

# Lab 2b: Pytorch Operators and Optimizers

University of Washington

ECE 596/AMATH 563

Spring 2022



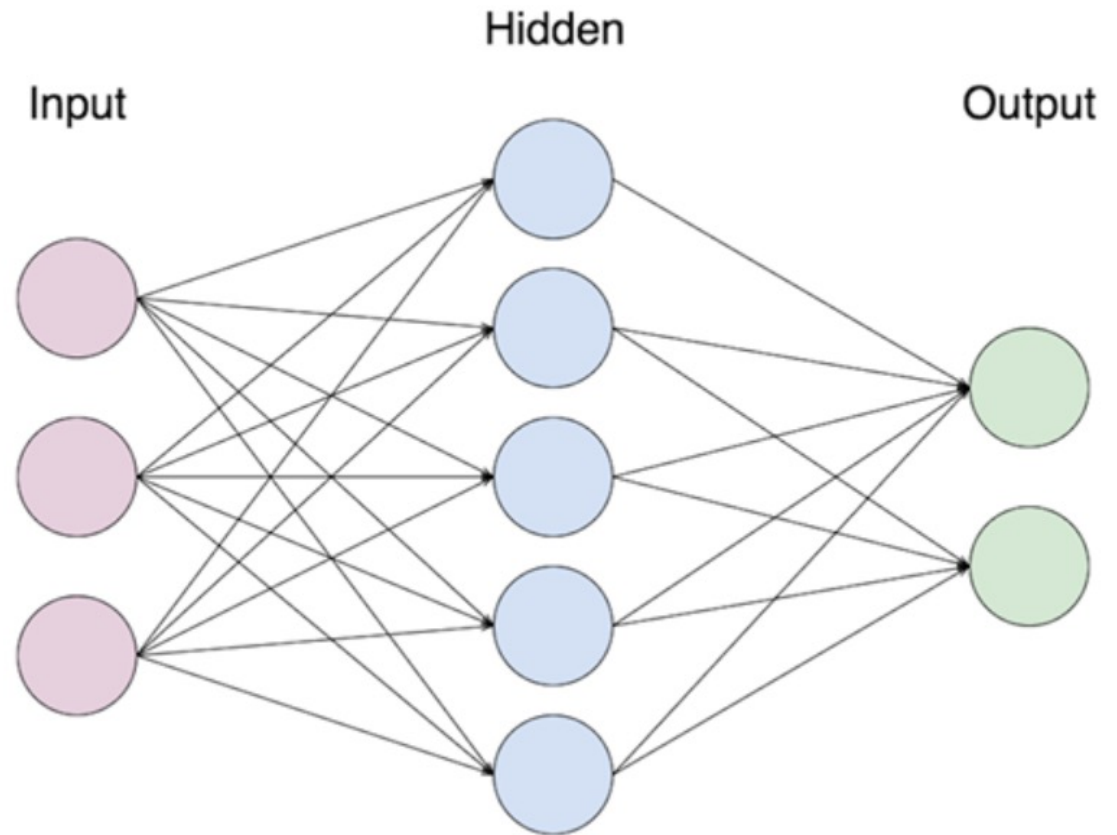
# Outline

- Pytorch operators and layers
  - Activation Functions
  - Normalization
  - Dropout
  - Loss Functions
- Designing Training Procedures: Pytorch Optimizers
- Assignment:

