Nature Communications, Volume 5, Article number: 4308 (2014)

The rest of this section will be presented in Colab. I will upload the Colab notebook on the course page, and you can download it from there, upload it to Colab, and run it yourself.



Course logistics

- **Reading for this lecture:**
  - **For example:** Deeplearningbook.org chapter 5.
- **Problem set**: Second problem set coming out today or tomorrow.