

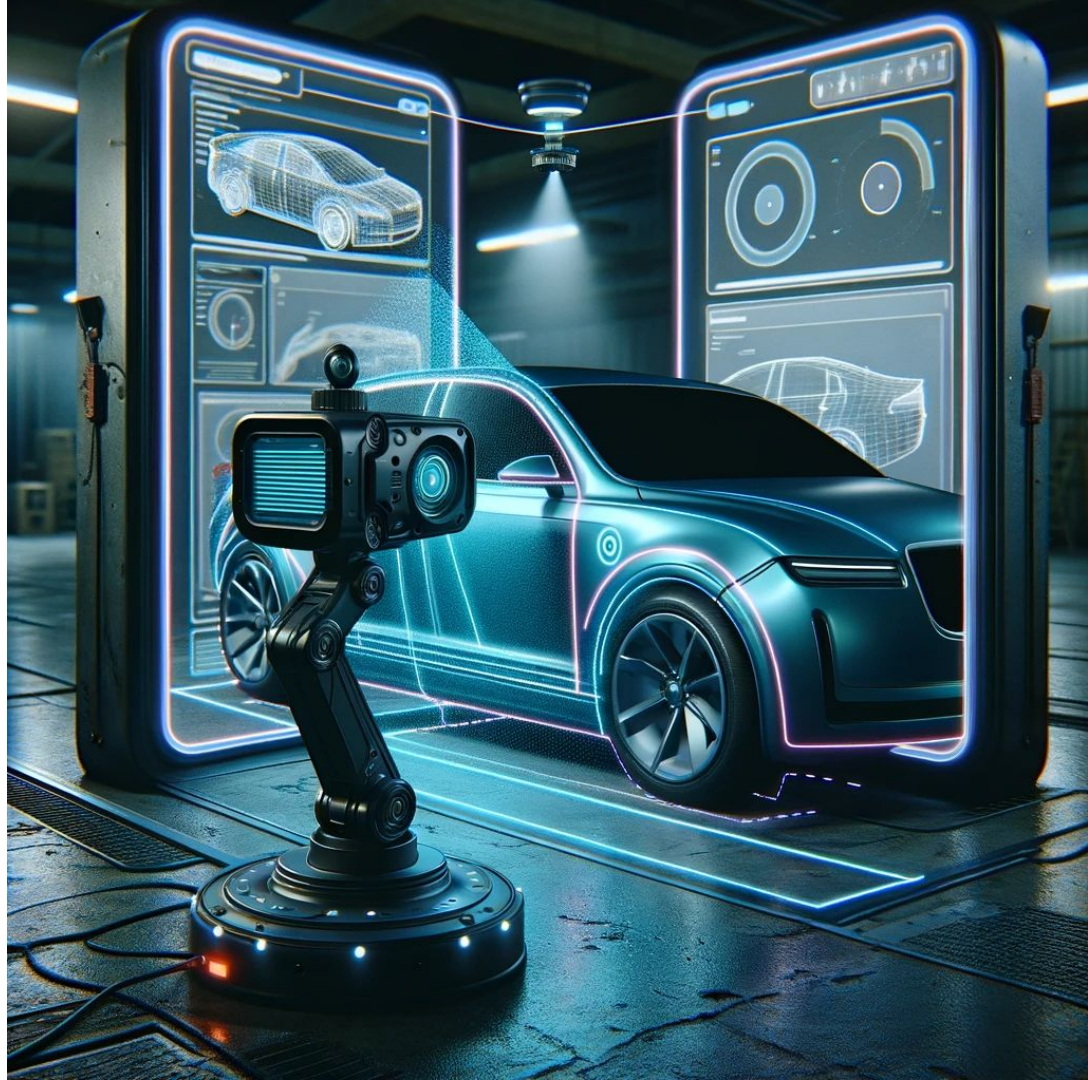
PPGEEC2318

Machine Learning

Computer Vision – Part I

Ivanovitch Silva

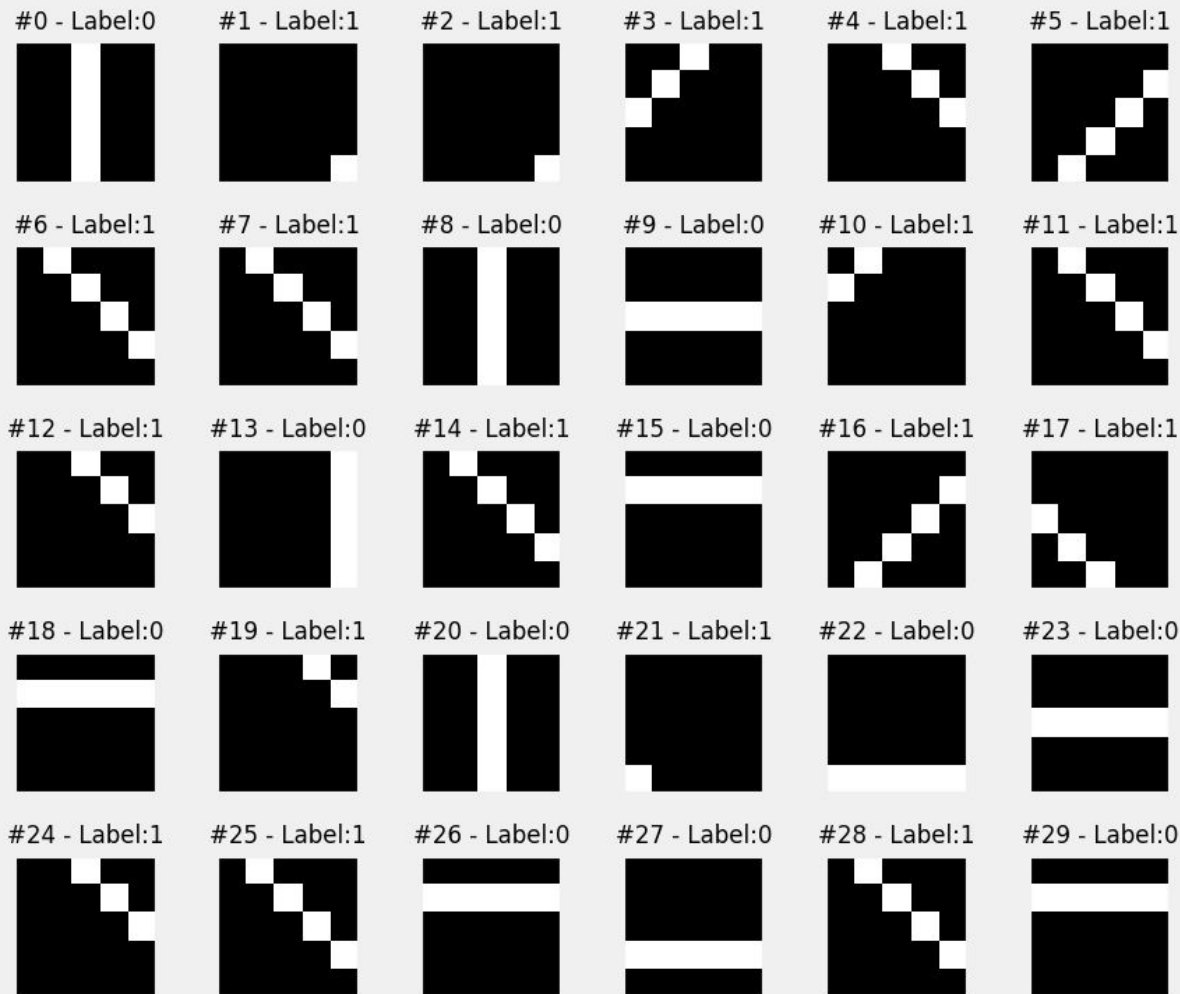
ivanovitch.silva@ufrn.br





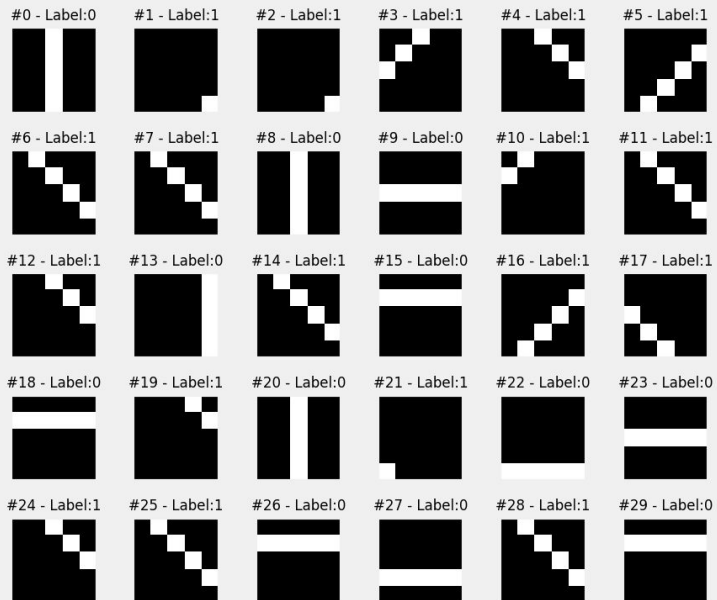
Is the line diagonal?