# PyTorch Operators and Layers

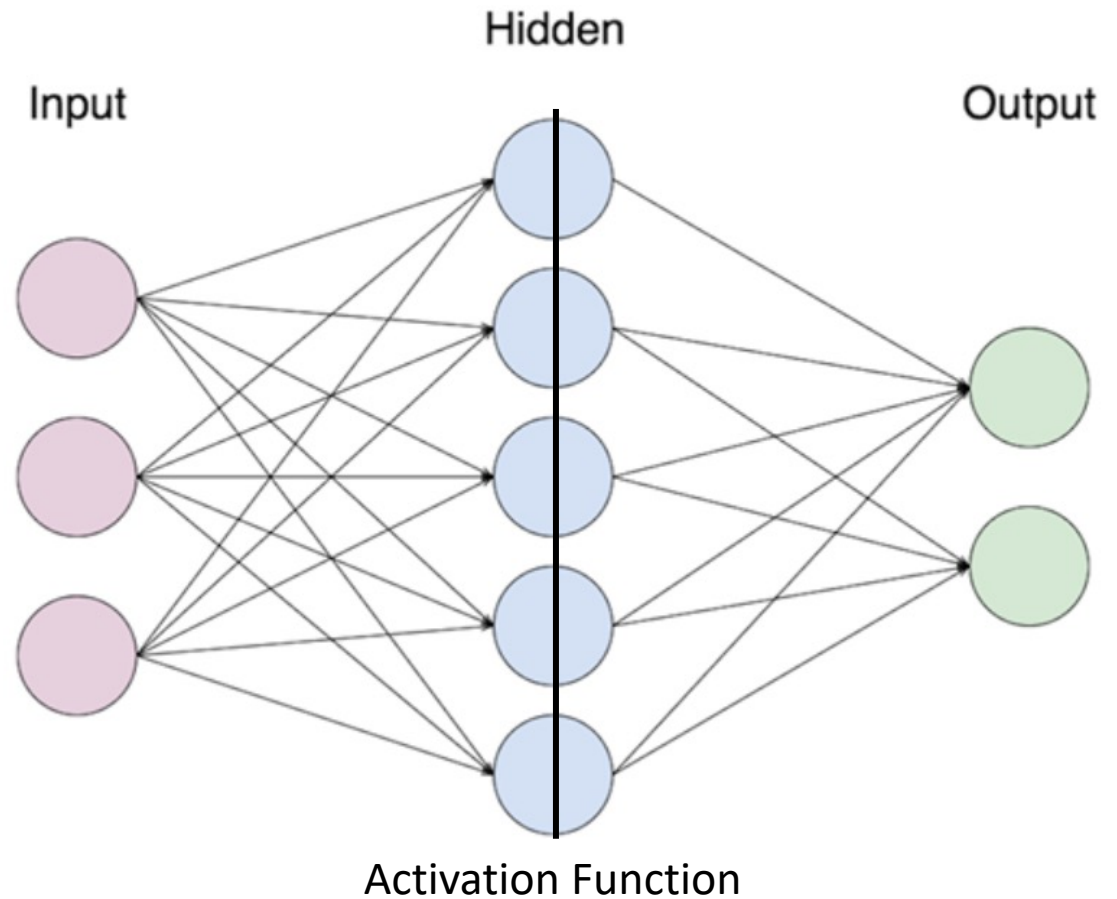
# PyTorch Operators/Layers

- Activation Functions
- Normalization
- Dropout
- Loss Functions



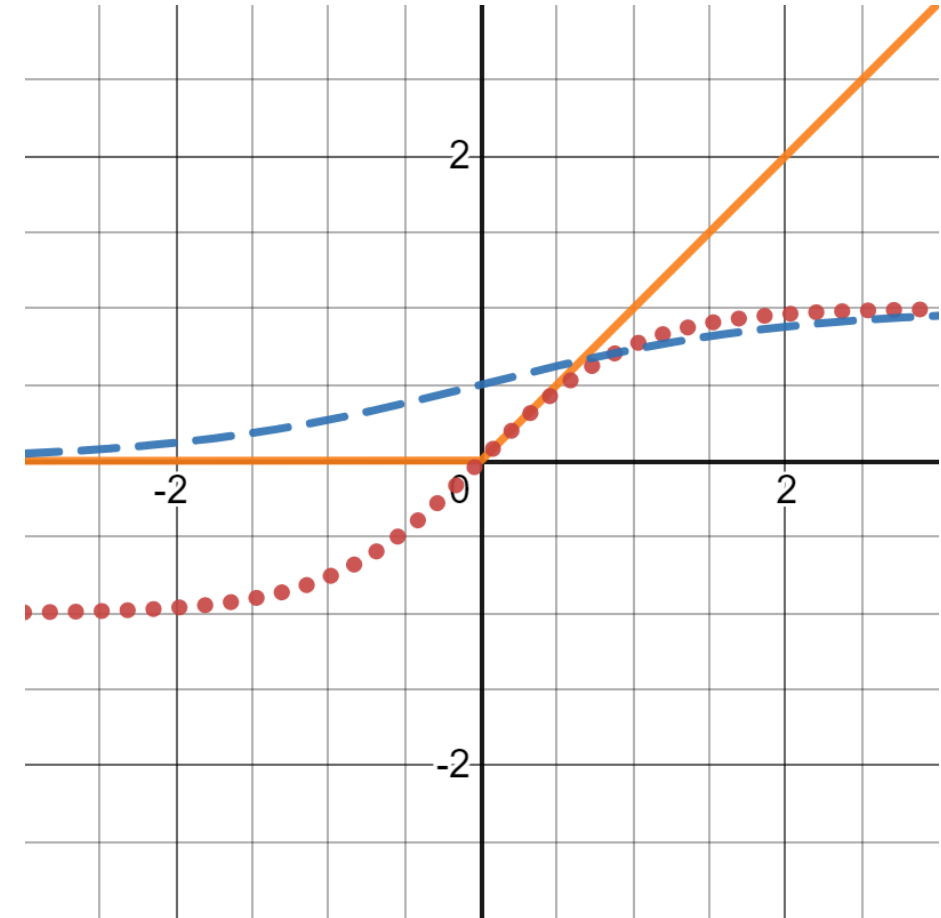
# PyTorch Operators/Layers

- Activation Functions
- Normalization
- Dropout
- Loss Functions



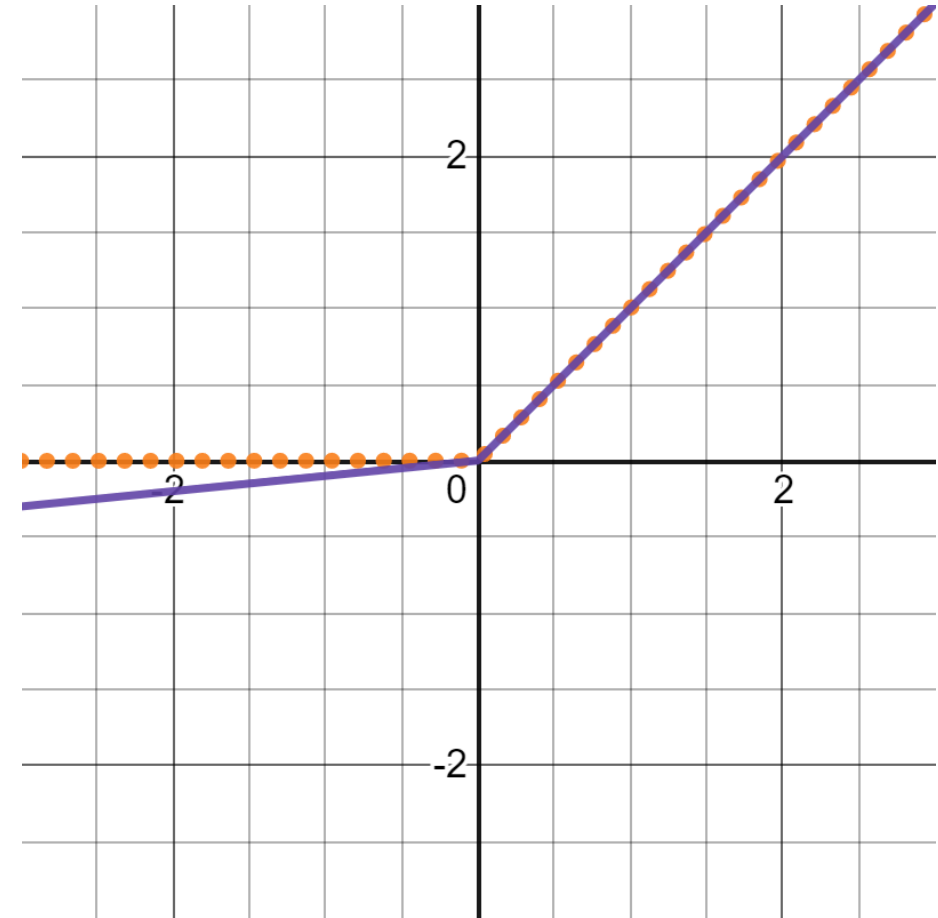
# Activation Functions

- Non-linear functions performed by neurons
