PPGEEC2318

Machine Learning

Computer Vision - Part I

Ivanovitch Silva

ivanovitch.silva@ufrn.br
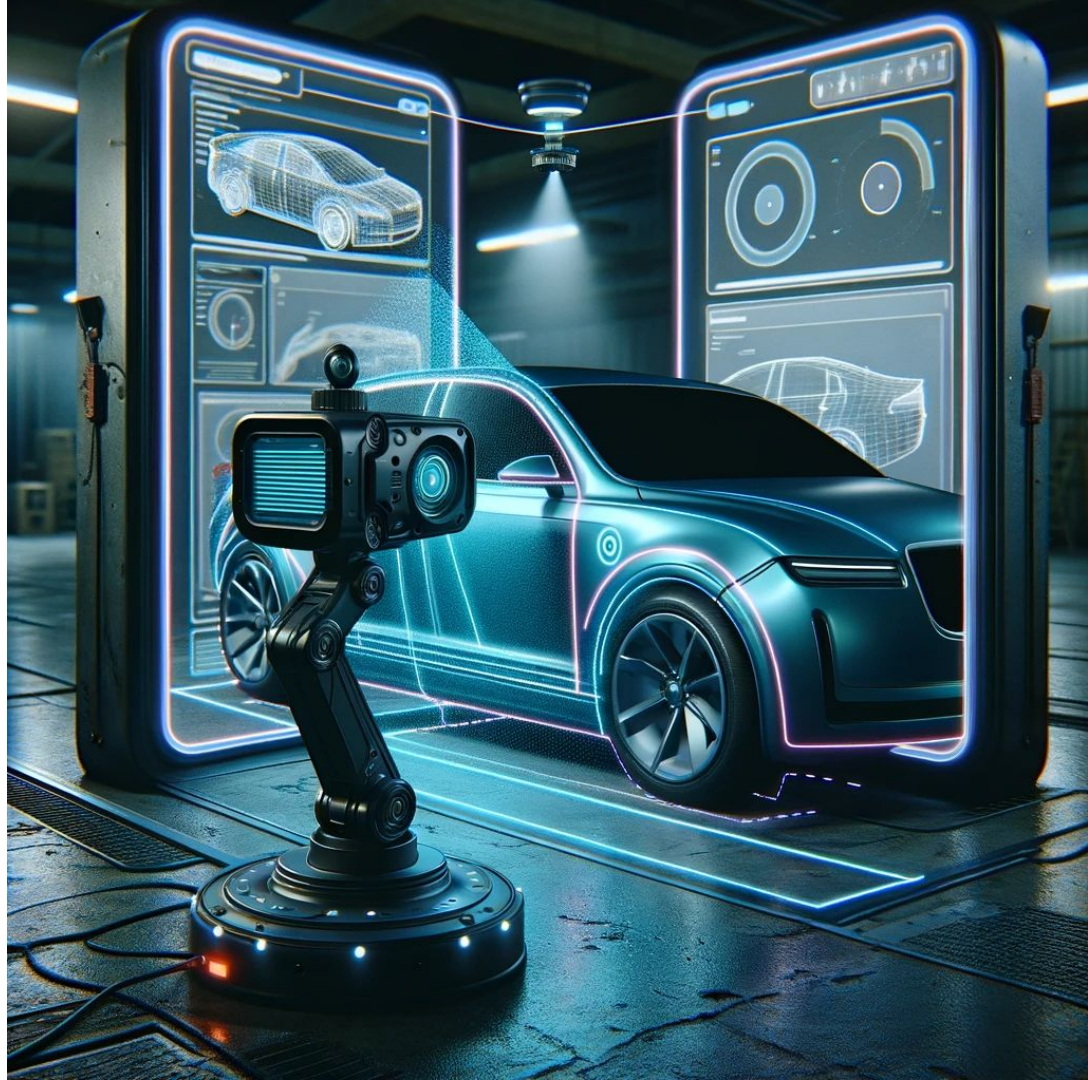
Is the line diagonal?

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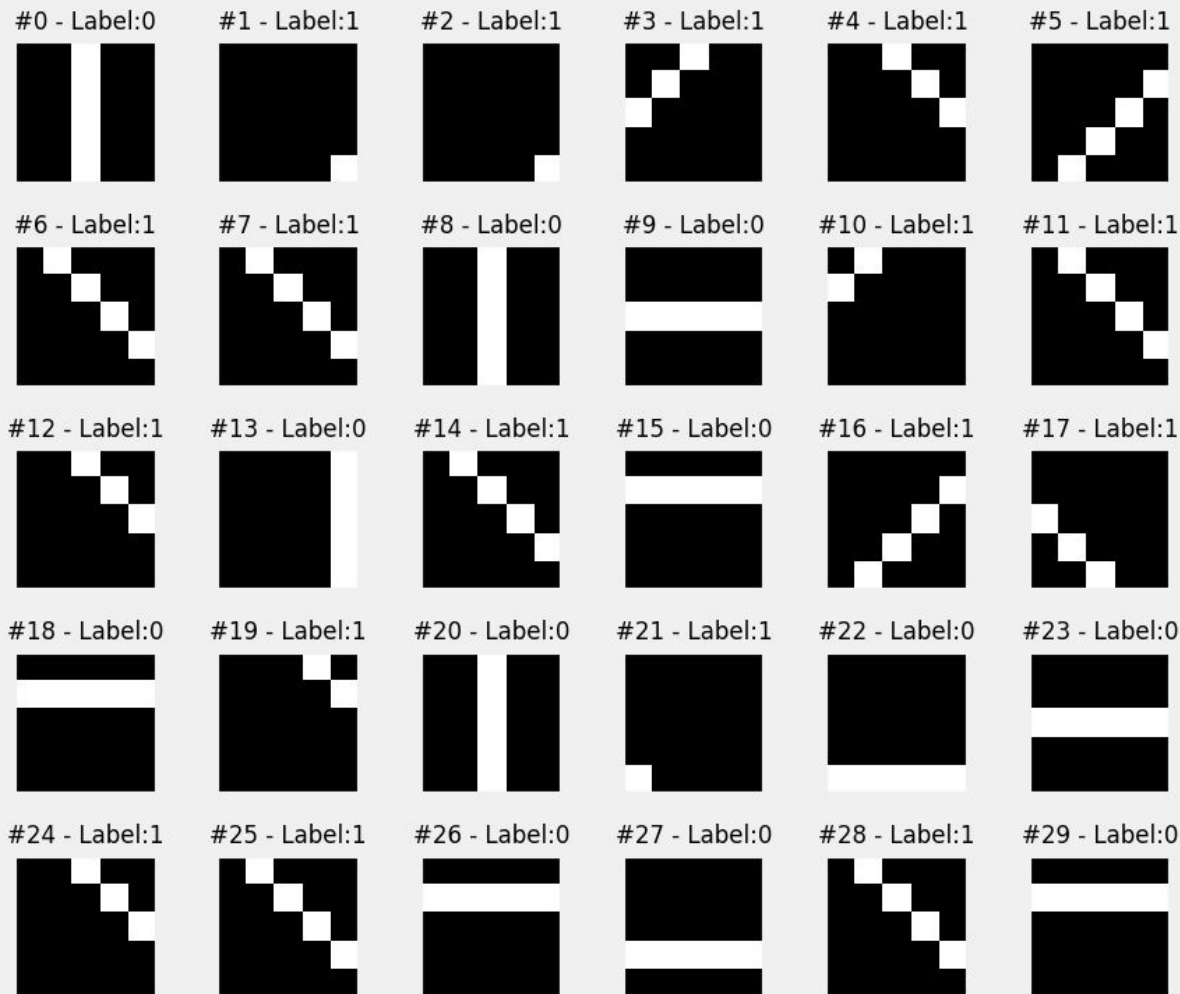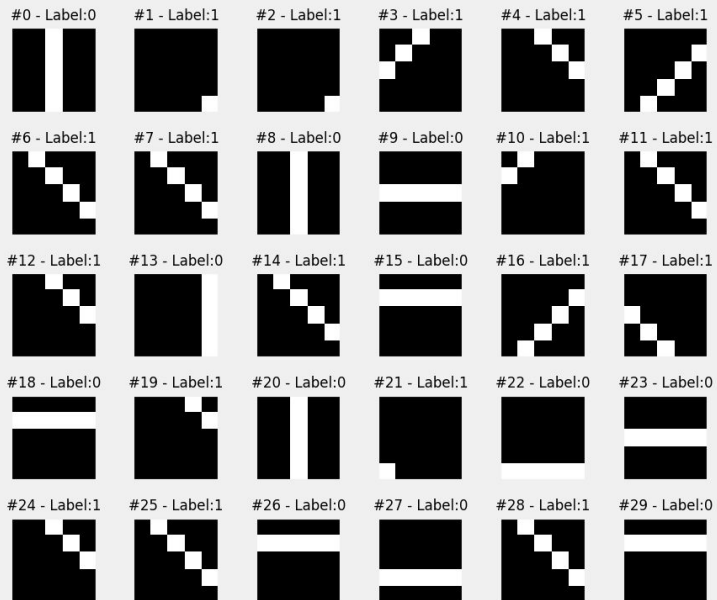
```
images[0]
array([[[  0,   0, 255,   0,   0],
        [  0,   0, 255,   0,   0],
        [  0,   0, 255,   0,   0],
        [  0,   0, 255,   0,   0],
        [  0,   0, 255,   0,   0]]], dtype=uint8)


labels[0]
0
```