Data Generation



Classifying Images: Data Generation

If the line **is diagonal**, then we assume it belongs to the **positive class**. If it **is not diagonal**, it belongs to the **negative class**.



```
images[0]
array([[[ 0,  0, 255,  0,  0],
        [ 0,  0, 255,  0,  0],
        [ 0,  0, 255,  0,  0],
        [ 0,  0, 255,  0,  0],
        [ 0,  0, 255,  0,  0]]], dtype=uint8)

labels[0]
0
```

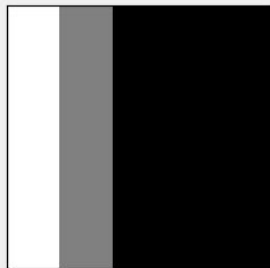
```
images, labels = generate_dataset(img_size=5, n_images=300, binary=True, seed=13)
```

Images and Channels

image_r

```
[[255 128 0 0 0]  
[255 128 0 0 0]  
[255 128 0 0 0]  
[255 128 0 0 0]  
[255 128 0 0 0]]
```

Red

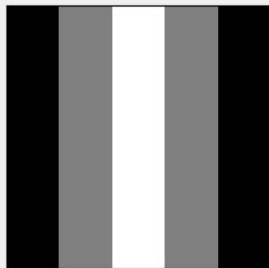


5x5

image_g

```
[[ 0 128 255 128 0]  
[ 0 128 255 128 0]  
[ 0 128 255 128 0]  
[ 0 128 255 128 0]  
[ 0 128 255 128 0]]
```

Green

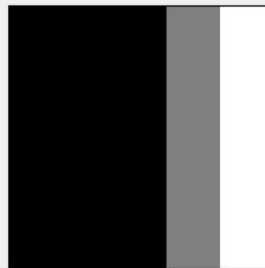


5x5

image_b

```
[[ 0 0 0 128 255]  
[ 0 0 0 128 255]  
[ 0 0 0 128 255]  
[ 0 0 0 128 255]  
[ 0 0 0 128 255]]
```

Blue

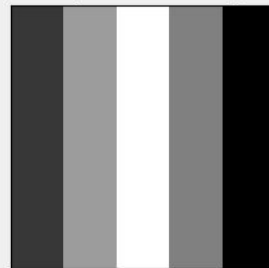


5x5

image_gray

```
[[ 54 118 182 100 18]  
[ 54 118 182 100 18]  
[ 54 118 182 100 18]  
[ 54 118 182 100 18]  
[ 54 118 182 100 18]]
```

Grayscale Image



5x5 = 0.21R + 0.72G + 0.07B

Single
Channel
(grayscale)

Images and Channels

image_r

```
[[255 128  0  0  0]
 [255 128  0  0  0]
 [255 128  0  0  0]
 [255 128  0  0  0]
 [255 128  0  0  0]]
```

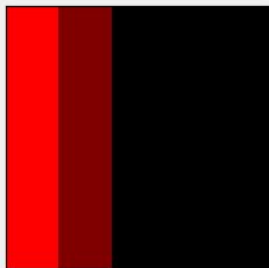
image_g

```
[[ 0 128 255 128  0]
 [ 0 128 255 128  0]
 [ 0 128 255 128  0]
 [ 0 128 255 128  0]
 [ 0 128 255 128  0]]
```

image_b

```
[[ 0  0  0 128 255]
 [ 0  0  0 128 255]
 [ 0  0  0 128 255]
 [ 0  0  0 128 255]
 [ 0  0  0 128 255]]
```

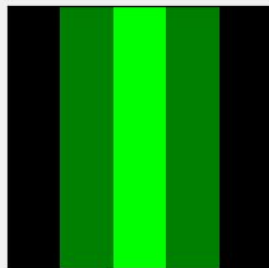
as first channel



Three
Channels
(color)

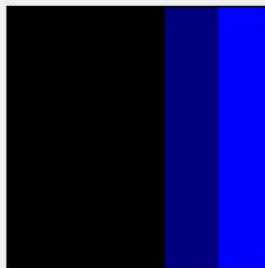
5x5x3 - R only

as second channel



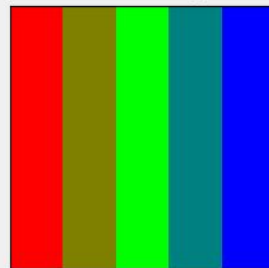
5x5x3 - G only

as third channel



5x5x3 - B only

RGB Image



5x5x3 = (R, G, B) stacked

Shape (NCHW vs NHWC)

- NCHW: (number of images, channels, height, width)
 - Pytorch
- NHWC: (number of images, height, width, channels)
 - Tensorflow
- HWC
 - PIL

```
images.shape  
(300, 1, 5, 5)
```

```
example = images[7]  
example  
array([[ [ 0, 255,  0,  0,  0],  
        [ 0,  0, 255,  0,  0],  
        [ 0,  0,  0, 255,  0],  
        [ 0,  0,  0,  0, 255],  
        [ 0,  0,  0,  0,  0]], dtype=uint8)
```



From dataset to Compose transforms

Torchvision: data transformation

Torchvision is a package containing popular datasets, model architectures, and common image transformations for computer vision.

main 123 Branches 144 Tags

Go to file

Add file

<> Code

About

Datasets, Transforms and Models specific to Computer Vision

pytorch.org/vision

machine-learning

computer-vision

Readme

BSD-3-Clause license

Code of conduct

Cite this repository

Activity

Custom properties

15.7k stars

420 watching

6.9k forks

Report repository

Releases 42

NicolasHug Typo fix: collab -> Colab (#8512) ✖

4d82b31 · 3 hours ago 3,884 Commits

.github	Typo fix: collab -> Colab (#8512)	3 hours ago
android	CXX_STANDARD 14 -> 17 (#7608)	last year
benchmarks	Adding GPU acceleration to encode_jpeg (#8391)	3 weeks ago
cmake	Remove include Python.h (#8413)	2 months ago
docs	Typo fix: collab -> Colab (#8512)	3 hours ago
examples	Rename cpp example README.rst to README.md (#8467)	last month
gallery	Typo fix: collab -> Colab (#8512)	3 hours ago
ios	CXX_STANDARD 14 -> 17 (#7608)	last year
packaging	Cleanup C++ model example and cmake CI job (#8411)	2 months ago
references	Fix use of cutmix_alpha in classification training reference...	2 months ago
scripts	Update classify_prs notebook (#8383)	3 months ago

```
from torchvision.transforms import Compose, ToTensor
from torchvision.transforms import Normalize, ToPILImage
from torchvision.transforms import RandomHorizontalFlip, Resize
```

```
composer = Compose([RandomHorizontalFlip(p=1.0),
                    Normalize(mean=(.5,), std=(.5,))])
composed_tensor = composer(example_tensor)
```

$$\begin{aligned}\text{input} = 0 &\implies \frac{0 - \text{mean}}{\text{std}} = \frac{0 - 0.5}{0.5} = -1 \\ \text{input} = 1 &\implies \frac{1 - \text{mean}}{\text{std}} = \frac{1 - 0.5}{0.5} = 1\end{aligned}$$



Convert, build and transform tensors into useful format

Data Preparation

Splitting the dataset: build a custom splitter

```
def index_splitter(n, splits, seed=13):  
    idx = torch.arange(n)  
    # Makes the split argument a tensor  
    splits_tensor = torch.as_tensor(splits)  
    # Finds the correct multiplier, so we don't have  
    # to worry about summing up to N (or one)  
    multiplier = n / splits_tensor.sum()  
    splits_tensor = (multiplier * splits_tensor).long()  
    # If there is a difference, throws at the first split  
    # so random_split does not complain  
    diff = n - splits_tensor.sum()  
    splits_tensor[0] += diff  
    # Uses PyTorch random_split to split the indices  
    torch.manual_seed(seed)  
    return random_split(idx, splits_tensor)
```

```
train_idx, val_idx = index_splitter(len(x_tensor), [80, 20])  
train_idx, val_idx = index_splitter(len(x_tensor), [0.8, 0.2])
```

Splitting the dataset: build a custom sampler

```
# Builds a loader of each set
train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=16,
    shuffle=True
)
val_loader = DataLoader(
    dataset=val_dataset,
    batch_size=16)
```

Before

```
# Builds a loader of each set
train_loader = DataLoader(
    dataset=dataset,
    batch_size=16,
    sampler=train_sampler)
val_loader = DataLoader(
    dataset=dataset,
    batch_size=16,
    sampler=val_sampler)
```

Option 1

```
train_sampler = SubsetRandomSampler(train_idx)
val_sampler = SubsetRandomSampler(val_idx)
```

Splitting the dataset: build a weighted sampler

```
x_train_tensor = x_tensor[train_idx]
y_train_tensor = y_tensor[train_idx]

x_val_tensor = x_tensor[val_idx]
y_val_tensor = y_tensor[val_idx]

classes, counts = y_train_tensor.unique(return_counts=True)
print(classes, counts)
tensor([0., 1.]) tensor([ 80, 160])
```

```
weights = 1.0 / counts.float()
weights
tensor([0.0125, 0.0063])
```



The class with fewer data points (minority class) should get larger weights, while the class with more data points (majority class) should get smaller weights. This way, on average, we'll end up with mini-batches containing roughly the same number of data points in each class: **A balanced dataset.**

Splitting the dataset: build a weighted sampler

```
weights
tensor([0.0125, 0.0063])

y_train_tensor.squeeze().long()
tensor([1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1,
        1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1,
        0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1,
        1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1,
        1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1,
        1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1,
        0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1,
        1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1,
        1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1,
        1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1])
```