- ReLU - Rectified Linear Unit (nn.ReLU)
  - $y \geq 0$
- Tanh (nn.tanh)
  - $-1 < y < 1$
  - nn.Tanh
- Sigmoid (nn.Sigmoid)
  - $0 < y < 1$



# Activation Functions

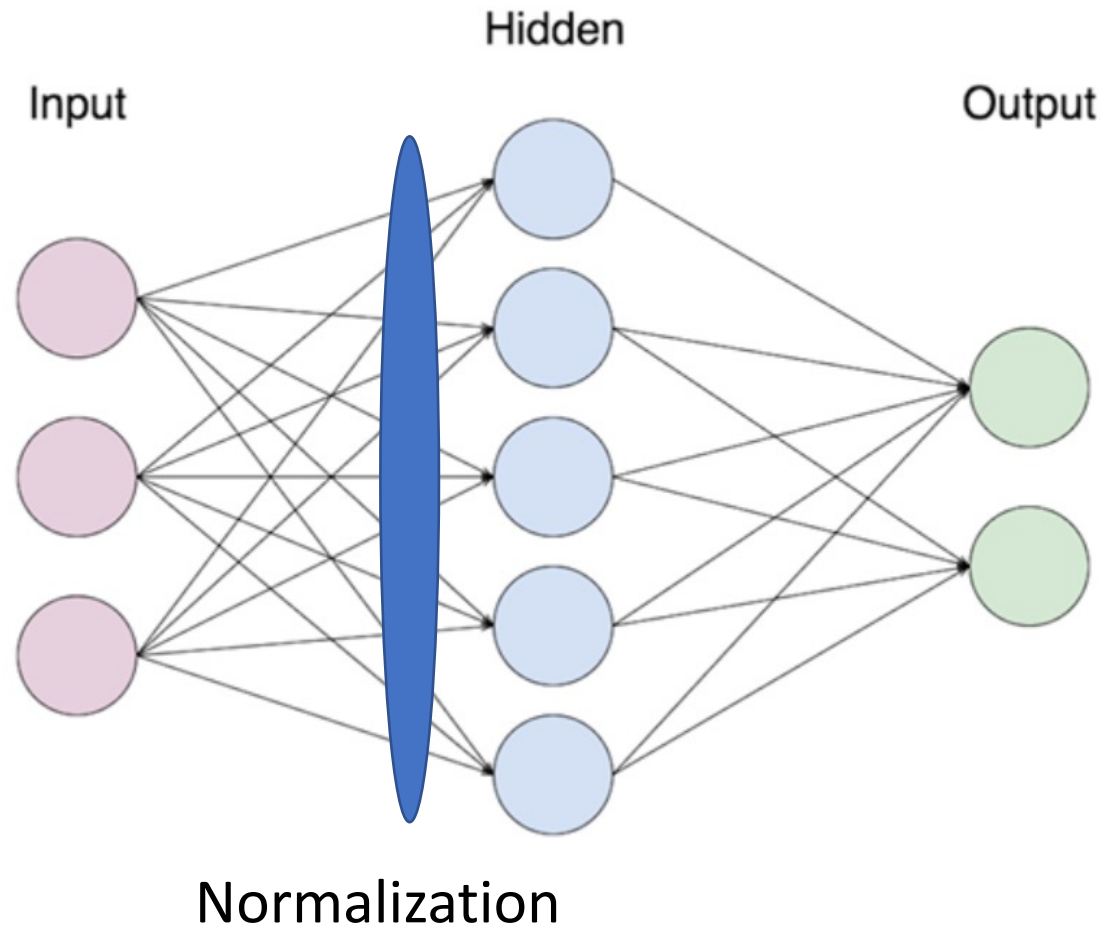
- Leaky ReLU
  - Similar to ReLU, but has non-zero values for negative  $x$
  - Takes argument *negative\_slope*, which determines the slope for  $x < 0$ .
- For full list of activation functions, see: <https://pytorch.org/docs/stable/nn.html>