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

Images and Channels

image_r	image_g	image_b	image_gray
[[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]]	[[　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]]	[[　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]]	[[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]]

Single Channel (grayscale)

Red	Green	Blue	Grayscale Image
5x5	5x5	5x5	5x5 = 0.21R + 0.72G + 0.07B

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]]
```
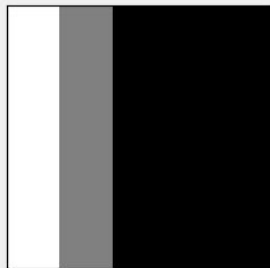
image_b

```
[[  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]]
```
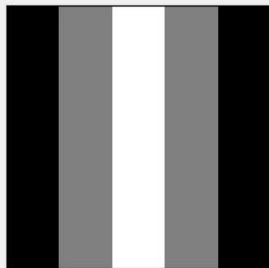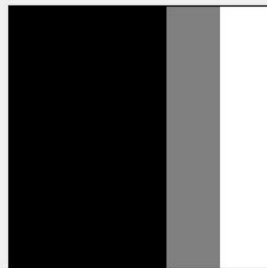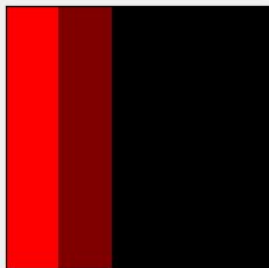
Three Channels (color)

as first channel

5x5x3 - R only

as second channel
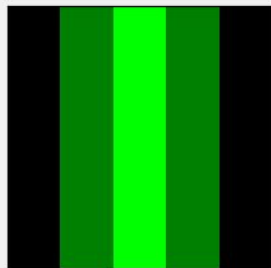
5x5x3 - G only

as third channel
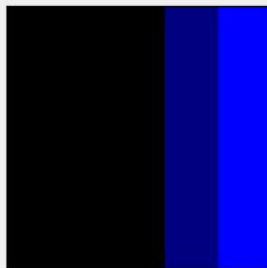
5x5x3 - B only

RGB Image
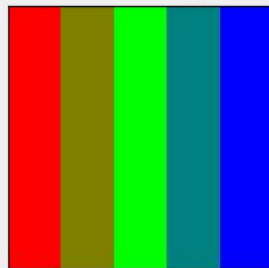
5x5x3 = (R, G, B) stacked

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.

vision Public

Watch 432 Fork 6.9k Star 16.1k

main

141 Branches 162 Tags

Go to file

Add file Code

NicolasHug remove printing info in datasets (#8683) ✕ 945bdad · last week 3,949 Commits

.github	Remove viable strict workflow (#8676)	2 weeks ago
android	CXX_STANDARD 14 -> 17 (#7608)	last year
benchmarks	GPU jpeg decoder: add batch support and hardware deco…	2 months ago
cmake	Remove include Python.h (#8413)	5 months ago
docs	Revamp decoding docs (#8633)	3 weeks ago
examples	Rename cpp example README.rst to README.md (#8467)	4 months ago
gallery	Allow decode_image to support paths (#8624)	last month
ios	CXX_STANDARD 14 -> 17 (#7608)	last year
packaging	Enable xpu windows CD (#8656)	3 weeks ago
references	Correcting a link bracket in Classification README.md (#…	3 months ago
scripts	Update classify_prs notebook (#8383)	6 months ago
test	Expose `transforms.v2` utils for writing custom transforms…	3 weeks ago
torchvision	remove printing info in datasets (#8683)	last week
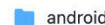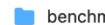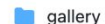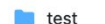
About

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

☆ 16.1k stars

👁 432 watching

⑂ 6.9k forks

Report repository

Releases 45

🏷 Torchvision 0.20 release (Latest)
5 days ago

+ 44 releases

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

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

$$\text{input} = 0 \implies \frac{0 - \text{mean}}{\text{std}} = \frac{0 - 0.5}{0.5} = -1$$

$$\text{input} = 1 \implies \frac{1 - \text{mean}}{\text{std}} = \frac{1 - 0.5}{0.5} = 1$$

Convert, build and transform tensors into useful format

Data Preparation

Splitting the dataset: build a custom splitter

```python
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)
```

```python
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, 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

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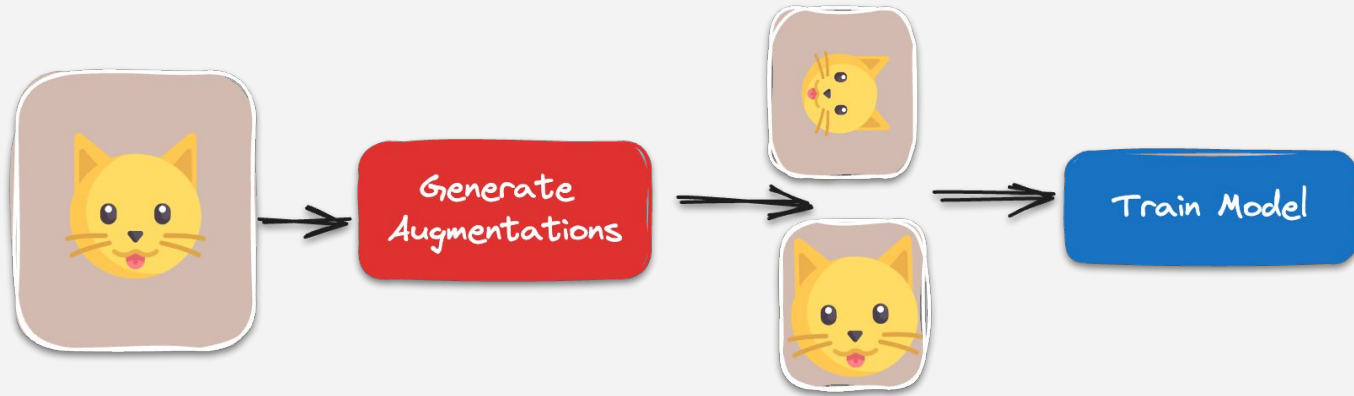
In general, we want to apply data augmentation to the **training data only**.