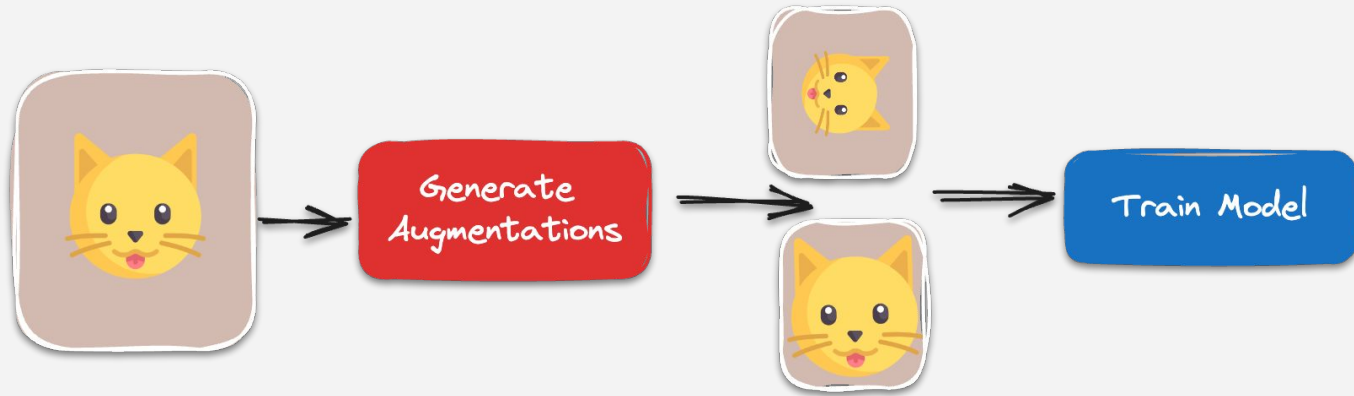
```
weights[y_train_tensor.squeeze().long()][:10]
tensor([0.0063, 0.0063, 0.0063, 0.0063, 0.0063, 0.0125, 0.0063, 0.0063, 0.0063,
        0.0063])
```

Splitting the dataset: build a weighted sampler

```
# Create an instance of the Generator class from PyTorch
generator = torch.Generator()

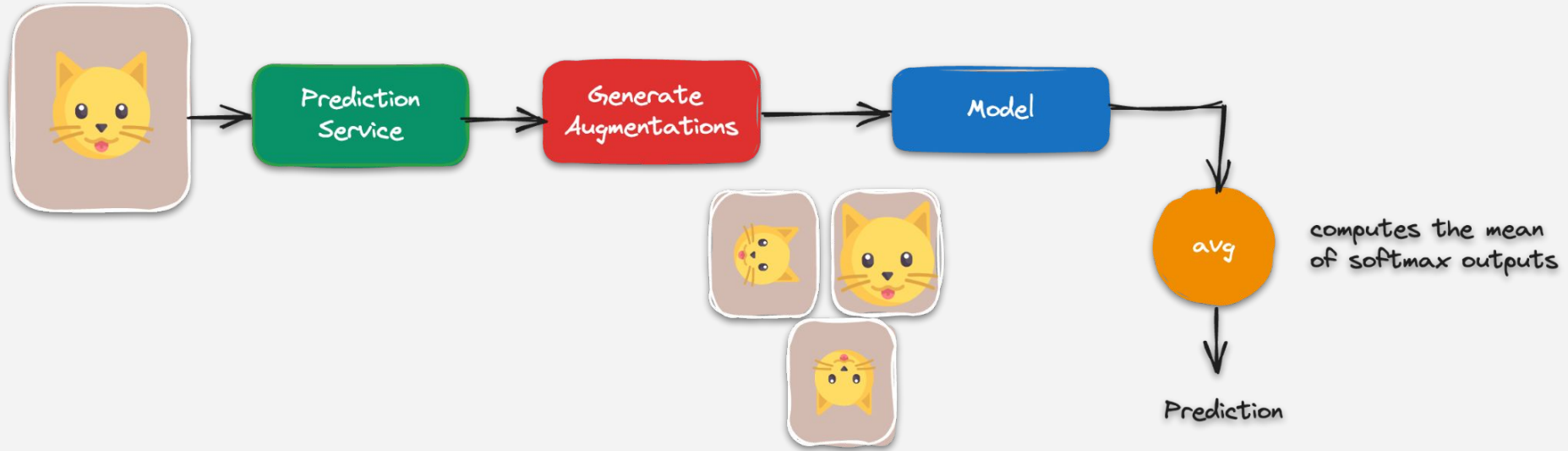
# Create a WeightedRandomSampler instance. This sampler will select elements randomly
# based on the specified weights but with a mechanism that allows for replacement,
# meaning the same item can be chosen more than once.
sampler = WeightedRandomSampler(
    weights=sample_weights,          # An array of weights, where each weight corresponds to a sample.
    num_samples=len(sample_weights), # The total number of samples to draw (equal to the length of weights).
    generator=generator,             # The random number generator object for reproducibility.
    replacement=True                 # Allows for the same item to be selected more than once.
)
```


Splitting the dataset: all you need is data augmentation



In general, we want to apply data augmentation to the **training data only**.

Splitting the dataset: all you need is data augmentation



(...) there is also "test-time augmentation", which can be used to improve the performance of a model after it is deployed.

Splitting the dataset: all you need is data augmentation

```
class TransformTensorDataset(Dataset):
    def __init__(self, x, y, transform=None):
        self.x = x
        self.y = y
        self.transform = transform

    def __getitem__(self, index):
        x = self.x[index]

        if self.transform:
            x = self.transform(x)

        return x, self.y[index]

    def __len__(self):
        return len(self.x)
```

```
train_composer = Compose([RandomHorizontalFlip(p=.5),
                          Normalize(mean=(.5,), std=(.5,))])

val_composer = Compose([Normalize(mean=(.5,), std=(.5,))])
```

[illegible]

Splitting the dataset: putting it together

[illegible]

Splitting the
dataset:
putting it
together

```
# Builds a weighted random sampler to handle imbalanced classes
sampler = make_balanced_sampler(y_train_tensor)

# Uses sampler in the training set to get a balanced data loader
train_loader = DataLoader(dataset=train_dataset,
                           batch_size=16,
                           sampler=sampler)
val_loader = DataLoader(dataset=val_dataset,
                        batch_size=16)
```

Splitting the dataset: pixels as features

	F1	F2	F3	F4	
Instances	0				
	1				
	2				
	3				
					Target



Original Image
(pixels)

Flatten
→



Splitting the dataset: pixels as features

```
import torch.nn as nn

dummy_xs, dummy_ys = next(iter(train_loader))
dummy_xs.shape
torch.Size([16, 1, 5, 5])

flattener = nn.Flatten()
dummy_xs_flat = flattener(dummy_xs)

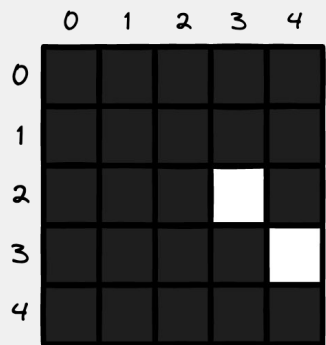
print(dummy_xs_flat.shape)
print(dummy_xs_flat[0])

torch.Size([16, 25])
tensor([-1., -1., -1.,  1., -1., -1., -1., -1.,  1., -1., -1., -1., -1.,  1., -1., -1., -1.,
        -1.,  1., -1., -1., -1., -1.,  1., -1.])
```



Defining a model to handle a binary classification task

Shallow Model

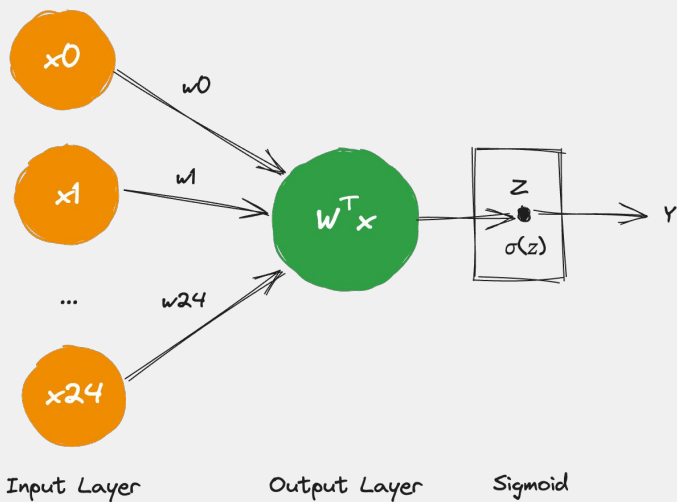


Original Image



Flatted Image

$$P(y = 1) = \sigma(z) = \sigma(w_0x_0 + w_1x_1 + \dots + w_{24}x_{24})$$



$$W = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{24} \end{bmatrix}_{(25 \times 1)}; X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{24} \end{bmatrix}_{(25 \times 1)}$$

$$z = W^T \cdot X = \begin{bmatrix} - & w^T & - \end{bmatrix}_{(1 \times 25)} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{24} \end{bmatrix}_{(25 \times 1)} = \begin{bmatrix} w_0 & w_1 & \dots & w_{24} \end{bmatrix}_{(1 \times 25)} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{24} \end{bmatrix}_{(25 \times 1)}$$