Leaky ReLU with negative slope = 0.1

# Python Operators/Layers

- Activation Functions
- **Normalization**
- Dropout
- Loss Functions



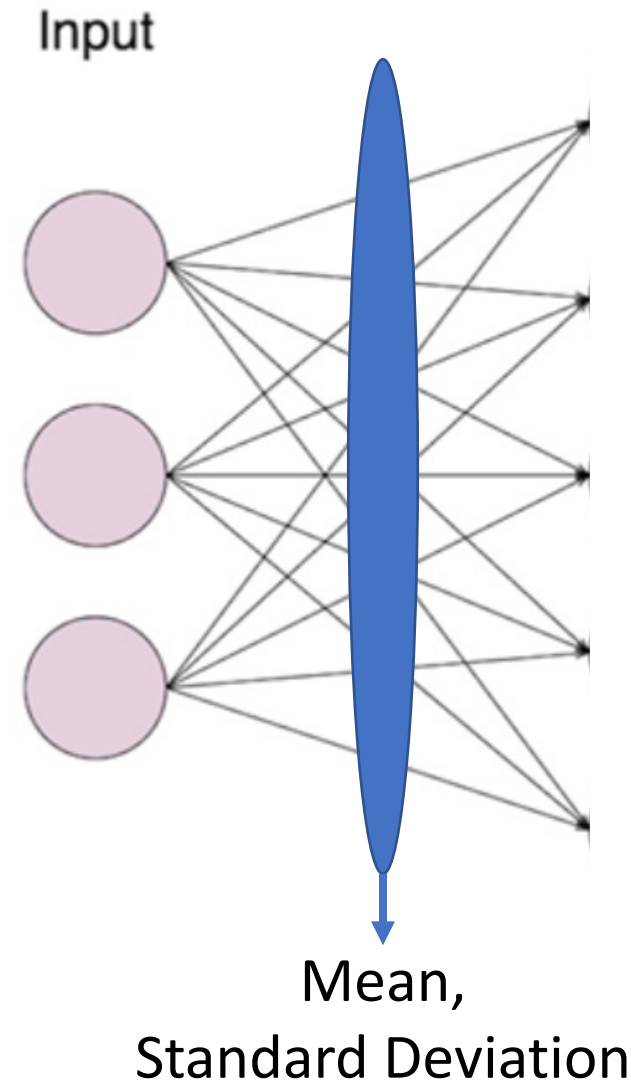


# Input Normalization: Batch Normalization

- Normalizes input into each layer for each training mini-batch
- Addresses issue of shifting input distributions over training
- Inputs:
  - num\_features: Number of features in the input vector
  - eps: numerical stability parameter