Splitting the dataset: all you need is data augmentation
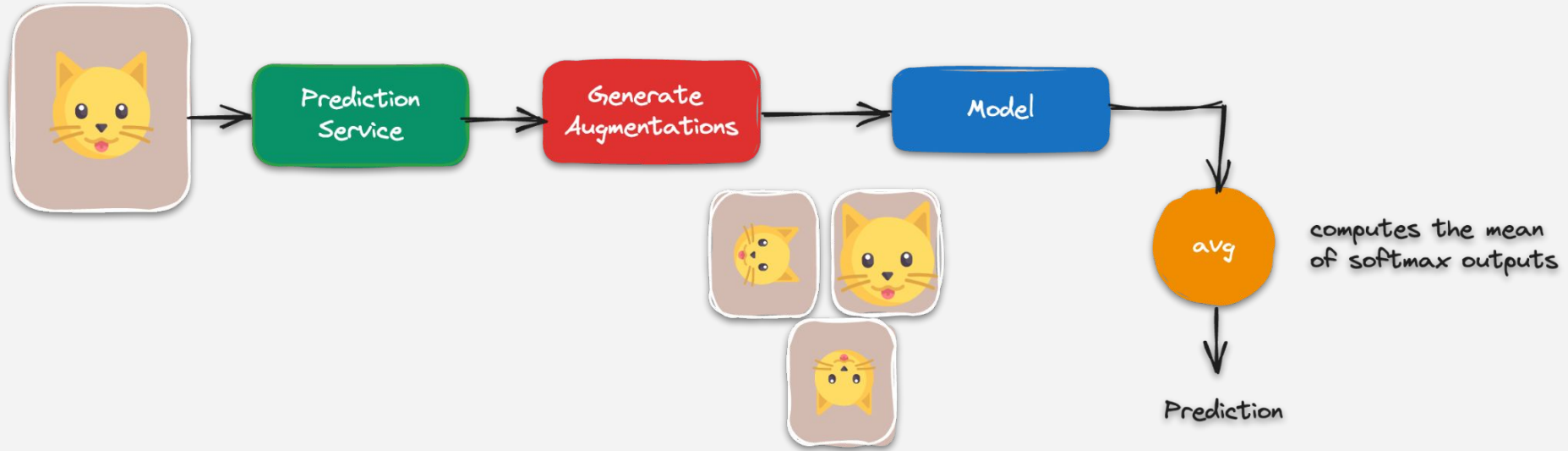
(...) there is also "test-time augmentation", which can be used to improve the performance of a model after it is deployed.

Splitting the dataset: all you need is data augmentation

```python
class TransformedTensorDataset(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)
```

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

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

```python
train_dataset = TransformedTensorDataset(x_train_tensor,
                                         y_train_tensor,
                                         transform=train_composer)
val_dataset = TransformedTensorDataset(x_val_tensor,
                                       y_val_tensor,
                                       transform=val_composer)
```

Splitting the dataset: putting it together

```python
# Builds tensors from numpy arrays BEFORE split
# Modifies the scale of pixel values from [0, 255] to [0, 1]
x_tensor = torch.as_tensor(images / 255).float()
y_tensor = torch.as_tensor(labels.reshape(-1, 1)).float()

# Uses index_splitter to generate indices for training and
# validation sets
train_idx, val_idx = index_splitter(len(x_tensor), [80, 20])

# Uses indices to perform the split
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]

# Builds different composers because of data augmentation on training set
train_composer = Compose([RandomHorizontalFlip(p=.5),
                          Normalize(mean=(.5,), std=(.5,))])
val_composer = Compose([Normalize(mean=(.5,), std=(.5,))])

# Uses custom dataset to apply composed transforms to each set
train_dataset = TransformedTensorDataset(x_train_tensor,
                                         y_train_tensor,
                                         transform=train_composer)
val_dataset = TransformedTensorDataset(x_val_tensor,
                                       y_val_tensor,
                                       transform=val_composer)
```

Splitting the dataset: putting it together

```python
# 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

Splitting the dataset: pixels as features

```python
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
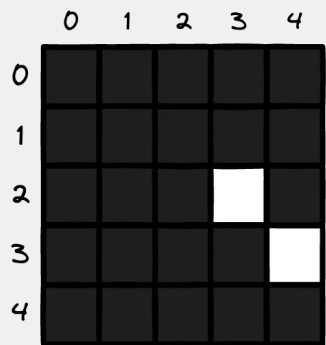
Original Image

Flatted Image

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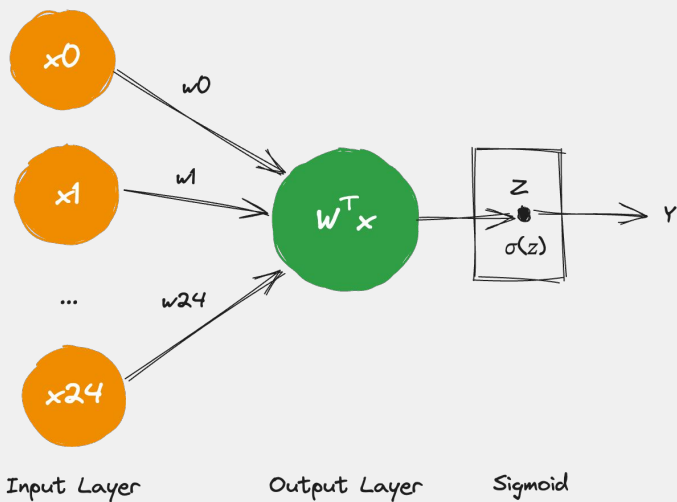
$$W = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{24} \end{bmatrix}_{(25 \times 1)}; X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{24} \end{bmatrix}_{(25 \times 1)}$$

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

$$= w_0 x_0 + w_1 x_1 + \cdots + w_{24} x_{24}$$

Input Layer   Output Layer   Sigmoid

```python
# 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()
```

```python
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)
```
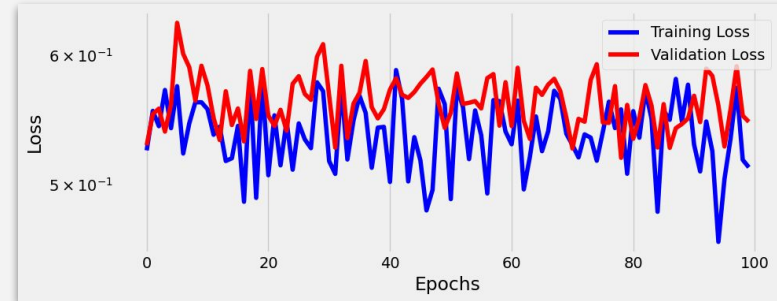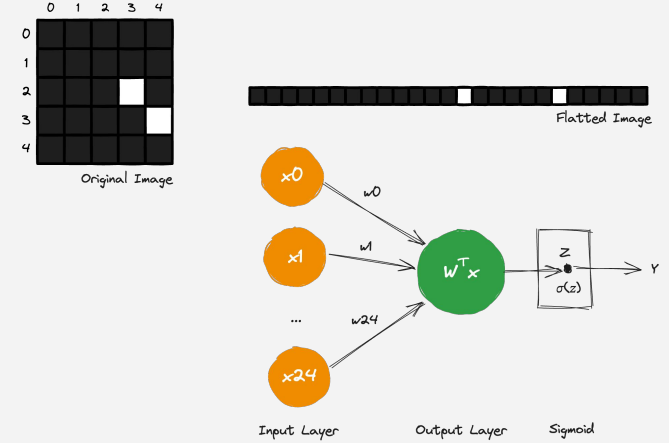
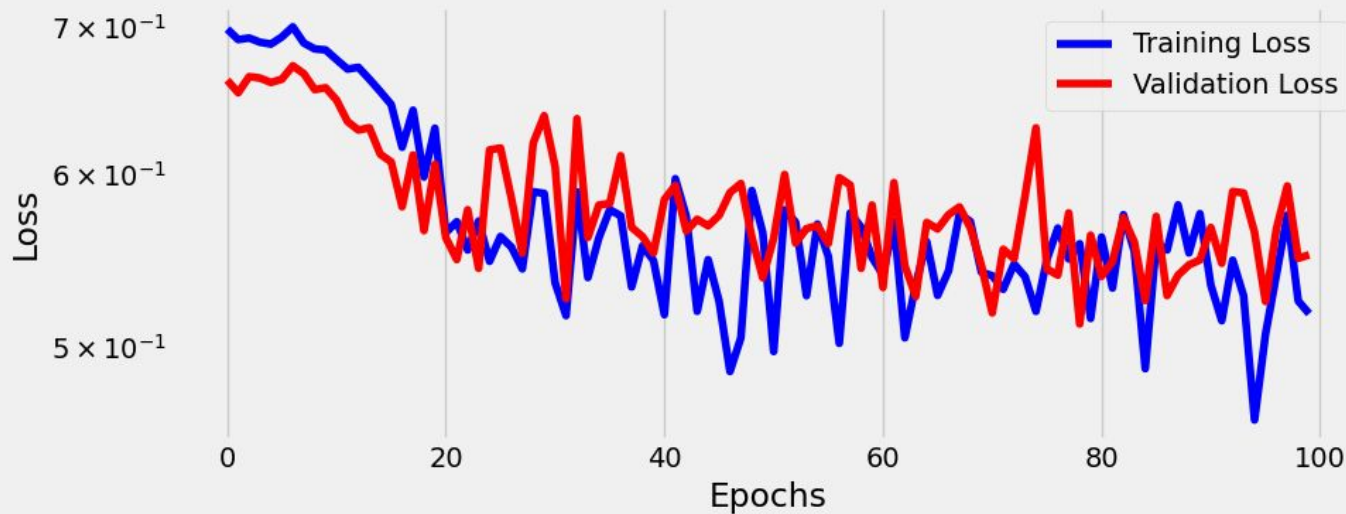*There we go, let's add not one, but two hidden layers*