$$= w_0x_0 + w_1x_1 + \dots + w_{24}x_{24}$$

```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(17)

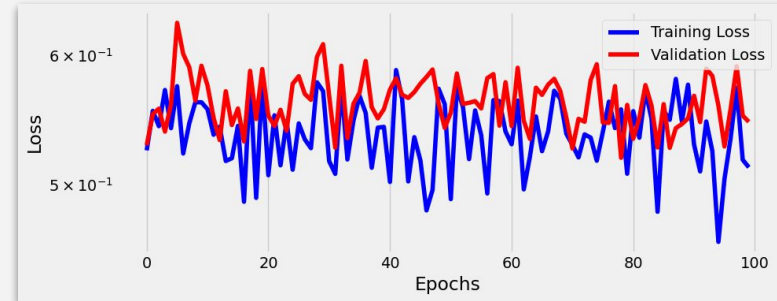
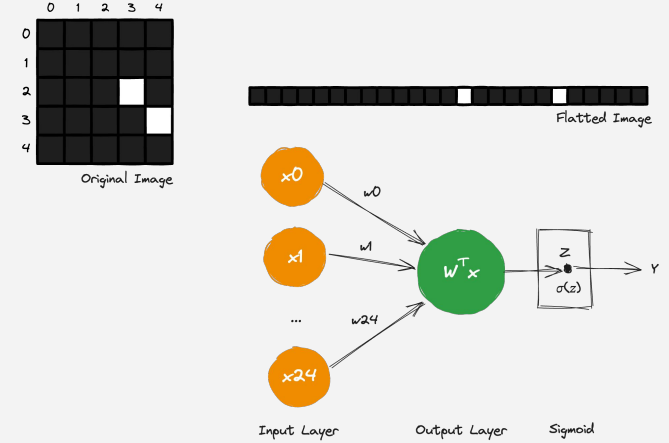
# Now we can create a model
model_logistic = nn.Sequential()
model_logistic.add_module('flatten', nn.Flatten())
model_logistic.add_module('output', nn.Linear(25, 1, bias=False))
model_logistic.add_module('sigmoid', nn.Sigmoid())

# Defines a SGD optimizer to update the parameters
optimizer_logistic = optim.SGD(model_logistic.parameters(), lr=lr)

# Defines a binary cross entropy loss function
binary_loss_fn = nn.BCELoss()
```

```
n_epochs = 100
```

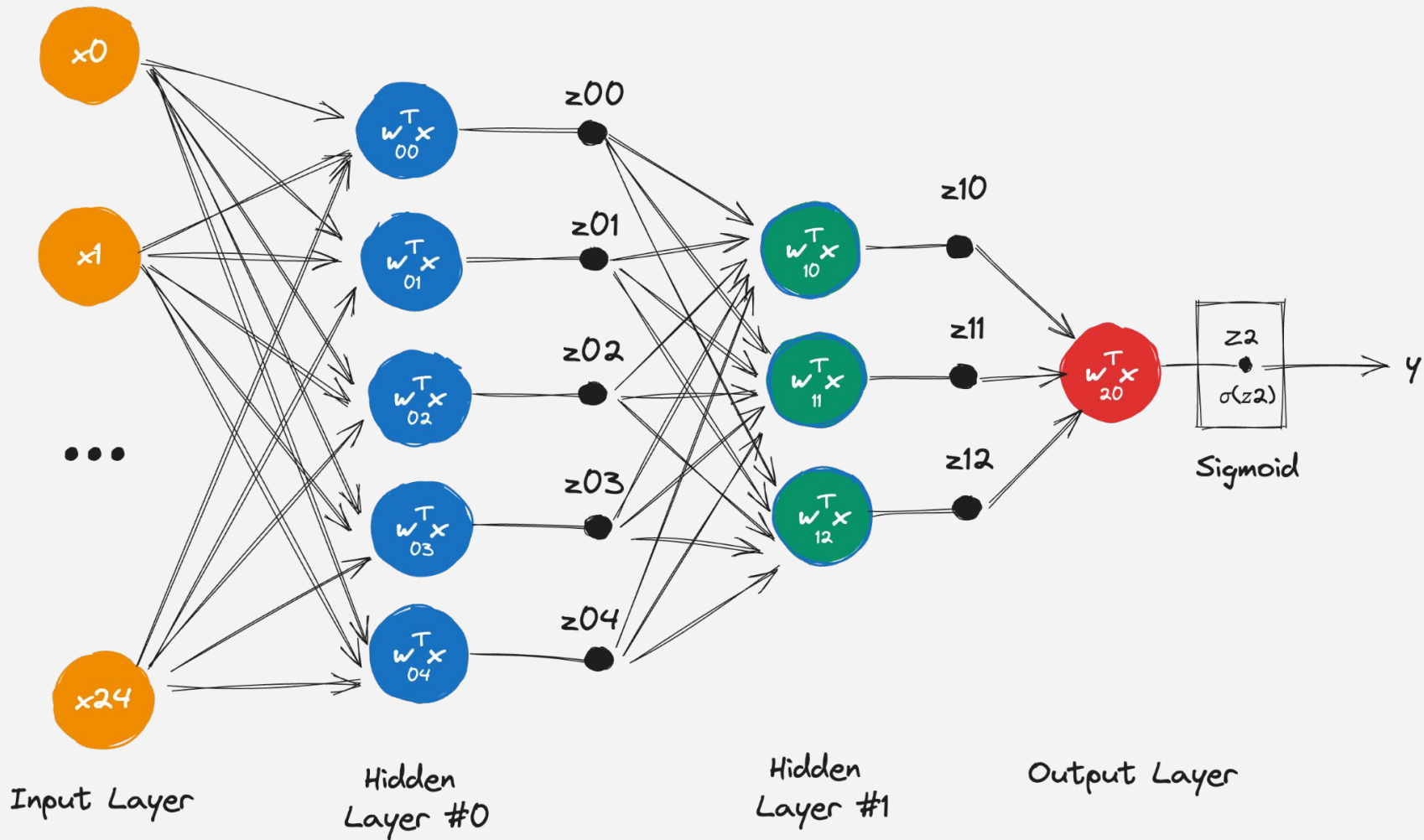
```
reg_logistic = Architecture(model_logistic, binary_loss_fn,
                             optimizer_logistic)
reg_logistic.set_loaders(train_loader, val_loader)
reg_logistic.train(n_epochs)
```

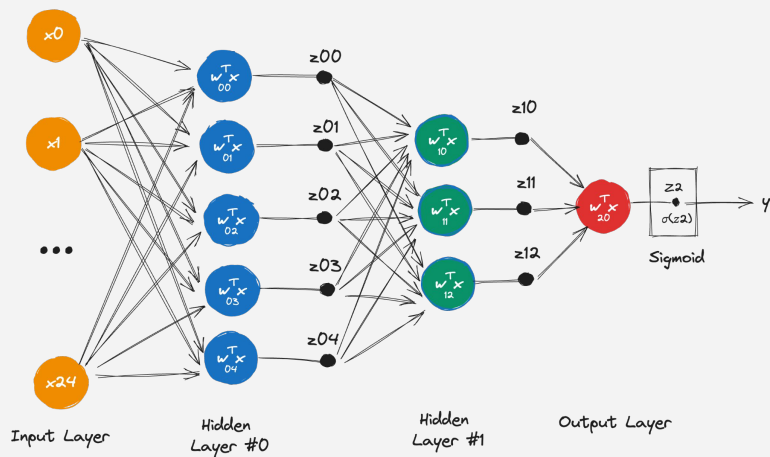




There we go, let's add not one, but two hidden layers

Deep-ish Model



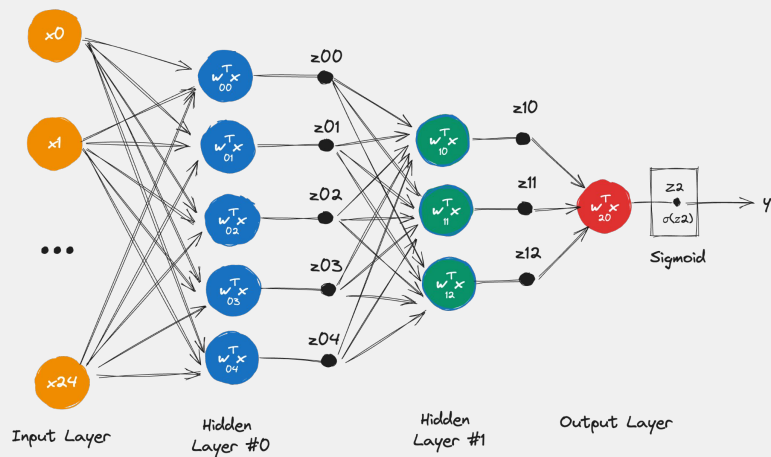


```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1
```

```
torch.manual_seed(17)
# Now we can create a model
model_nn = nn.Sequential()
model_nn.add_module('flatten', nn.Flatten())
model_nn.add_module('hidden0', nn.Linear(25, 5, bias=False))
model_nn.add_module('hidden1', nn.Linear(5, 3, bias=False))
model_nn.add_module('output', nn.Linear(3, 1, bias=False))
model_nn.add_module('sigmoid', nn.Sigmoid())
```

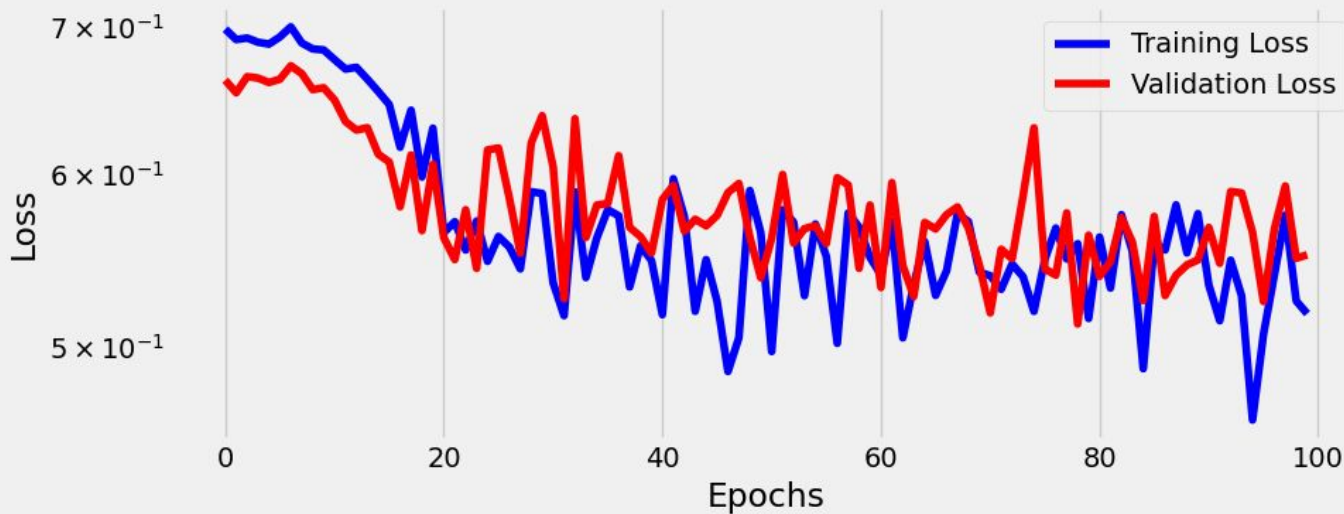
```
# Defines a SGD optimizer to update the parameters
optimizer_nn = optim.SGD(model_nn.parameters(), lr=lr)
```