## Example:

```
b = torch.nn.BatchNorm1D(100)
input = torch.rand(50, 100)
output = b(input)
```



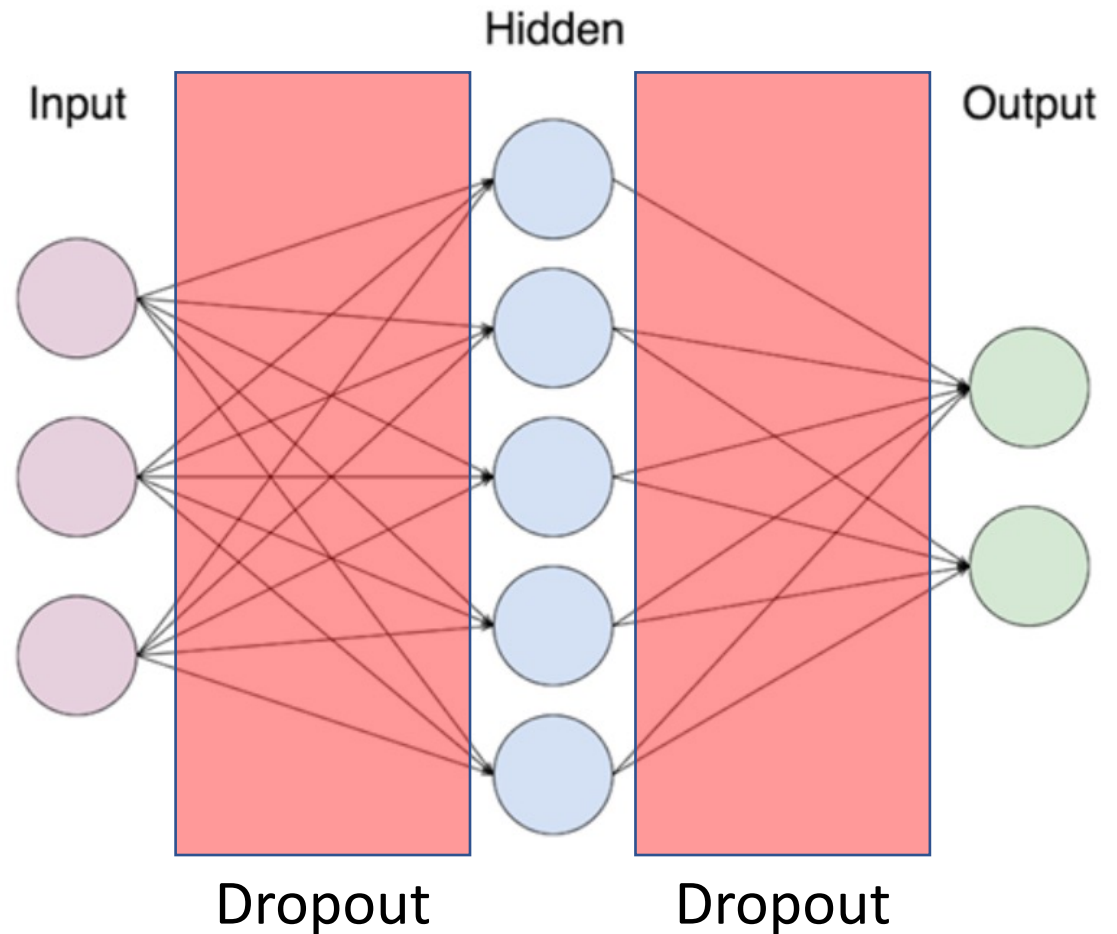
# Input Normalization

Other normalization procedures include:

- Layer Norm: Transposes Batch Norm. Normalizes over all summed inputs to a layer
  - <https://arxiv.org/abs/1607.06450>
- Group Norm: Normalizes by grouped channels instead of batches
  - <https://arxiv.org/abs/1803.08494>