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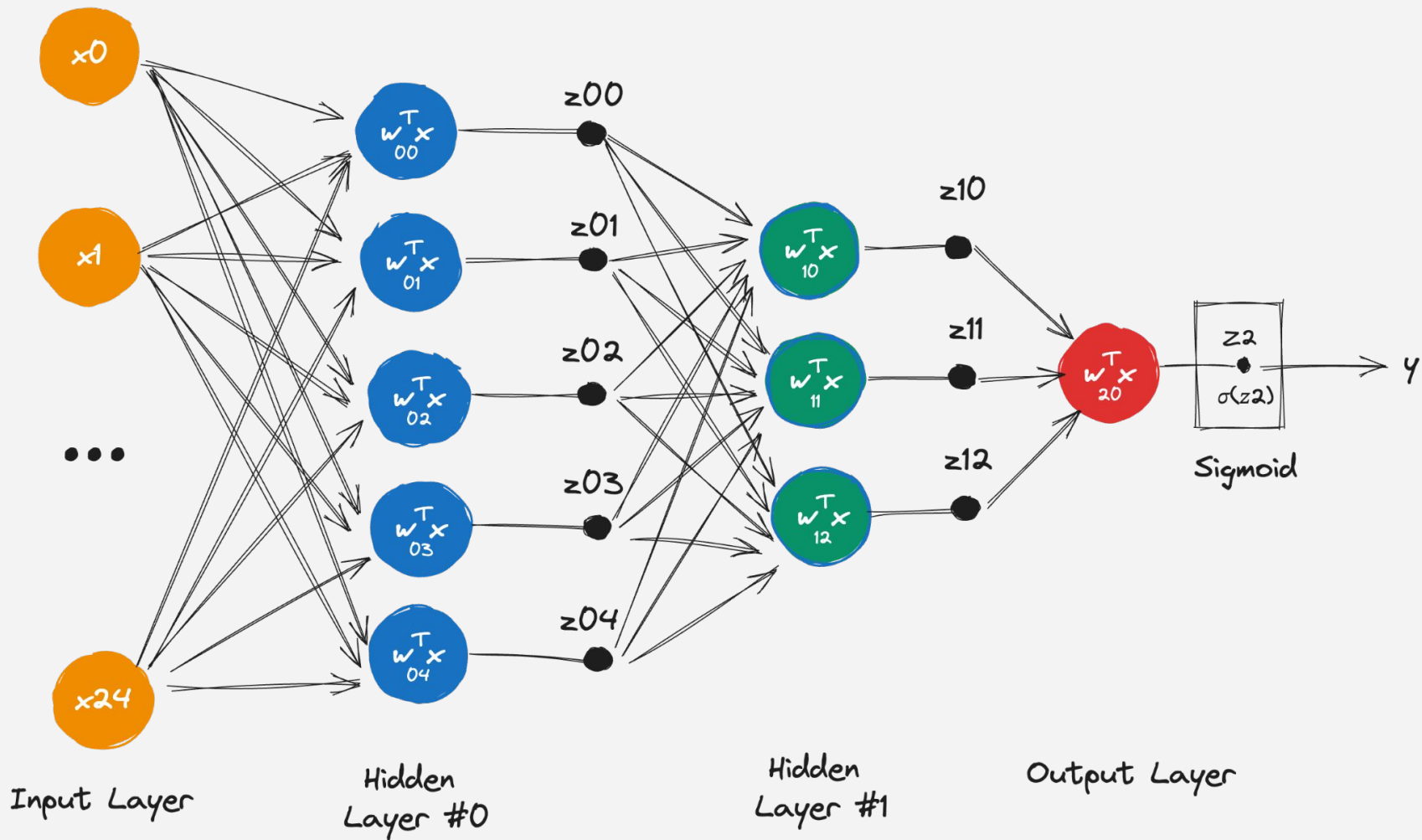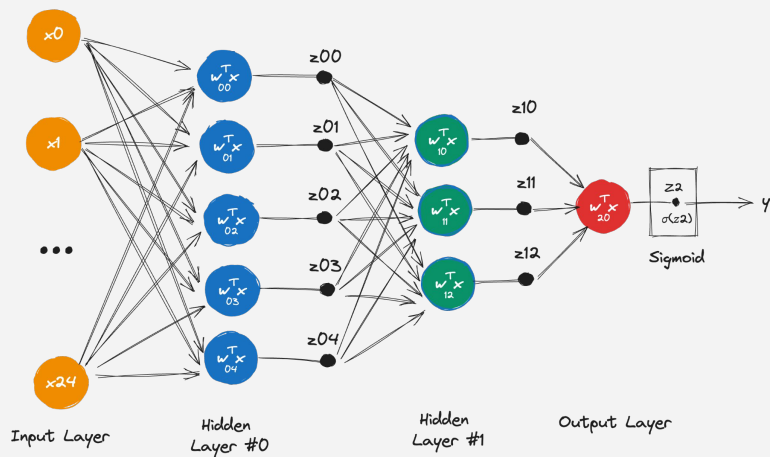
```python
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

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()
```
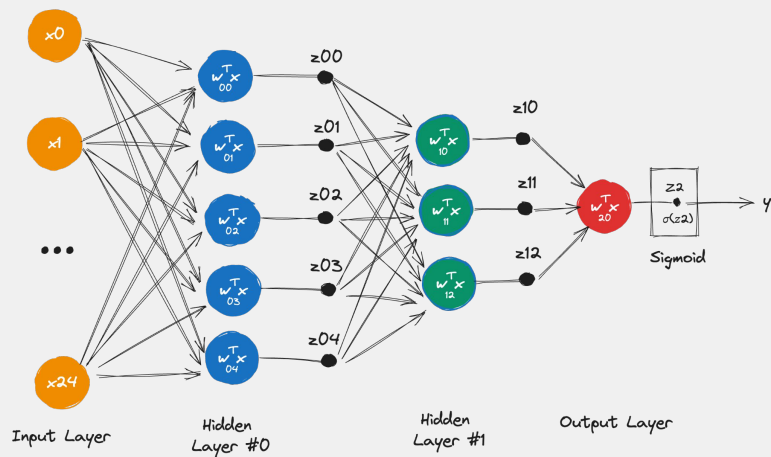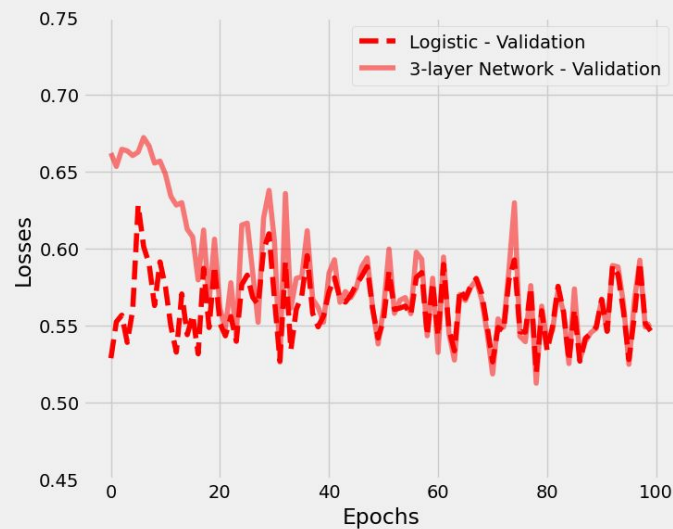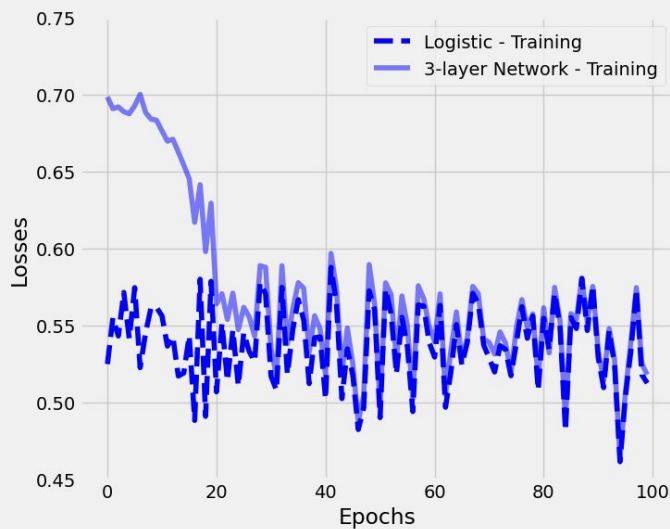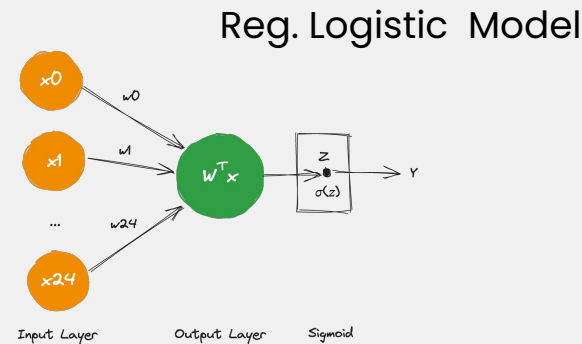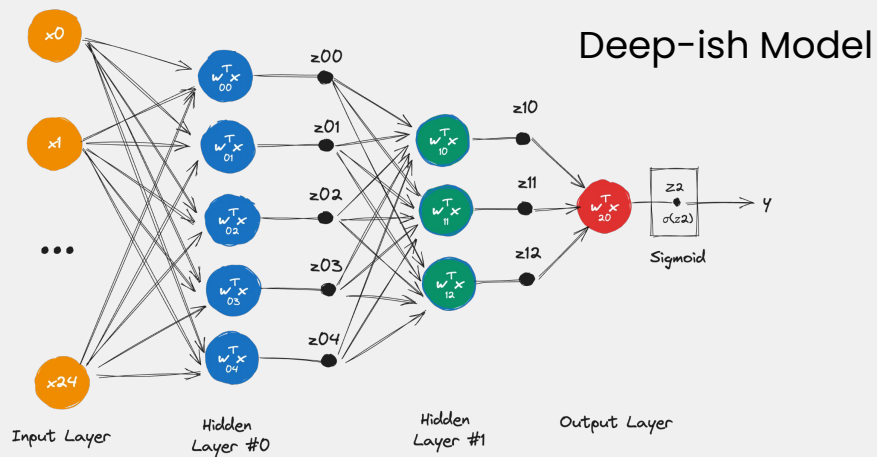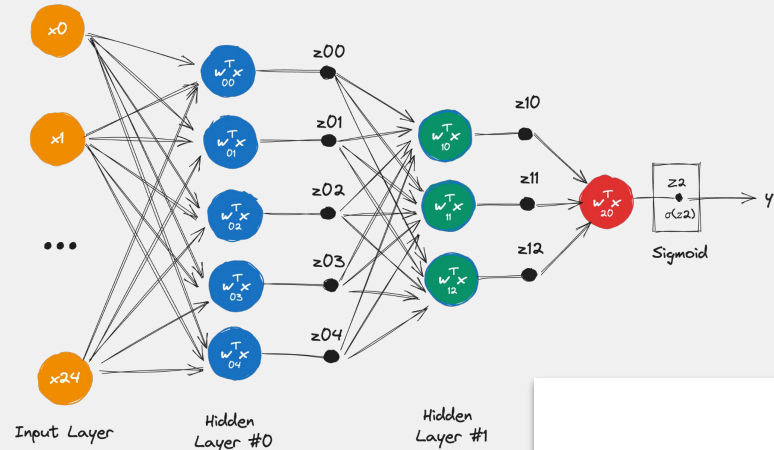
```
n_epochs = 100

arch_nn = Architecture(model_nn, binary_loss_fn, optimizer_nn)
arch_nn.set_loaders(train_loader, val_loader)
arch_nn.train(n_epochs)
```
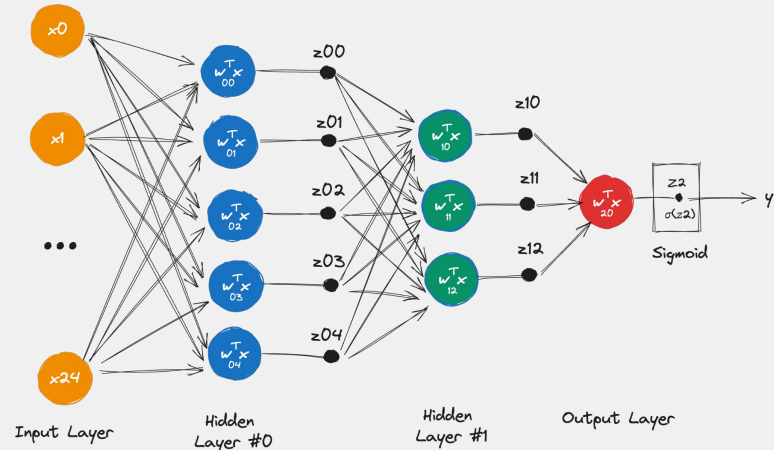
Hidden #0