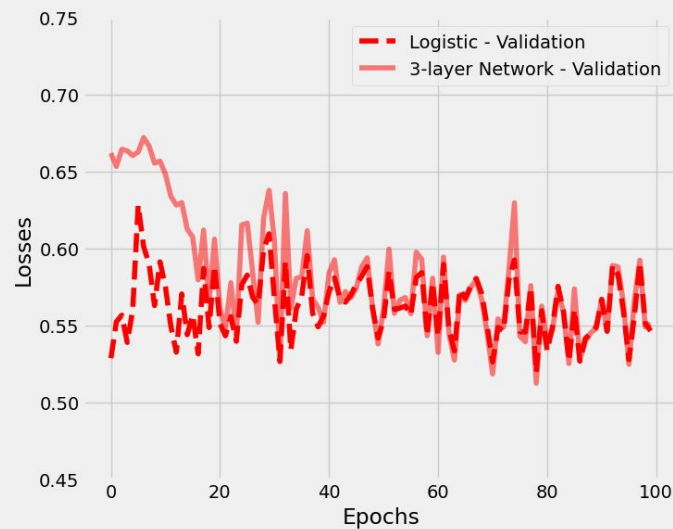
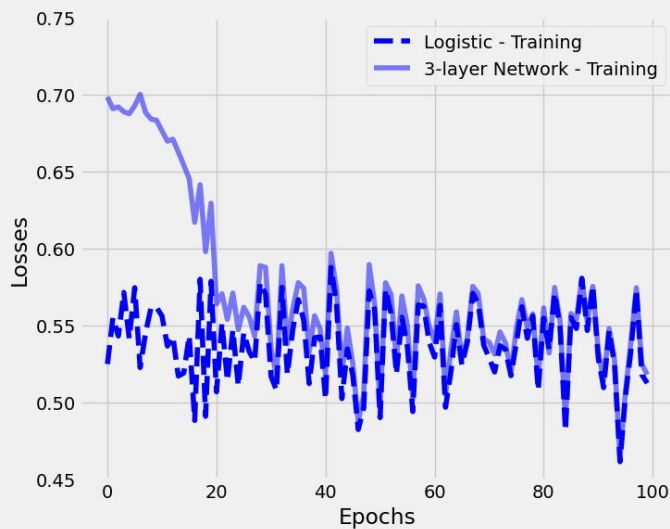
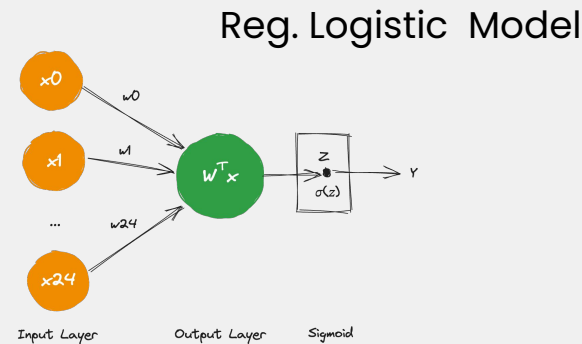
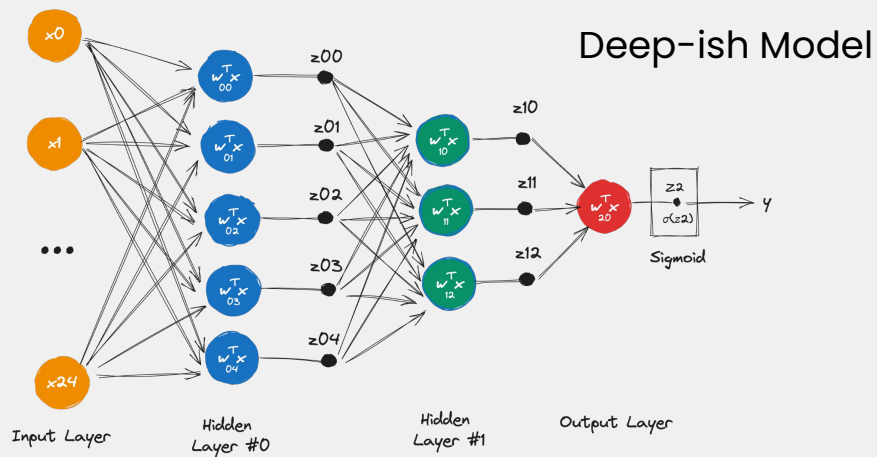
```
# Defines a binary cross entropy loss function
binary_loss_fn = nn.BCELoss()
```

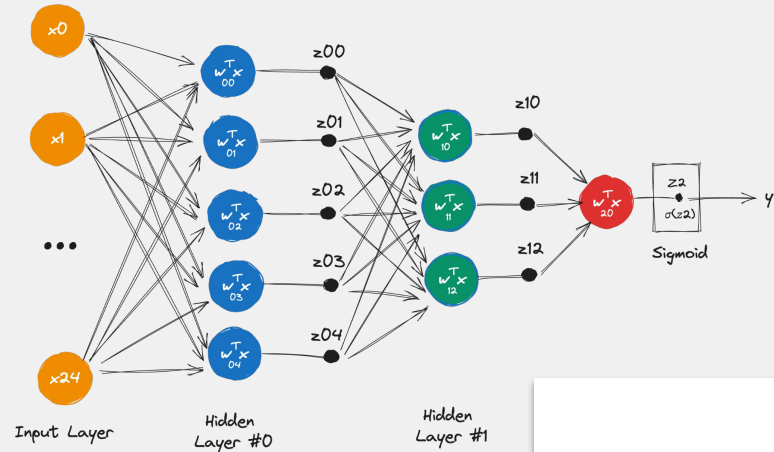


`n_epochs = 100`

```
arch_nn = Architecture(model_nn, binary_loss_fn, optimizer_nn)
arch_nn.set_loaders(train_loader, val_loader)
arch_nn.train(n_epochs)
```







Hidden #0

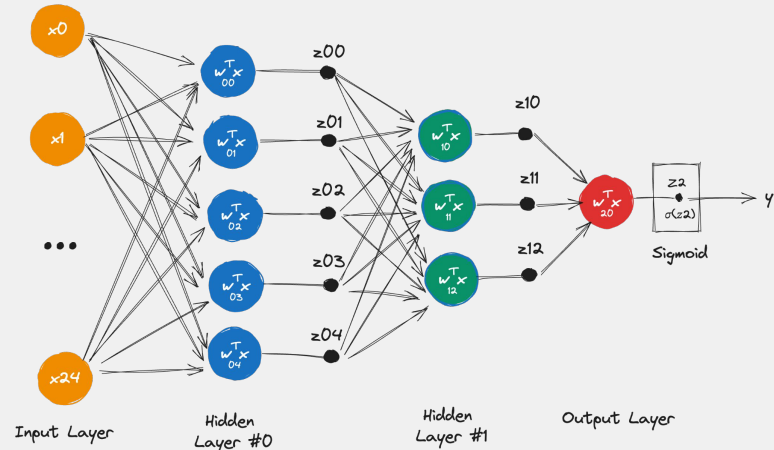
$$\begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}_{(5 \times 1)} = \begin{bmatrix} - & w^T_{00} & - \\ - & w^T_{01} & - \\ - & w^T_{02} & - \\ - & w^T_{03} & - \\ - & w^T_{04} & - \end{bmatrix}_{(5 \times 25)} \begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}_{(25 \times 1)}$$

Hidden #1

$$\begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}_{(3 \times 1)} = \begin{bmatrix} - & w^T_{10} & - \\ - & w^T_{11} & - \\ - & w^T_{12} & - \end{bmatrix}_{(3 \times 5)}$$

$$\begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}_{(5 \times 1)}$$

$$\text{Output } \begin{bmatrix} z_2 \end{bmatrix}_{(1 \times 1)} = \begin{bmatrix} - & w^T_{20} & - \end{bmatrix}_{(1 \times 3)} \begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}_{(3 \times 1)}$$



Hidden #0

$$\begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}_{(5 \times 1)} = \begin{bmatrix} - & w_{00}^T & - \\ - & w_{01}^T & - \\ - & w_{02}^T & - \\ - & w_{03}^T & - \\ - & w_{04}^T & - \end{bmatrix}_{(5 \times 25)} \begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}_{(25 \times 1)}$$