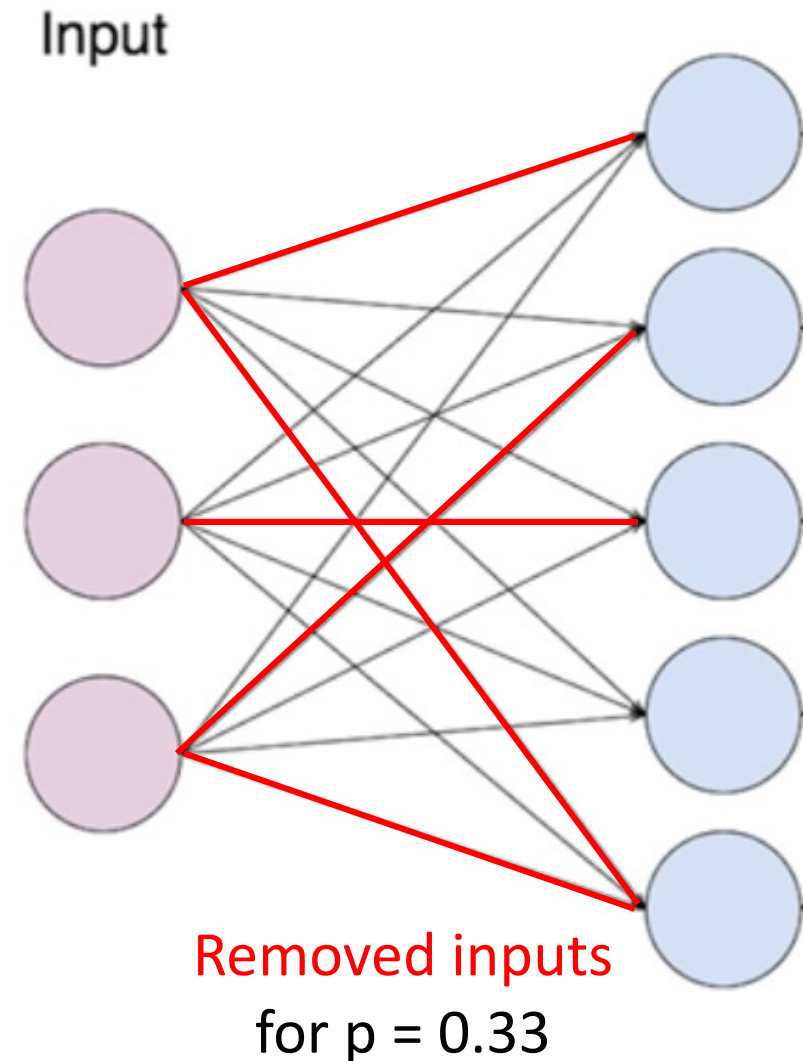
# Python Operators/Layers

- Activation Functions
- Normalization
- **Dropout**
- Loss Functions



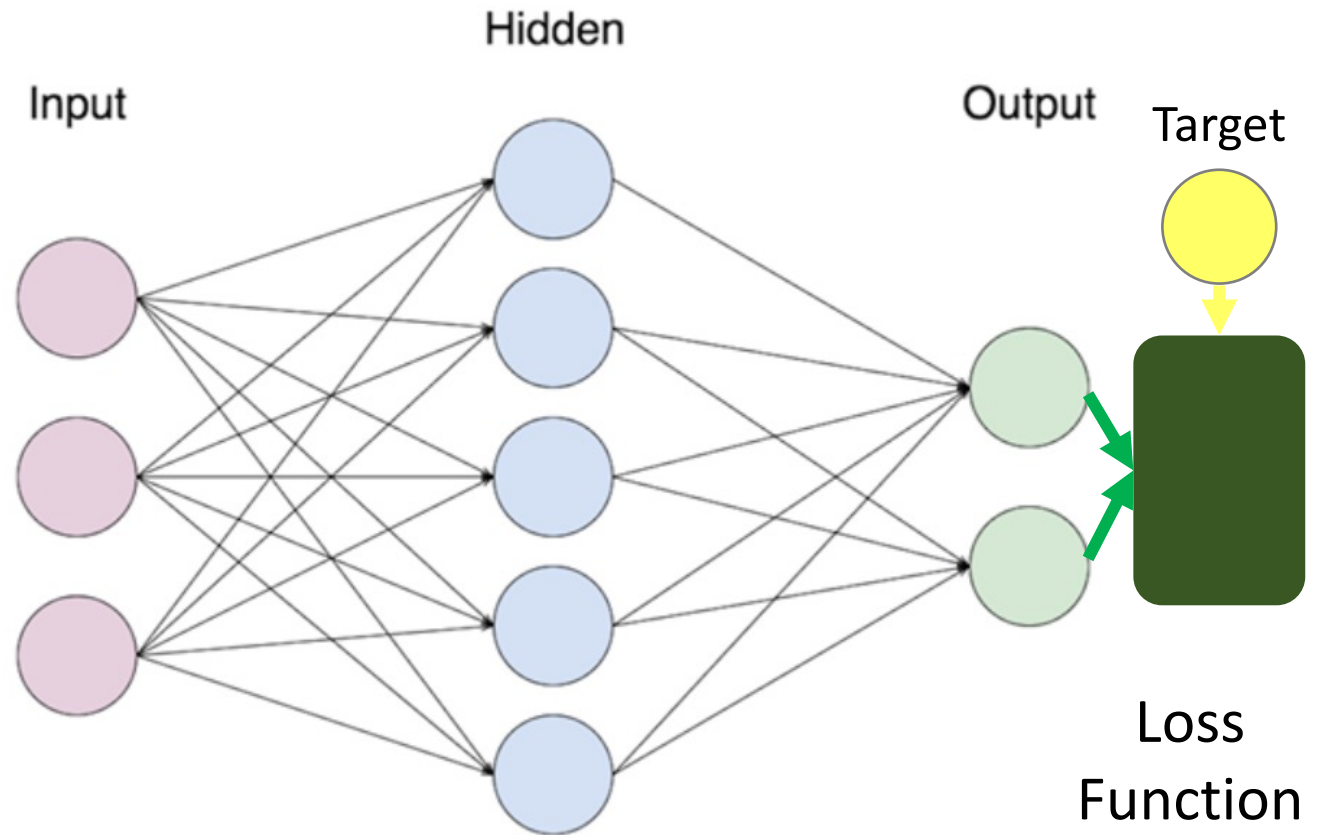
# Dropout

- Randomly zeroes some elements of input tensor with probability  $p$
- Effective technique for regularization
- Outputs scaled by  $1/(1-p)$
- Treated as identity during evaluation



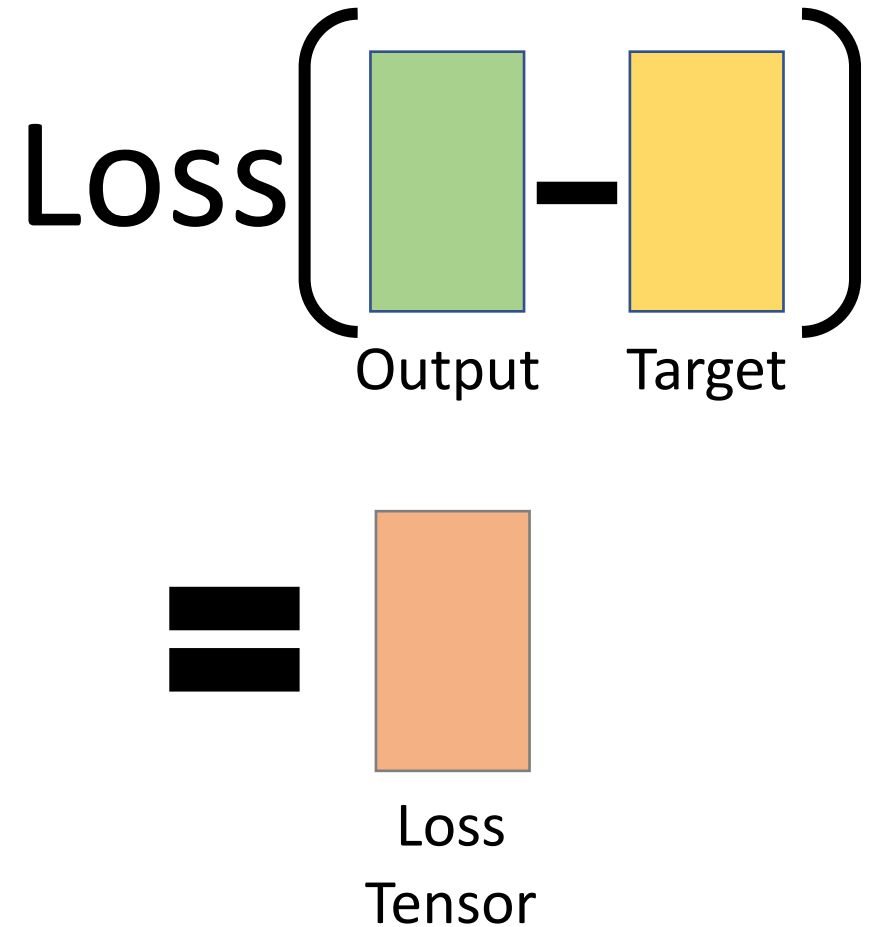
# Python Operators/Layers

- Activation Functions
- Normalization
- Dropout
- Loss Functions



# Loss Functions

- Loss function parameters:
  - Reduction: how the output will be reduced in dimension:
    - None: Gives entire Loss Tensor with no reduction over batches
    - Sum: Takes sum of the loss tensor across batches, returning a single number
    - Mean: Same as sum, but divides by the number of batches to get the mean