$$
\begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}_{(5\times1)} = \begin{bmatrix} - & w_{00}^T & - \\ - & w_{01}^T & - \\ - & w_{02}^T & - \\ - & w_{03}^T & - \\ - & w_{04}^T & - \end{bmatrix}_{(5\times25)} \begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}_{(25\times1)}
$$

Hidden #1

$$
\begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}_{(3\times1)} = \begin{bmatrix} - & w_{10}^T & - \\ - & w_{11}^T & - \\ - & w_{12}^T & - \end{bmatrix}_{(3\times5)} \begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}_{(5\times1)}
$$

Output

$$
\begin{bmatrix} z_2 \end{bmatrix}_{(1\times1)} = \begin{bmatrix} - & w_{20}^T & - \end{bmatrix}_{(1\times3)} \begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}_{(3\times1)}
$$

x0
x1
x24

Input Layer

Hidden Layer #0

Hidden Layer #1

Output Layer

z00
z01
z02
z03
z04

z10
z11
z12

z2
$\sigma(z2)$

y

Sigmoid

Hidden #0

$$\begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}_{(5\times1)} = \begin{bmatrix} - & w_{00}^T & - \\ - & w_{01}^T & - \\ - & w_{02}^T & - \\ - & w_{03}^T & - \\ - & w_{04}^T & - \end{bmatrix}_{(5\times25)} \begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}_{(25\times1)}$$

Hidden #1

$$\begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}_{(3\times1)} = \begin{bmatrix} - & w_{10}^T & - \\ - & w_{11}^T & - \\ - & w_{12}^T & - \end{bmatrix}_{(3\times5)} \begin{bmatrix} z_{00} \\ z_{01} \\ z_{02} \\ z_{03} \\ z_{04} \end{bmatrix}_{(5\times1)}$$

Output

$$\begin{bmatrix} z_2 \end{bmatrix}_{(1\times1)} = \begin{bmatrix} - & w_{20}^T & - \end{bmatrix}_{(1\times3)} \begin{bmatrix} z_{10} \\ z_{11} \\ z_{12} \end{bmatrix}_{(3\times1)}$$

This is just the reg. logistic

x0
x1
...
x24

w0
w1
w24

Input Layer

Output Layer

$w^T x$

z
$\sigma(z)$

Y

Sigmoid

substituting $z's\ldots$

$$\begin{bmatrix} z_2 \end{bmatrix}_{(1\times1)} = \underbrace{\begin{bmatrix} - & w_{20}^T & - \end{bmatrix}_{(1\times3)}}_{\text{Output Layer}} \underbrace{\begin{bmatrix} - & w_{10}^T & - \\ - & w_{11}^T & - \\ - & w_{12}^T & - \end{bmatrix}_{(3\times5)}}_{\text{Hidden Layer \#1}} \underbrace{\begin{bmatrix} - & w_{00}^T & - \\ - & w_{01}^T & - \\ - & w_{02}^T & - \\ - & w_{03}^T & - \\ - & w_{0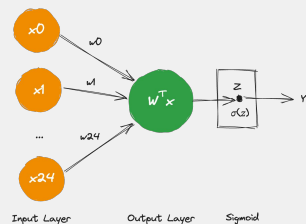4}^T & - \end{bmatrix}_{(5\times25)}}_{\text{Hidden Layer \#0}} \begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}_{(25\times1)}$$

multiplying...