Hidden #1

$$\begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}_{(3 \times 1)} = \begin{bmatrix} - & w_{10}^T & - \\ - & w_{11}^T & - \\ - & w_{12}^T & - \end{bmatrix}_{(3 \times 5)}$$

$$\text{Output } \begin{bmatrix} z_2 \end{bmatrix}_{(1 \times 1)} = \begin{bmatrix} - & w_{20}^T & - \end{bmatrix}_{(1 \times 3)} \begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}_{(3 \times 1)}$$

substituting z' 's...

$$\begin{bmatrix} z_2 \end{bmatrix}_{(1 \times 1)} = \begin{bmatrix} - & w_{20}^T & - \end{bmatrix}_{(1 \times 3)}$$

Output Layer

$$\begin{bmatrix} - & w_{10}^T & - \\ - & w_{11}^T & - \\ - & w_{12}^T & - \end{bmatrix}_{(3 \times 5)}$$

Hidden Layer #1

$$\begin{bmatrix} - & w_{00}^T & - \\ - & w_{01}^T & - \\ - & w_{02}^T & - \\ - & w_{03}^T & - \\ - & w_{04}^T & - \end{bmatrix}_{(5 \times 25)}$$

Hidden Layer #0

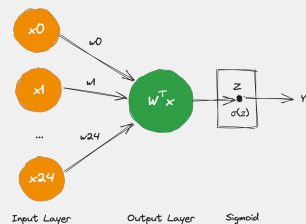
multiplying...

$$= \begin{bmatrix} - & w^T & - \end{bmatrix}_{(1 \times 25)}$$

Matrices Multiplied

$$\begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}_{(25 \times 1)}$$

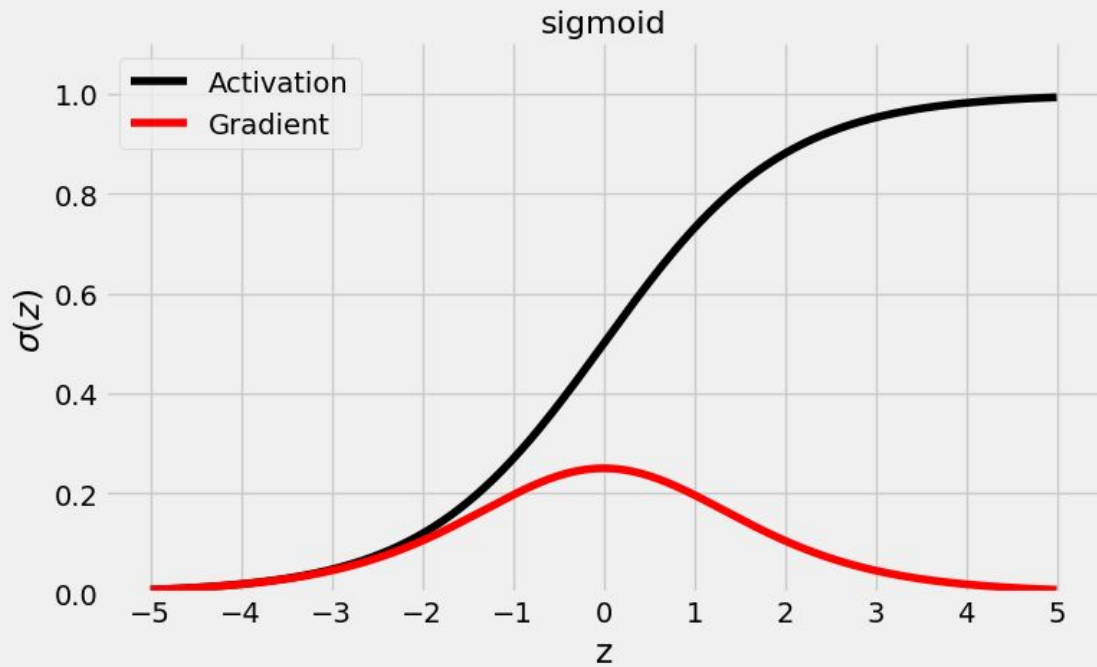
This is just the
reg. logistic





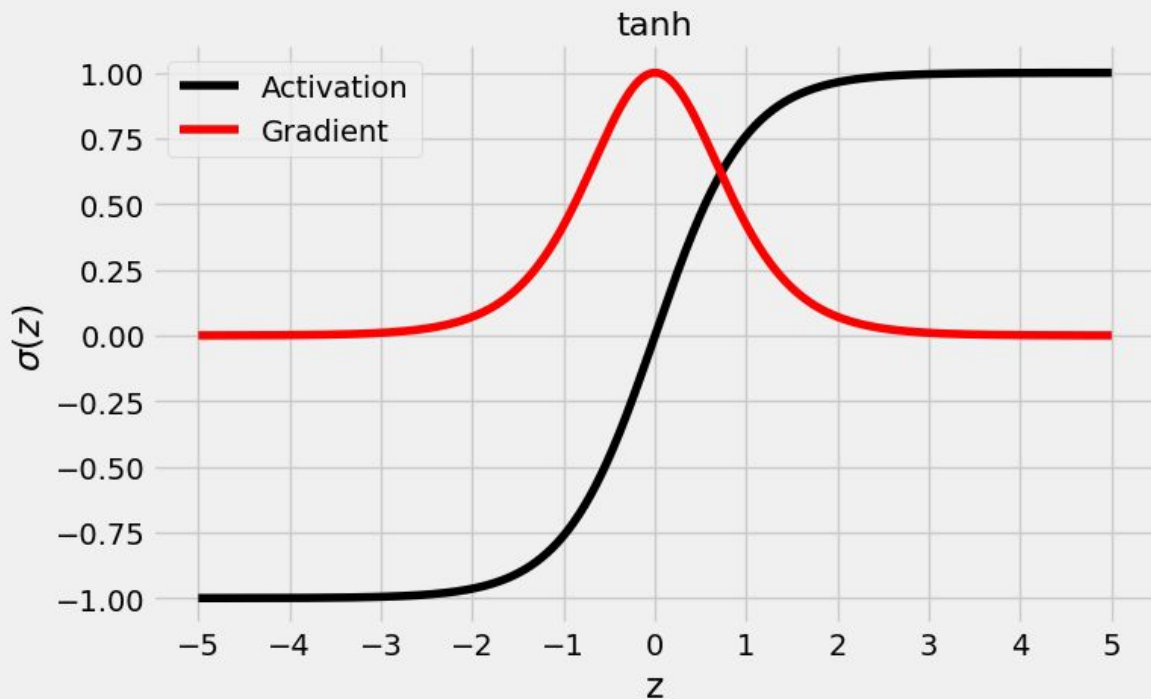
Nonlinear functions either squash or bend straight lines

Activation Functions



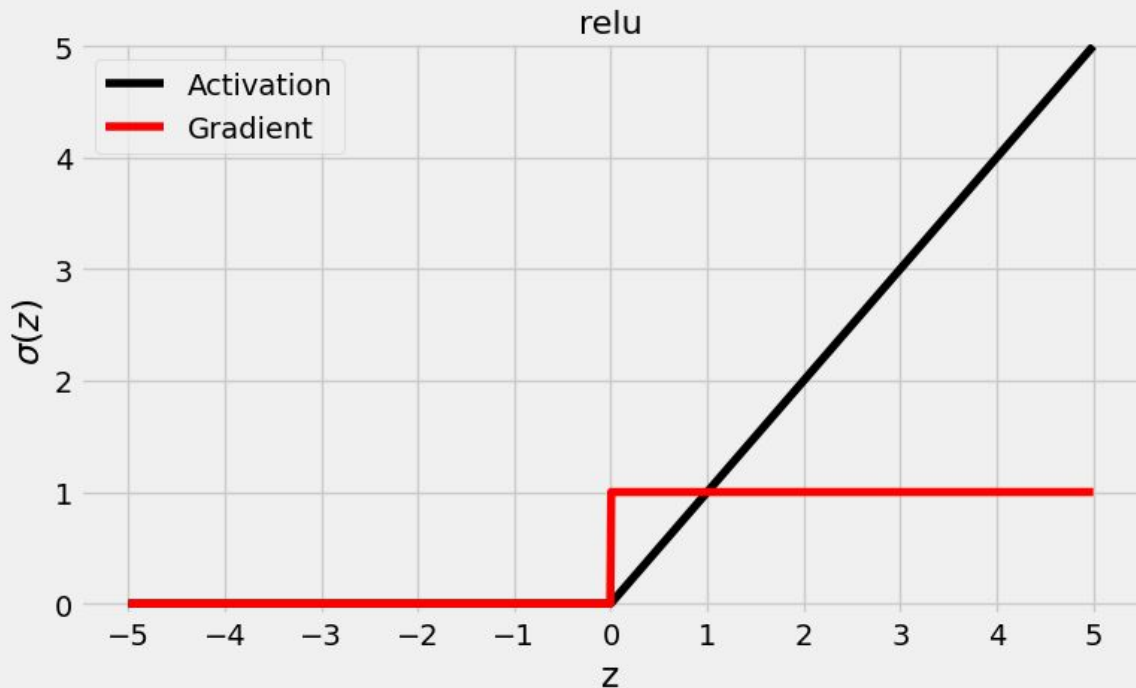
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- Range (0,1) same range probabilities can take, which is why it is used in the output layer for binary classification tasks.
- Gradient peak value is only 0.25 (for $z=0$).
- Gradient gets close to zero as the absolute value of z reaches a value of -5 or 5 .
- The activation values are going to be centered around 0.5, instead of zero. This means that, even if we normalize our inputs to feed the first layer, it will not be the case anymore for the other layers.