# Loss Functions – Cross Entropy Loss

- Cross Entropy Parameters

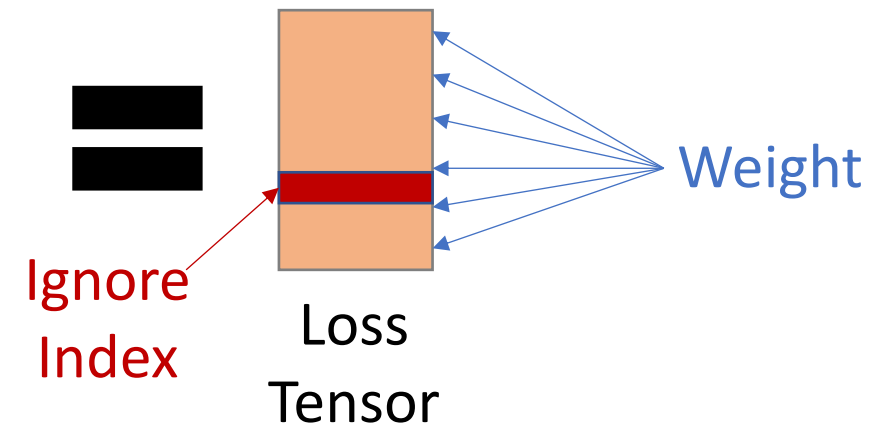
- Weight

- 1D tensor assigning weights to each class, which is helpful if you have an unbalanced training set

- ignore\_index

- Specifies a target value that is ignored and does not contribute to the input gradient

$$\text{Loss} \left( \begin{array}{c} \text{Output} \\ \text{Target} \end{array} \right)$$

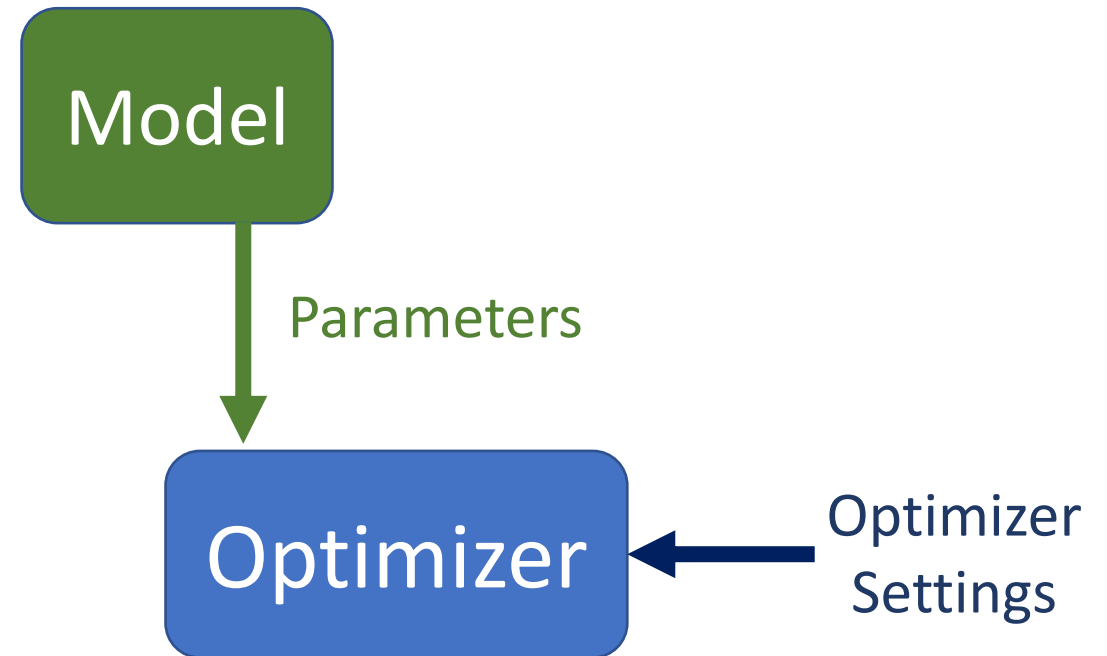


# Designing Training Procedures



# Optimizer Initialization

- Parameters:
  - Should be iterable containing parameters to optimize
  - E.g., `model.parameters()` or `[var1, var2]`
  - Parameters must be defined BEFORE the optimizer
- Optimizer Settings
  - Learning rate, weight decay, etc.