$$= \underbrace{\begin{bmatrix} - & w^T & - \end{bmatrix}_{(1\times25)}}_{\text{Matrices Multiplied}} \begin{bmatrix} x_0 \\ \vdots \\ x_{11} \\ \vdots \\ x_{24} \end{bmatrix}_{(25\times1)}$$
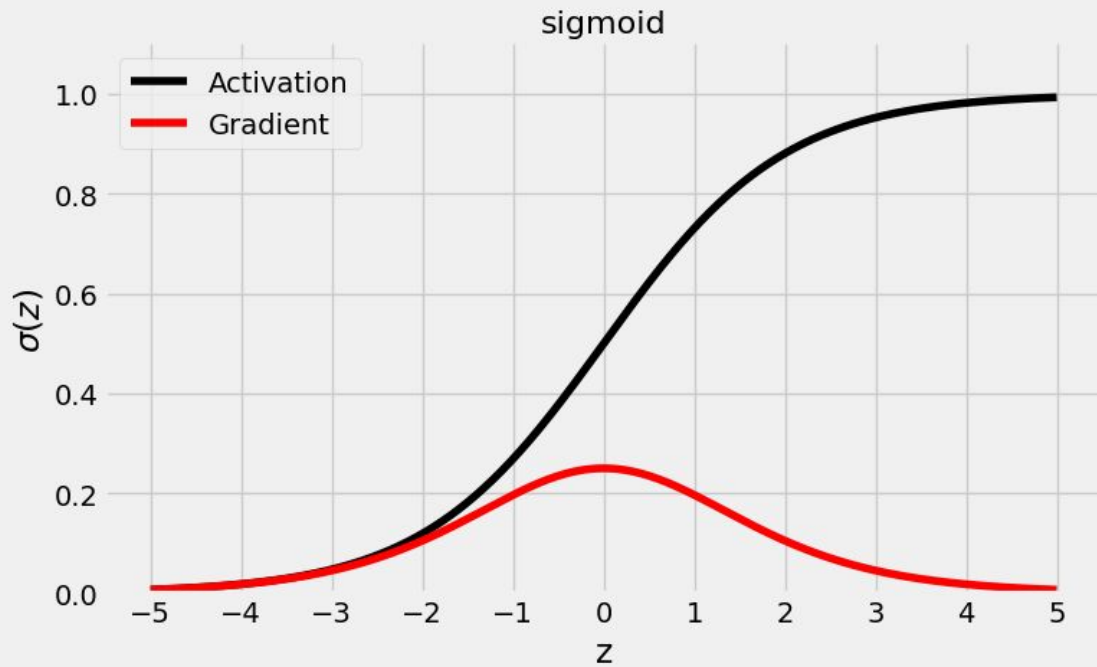
Nonlinear functions either squash or bend straight lines

Activation Functions

sigmoid

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

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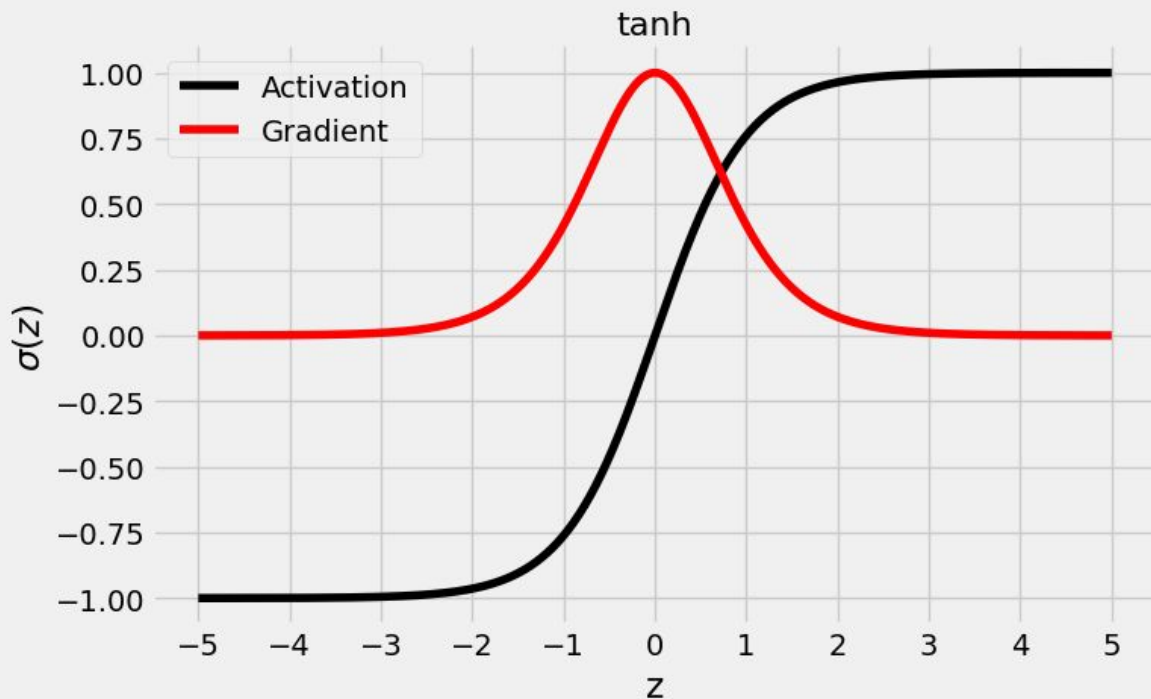
$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- 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 it's 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**.
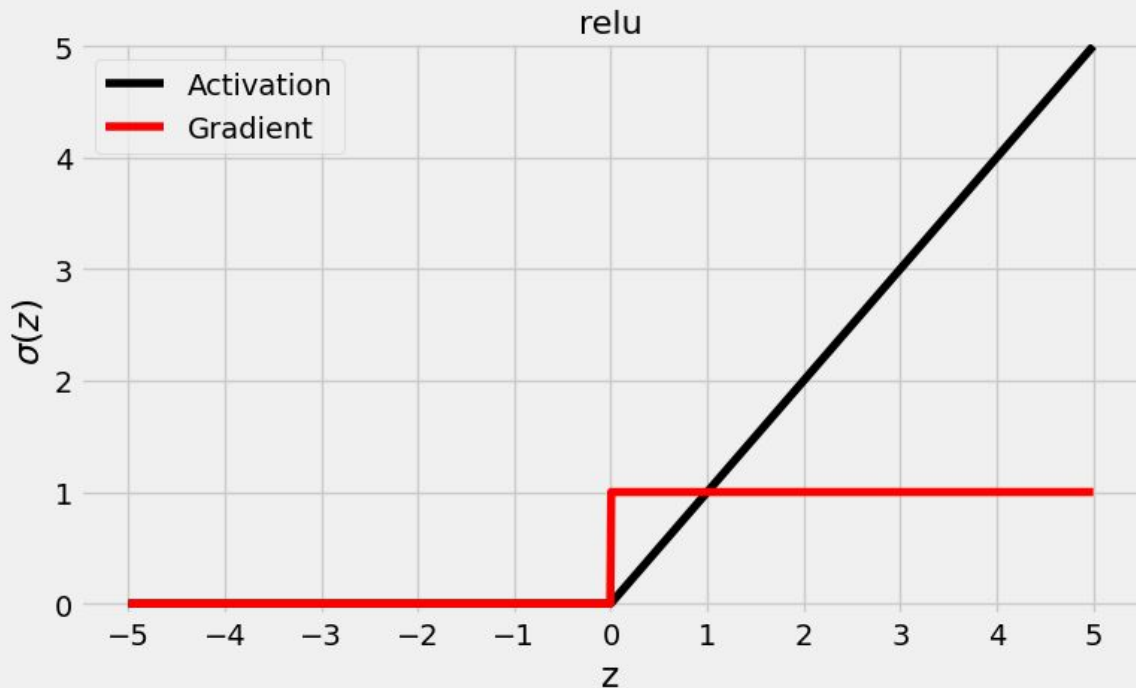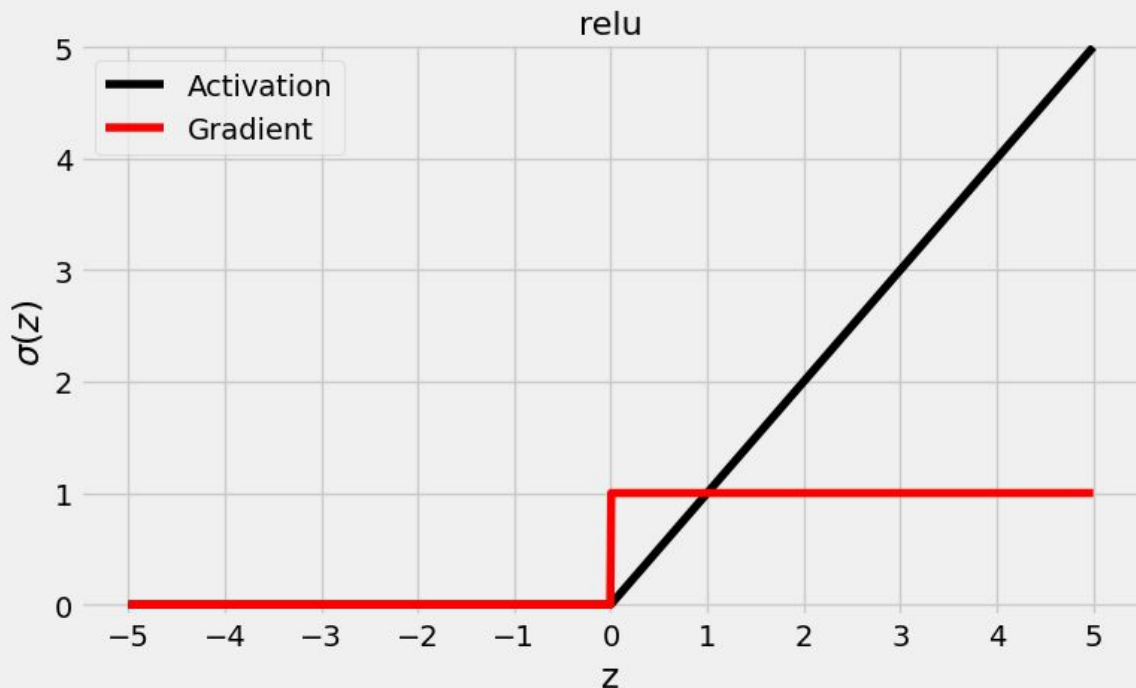
relu