$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- TanH activation function "squashes" the input values into the range $(-1, 1)$.
- Being centered at zero, the activation values are already (somewhat) normalized inputs for the next layer, making the hyperbolic tangent a better activation function than the sigmoid.
- Gradient has a much larger peak value of 1.0 (again, for $z = 0$), but its decrease is even faster, approaching zero to absolute values of z as low as three. This is the underlying cause of what is referred to as the problem of **vanishing gradients**.

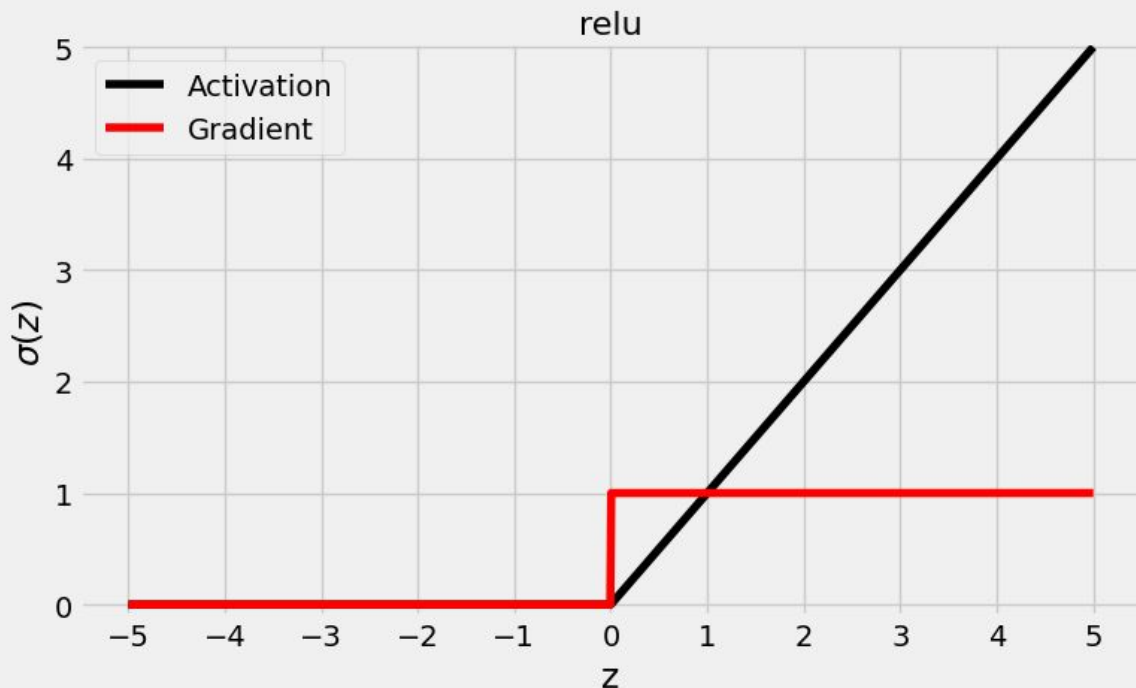


$$\sigma(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

or

$$\sigma(z) = \max(0, z)$$

- RELU addresses the problem of vanishing gradients found with its two predecessors, while also being the fastest to compute gradients for. It does not "squash" the values into a range—it simply preserves positive values and turns all negative values into zero. This pattern leads for a faster convergence.
- This behavior can also lead to what is called a "**dead neuron**"; that is, a neuron whose inputs are consistently negative and, therefore, always has an activation value of zero.
- Worse yet, the gradient is also zero for negative inputs, meaning the weights are not updated. It's like the **neuron got stuck**.

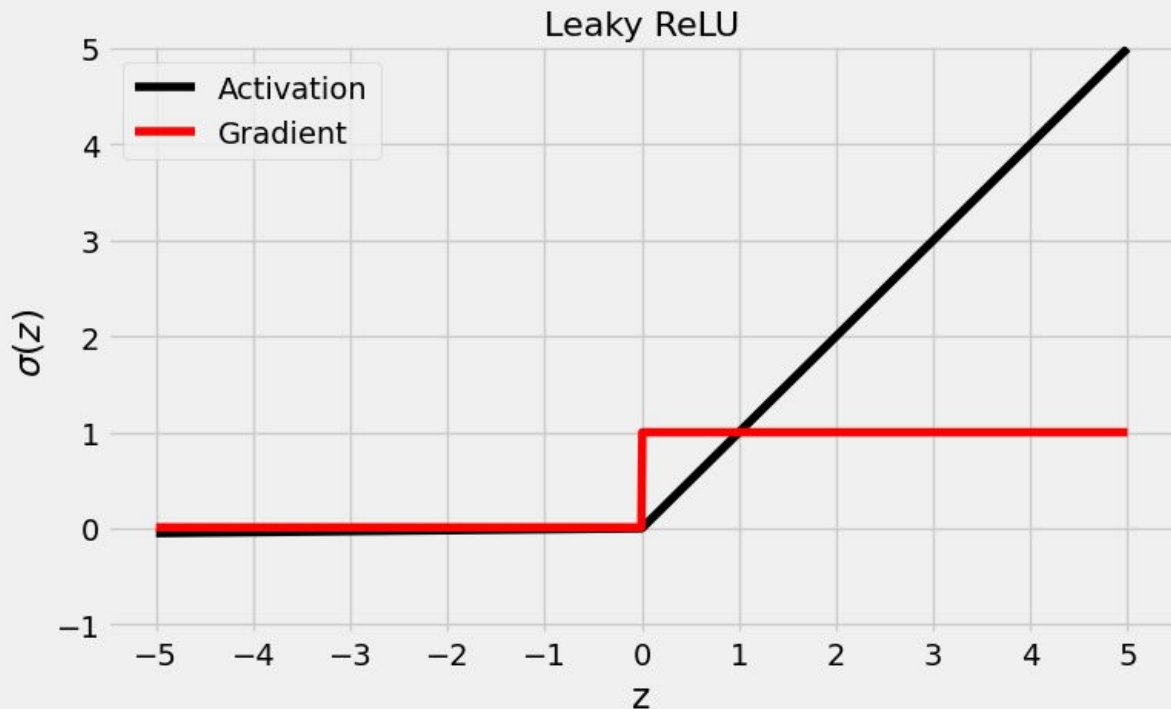


$$\sigma(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

or

$$\sigma(z) = \max(0, z)$$

- The activation values of the ReLU are obviously not centered at zero.
- For deeper and more complex models, this may become an issue commonly called "**internal covariate shift**," which is just fancy for there being different distributions of activation values in different layers.
- In general, **we would like to have all layers producing activation values with similar distributions**, ideally zero centered and with unit standard deviation. **To address this issue, you can use normalization layers (batch normalization).**

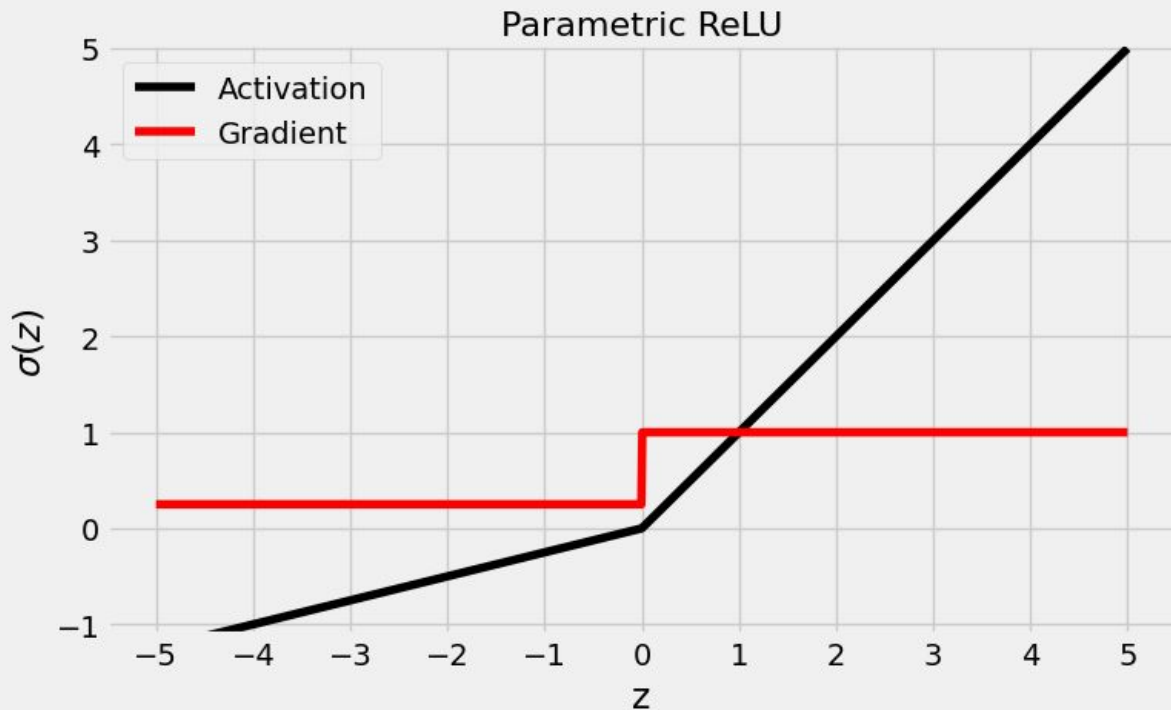


$$\sigma(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0.01z, & \text{if } z < 0 \end{cases}$$

or

$$\sigma(z) = \max(0, z) + 0.01 \min(0, z)$$

- For negative inputs, it returns a tiny activation value and yields a tiny gradient, instead of a fixed zero for both. It may not be much, but **it gives the neuron a chance to get unstuck**.
- The multiplier for negative values, 0.01, is called the **coefficient of leakage**.