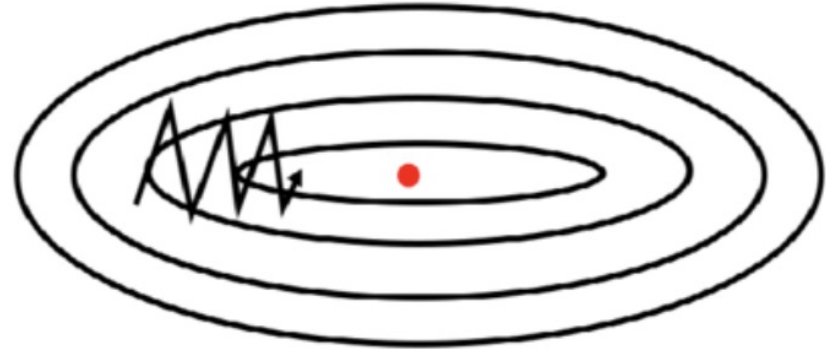


# Optimizers

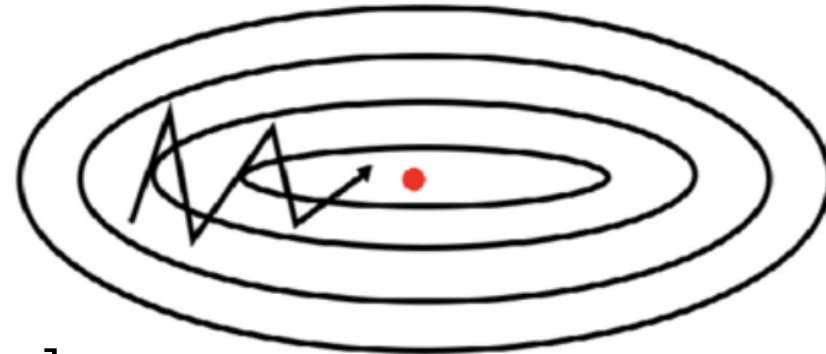
- Stochastic Gradient Descent (`torch.optim.SGD`)
  - `params`: Model parameters
  - `lr`: Learning rate (required)
  - `momentum`: momentum factor (default: 0)
  - `weight_decay`: (default: 0)

**Example:** `torch.optim.SGD(model.parameters(), lr = 0.001, momentum = 0.2, weight_decay = 0.1)`

SGD without momentum



SGD with momentum



# Optimizers

- Adam (`torch.optim.Adam`)
  - params: Model parameters
  - lr: Learning rate (default: 0.001)
  - betas: coefficients (tuple) used for computing running averages of gradient and its square (default: (0.9, 0.999))
  - eps: term added to denominator to improve numerical stability (default: 1e-8)
  - weight\_decay: (default: 0)

## Example:

```
torch.optim.Adam(model.parameters(),  
lr = 0.01, betas = (0.95, 0.998),  
eps = 1e-7)
```

# Other Common Optimizers

- AdaDelta (`torch.optim.Adadelta`)
  - Precursor to Adam which uses first-order estimates to adapt learning rate
- Adamax (`torch.optim.Adamax`)
  - Variant on Adam based on infinity norm
- RMSProp (`torch.optim.RMSprop`)
  - Take the square root of the gradient average before adding epsilon to normalization of LR

# Network Hyperparameters

- Architecture choice
  - Fully connected/Linear layer
  - Convolutional Layer
  - Recurrent layer

# Lab Assignment: Fashion MNIST Classification

# The Fashion MNIST Dataset

- Labels 0–9 (Dress, shirt, coat, etc.)
- Data consists of grayscale images of fixed size (28x28) – flattens to 784
- Drop-in replacement for the original MNIST, but more complicated



# Fashion MNIST Classification Task

- Load the Fashion MNIST dataset using torchvision.datasets
- A fully-connected network for classification
- Try different optimizers, regularization, initialization and batch normalization and form a table of the results.
- Report your loss as "loss curve" and accuracy for different settings and draw conclusions

```
import torch
import torchvision

train_batch_size = # Define train batch size
test_batch_size = # Define test batch size (can be larger than train batch size)

# Use the following code to load and normalize the dataset
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.FashionMNIST('/files/', train=True, download=True,
                                     transform=torchvision.transforms.Compose([
                                         torchvision.transforms.ToTensor(),
                                         torchvision.transforms.Normalize(
                                             (0.1307,), (0.3081,))
                                     ])),
    batch_size=train_batch_size, shuffle=True)

test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.FashionMNIST('/files/', train=False, download=True,
                                     transform=torchvision.transforms.Compose([
                                         torchvision.transforms.ToTensor(),
                                         torchvision.transforms.Normalize(
                                             (0.1307,), (0.3081,))
                                     ])),
    batch_size=test_batch_size, shuffle=True)
```



# Optimizers to consider

- RMSProp
- Adam
- SGD

Which optimizer works well and in which aspects, such as convergence time, accuracy, training and testing loss why? Try to explain it.

# Regularization to consider

- L1 / L2
- Dropout layers  
(`torch.nn.Dropout`)

Apply regularization after  
check for over-fitting or  
under-fitting?

Consider how to balance regularization  
and loss optimization?

`torch.optim.Adam` parameter:

## Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[**float**, **float**], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight\_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **amsgrad** (*boolean, optional*) – whether to use the AMSGrad variant of this algorithm from the paper [On the Convergence of Adam and Beyond](#) (default: False)
- **maximize** (*bool, optional*) – maximize the params based on the objective, instead of minimizing (default: False)

# Initialization & Normalization to consider

- Random normal
- Xavier
- He (Kaiming)

Check Torch.nn.init:

<https://pytorch.org/docs/stable/nn.init.html>

- Batch Normalization
- Layer Normalization

How do they work?

# Hyperparameters to consider

- Number of layers
- Neurons in each layer
- Training Batch Size
- Learning Rate
- Activation function (ReLU vs tanh vs sigmoid)
- Number of training epochs