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

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**.

relu

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

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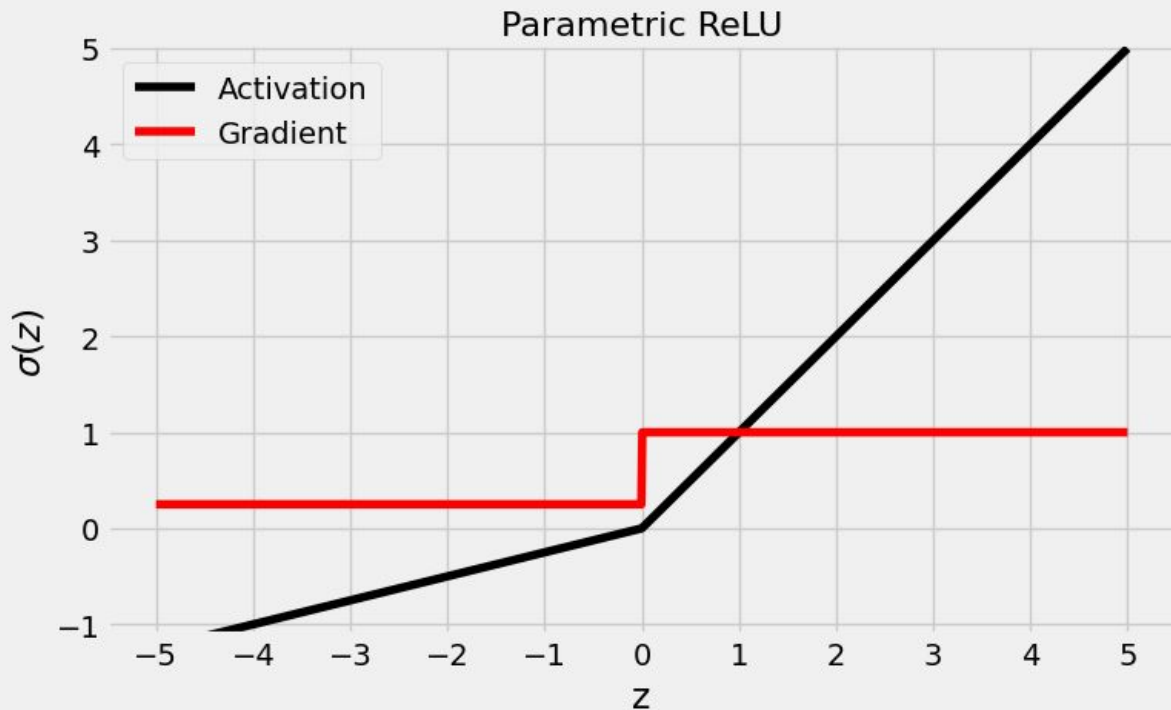
Leaky ReLU
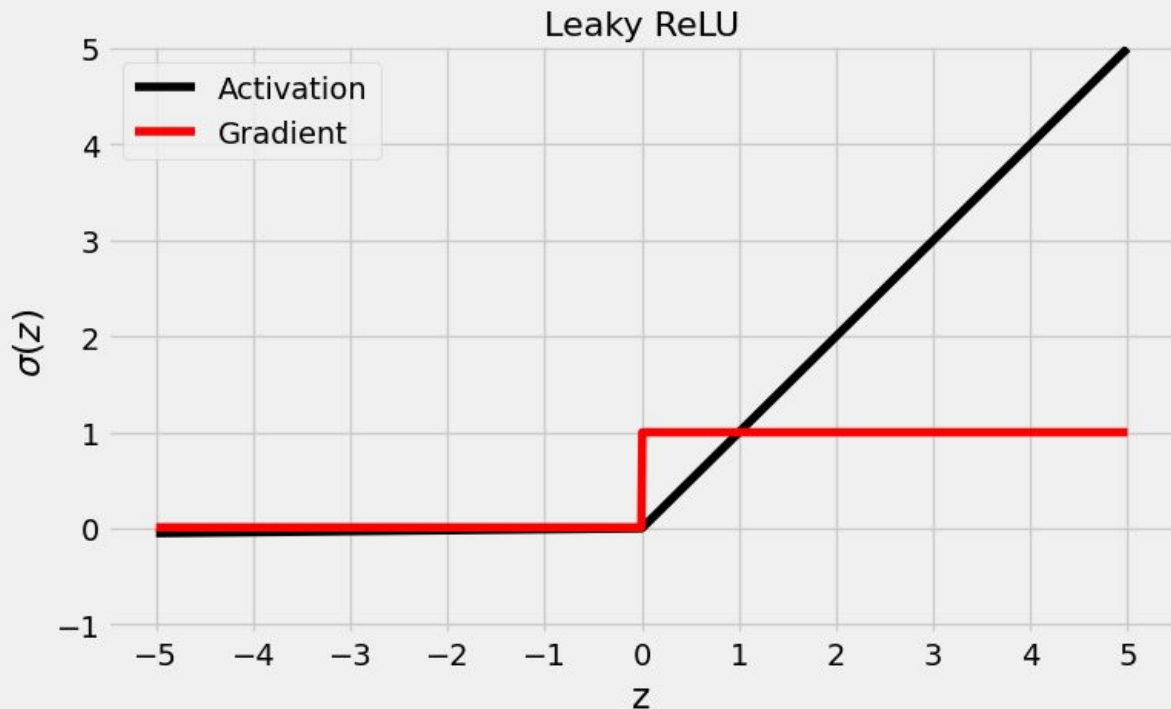
$$\sigma(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0.01z, & \text{if } z < 0 \end{cases}$$