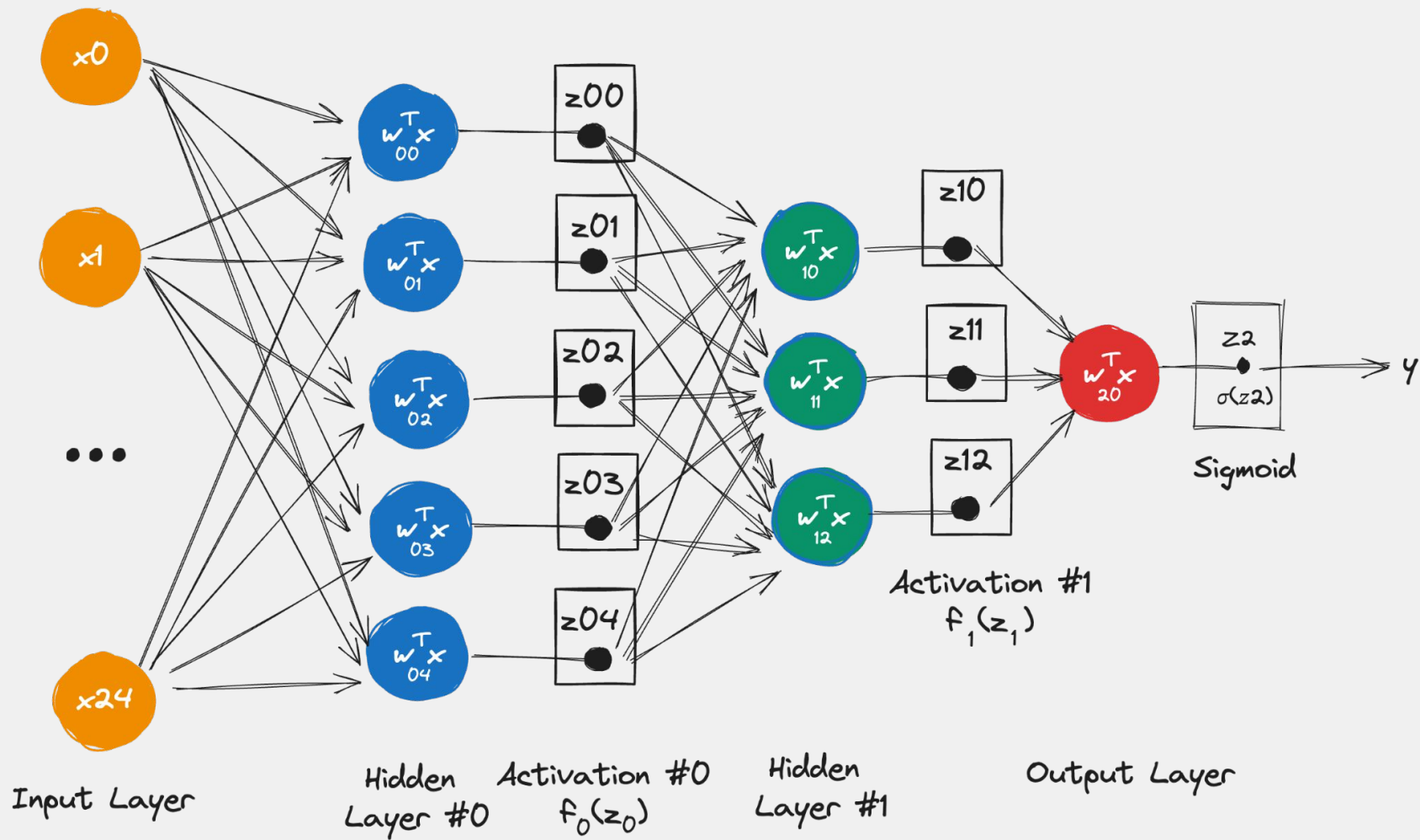
$$\sigma(z) = \begin{cases} z, & \text{if } z \geq 0 \\ az, & \text{if } z < 0 \end{cases}$$

or

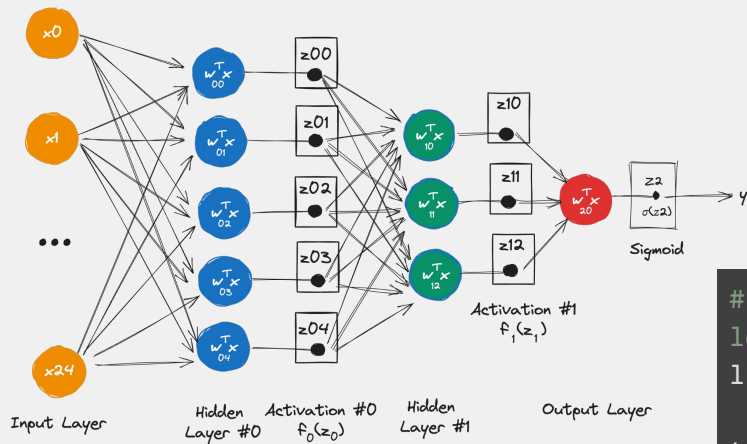
$$\sigma(z) = \max(0, z) + a \min(0, z)$$

- The Parametric ReLU is the natural evolution of the Leaky ReLU: Instead of arbitrarily choosing a coefficient of leakage (such as 0.01), let's make it a parameter (a).
- We can set the initial value for a despite it is going to be learned.

Deep Model



Deep Model



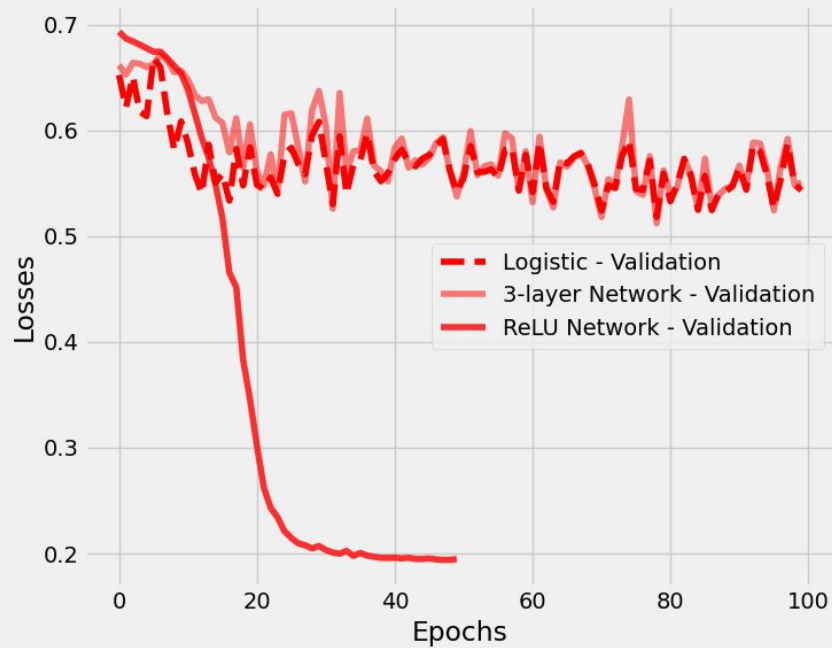
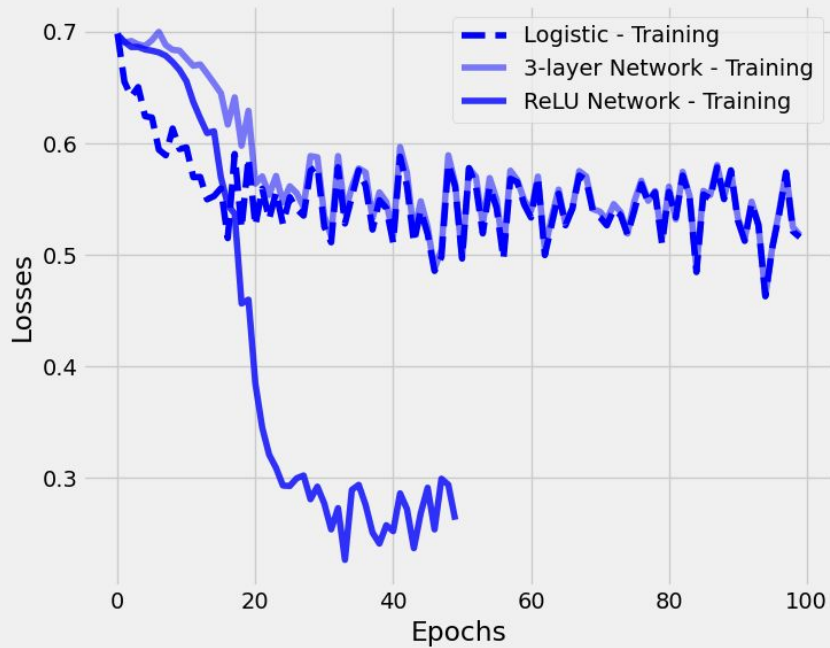
```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(17)
# Now we can create a model
model_relu = nn.Sequential()
model_relu.add_module('flatten', nn.Flatten())
model_relu.add_module('hidden0', nn.Linear(25, 5, bias=False))
model_relu.add_module('activation0', nn.ReLU())
model_relu.add_module('hidden1', nn.Linear(5, 3, bias=False))
model_relu.add_module('activation1', nn.ReLU())
model_relu.add_module('output', nn.Linear(3, 1, bias=False))
model_relu.add_module('sigmoid', nn.Sigmoid())

# Defines a SGD optimizer to update the parameters
optimizer_relu = optim.SGD(model_relu.parameters(), lr=lr)

# Defines a binary cross entropy loss function
binary_loss_fn = nn.BCELoss()
```

Deep Model



Bonus Chapter: feature space