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**.
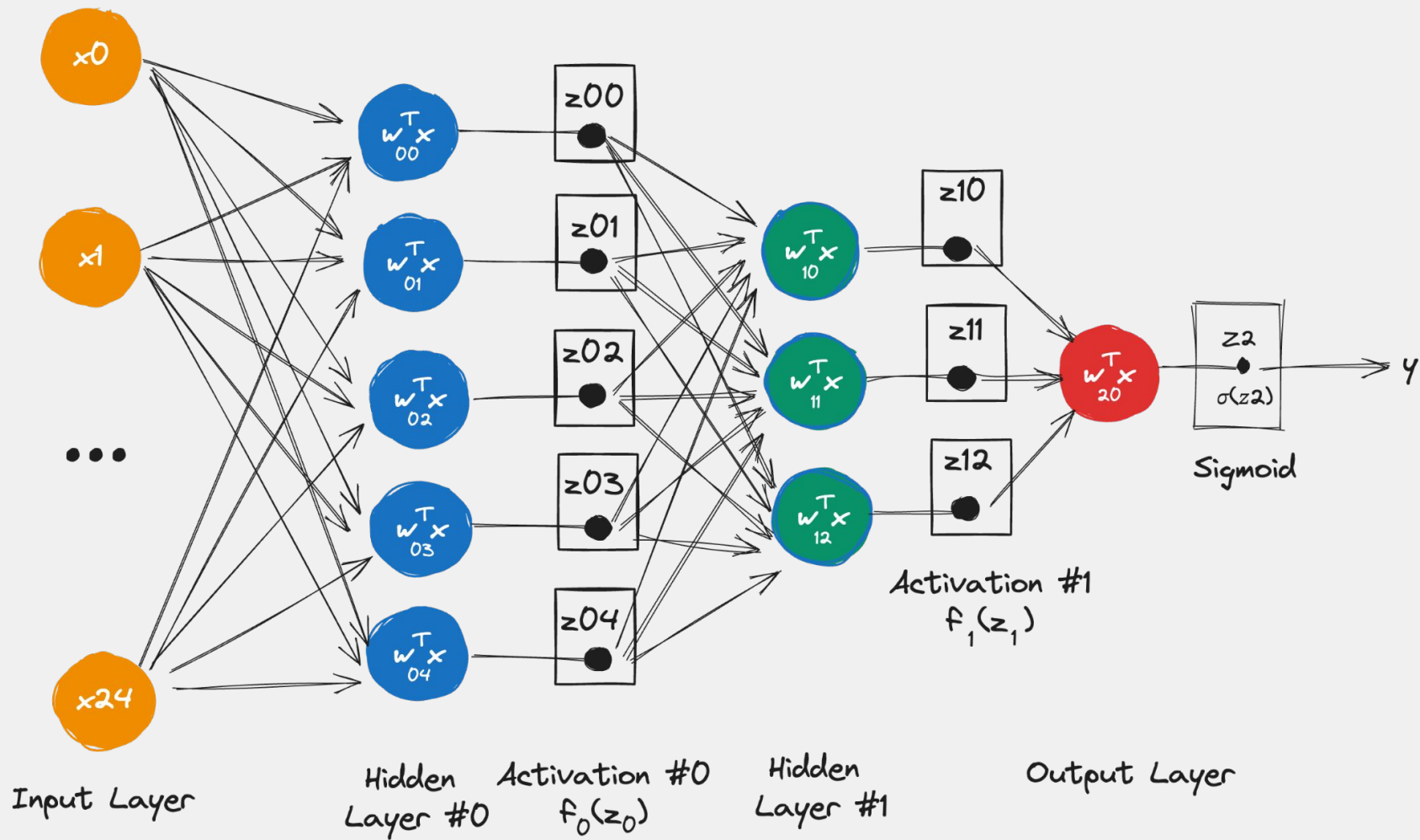
Parametric ReLU

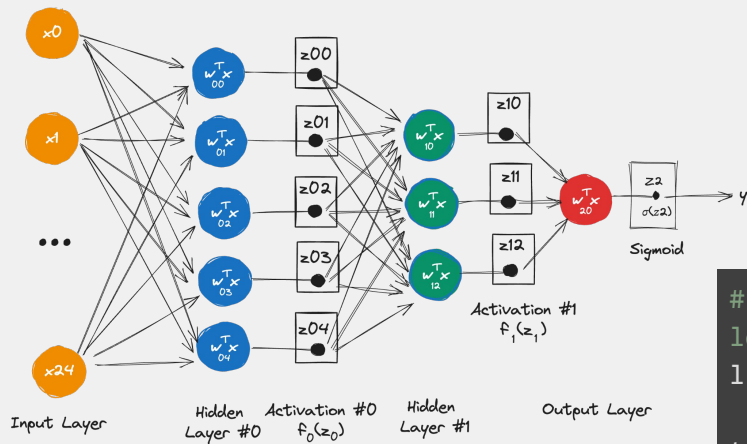$$\sigma(z) = \begin{cases} z, & \text{if } z \geq 0 \\ az, & \text{if } z < 0 \end{cases}$$

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.
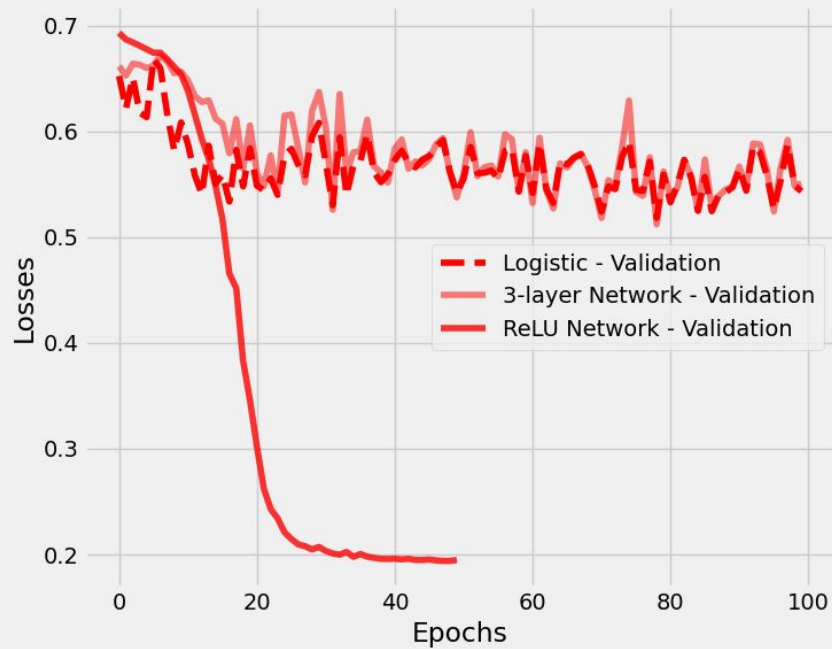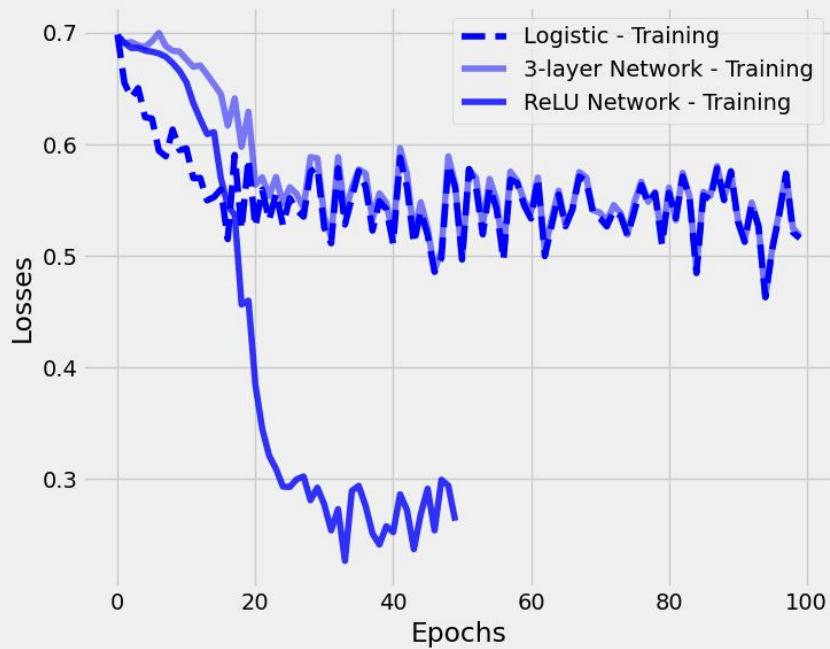
Deep Model

Deep Model

```python
# 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