Homework 5

In [9]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
```

This homework is designed to help build up some of the pieces you will use going forward when making neural networks.

Let's imagine we are building a classifier for images. These images will be given to us as grayscale 3x3 images. (These would obviously be VERY simple images, this is just for tutorial reasons.) These images can either be bears 🐻, dogs 🐕, cats 🐈, or fish 🐟.

First a few questions to think about:

1. How many inputs does our neural network have?
2. How should we change our input to make it easier to feed to the neural network?
3. How many outputs does our neural network have?
4. If our neural network does not use hidden layers, what would the dimensions of the array that represent our transformation from input to output be?
5. What does this say about the dimensions of the arrays that make up all of our stages?

Task 1: Convert the 3x3 tensor into a 1x9 tensor which will be a better input for our neural network. First do this with the individual tensor (in1), then try to do it with the list of tensors (our "training data"). Then append the flattened in1 to the flattened training data (making it the 100th piece of data).

Hint: consider using one of the following:

- torch.flatten
- torch.reshape
- torch.cat

In [10]:

```
#3x3 tensor

np.random.seed(0)
in1_numpy = np.random.randint(256, size=(3,3))
in1 = torch.tensor(in1_numpy, dtype=torch.float)

in99_numpy = np.random.randint(256, size=(99, 3, 3))
in99 = torch.tensor(in99_numpy, dtype=torch.float)

picture_labels_numpy = np.random.randint(4, size=(100))
picture_labels = torch.tensor(picture_labels_numpy, dtype=torch.float)

########################
## YOUR CODE STARTS HERE
########################

# Task 1: Flatten in1 to be a (1x9) tensor, then flatten training_data to a (99x9) tenso
in1_flattened = in1.reshape([1,9])
training_data_flattened = in99.reshape([99, 9])

final_data = torch.cat((in1_flattened, training_data_flattened), 0)
```

```
########################
## YOUR CODE ENDS HERE
########################
```

🚆 Task 2: Make a simple neural network with one hidden layer for your data. The hidden layer should have 5 neurons on it. You can store your weights in two matrices, and initialize them to 1s or random numbers. Send their input through the hidden and output layers to get your final output.

Things to consider:

1. Why do we use matrices to store our weights?
2. What will the dimensions of our matrices be?

Hints:

1. You can use torch.rand((R,C)) to initialize a matrix of random numbers.
2. For simplicity, you do not need to consider biases for this problem, just think about weights.
3. PyTorch has a matrix multiplication function called torch.matmul(m1, m2).
4. You can use the ReLU activation function by calling F.relu(h1).

In [11]:

```
########################
## YOUR CODE STARTS HERE
########################
weights1 = torch.rand((9,5))
weights2 = torch.rand((5, 4))

h1 = torch.matmul(final_data, weights1)  # Matrix multiplication between input data and
h1_activated = F.relu(h1)  # Activation function (ReLU) applied to the hidden layer's ou

# Hidden layer to output
output = torch.matmul(h1_activated, weights2)  # Matrix multiplication


########################
## YOUR CODE ENDS HERE
########################
```

Your output should now be four numbers, each of which are unbounded. We would prefer to have four bounded numbers, so we must apply the sigmoid function to each one.

🧮 Task 3: Use the sigmoid function to make all elements of your output between zero and one. Use .shape to ensure that your output is the expected size.

Note: the torch.sigmoid() function is actualy an alias of the torch.special.expit() function, so try searching for both if one doesn't work.

In [12]:

```
########################
## YOUR CODE STARTS HERE
########################
sigmoid_output = torch.sigmoid(output)
print(sigmoid_output.shape)

########################
## YOUR CODE ENDS HERE
########################
```

```
torch.Size([100, 4])
```

Now we come across a common, but ultimately easy-to-solve problem in neural networks: how we should represent the output. Currently our output is four numbers between zero and one. We can convert these to all zero and a single one value if we want to produce an actual guess. This is called a one-hot encoding. However, as defined in the first code block, our actual picture label is only a single value for each image (thus a tensor of 100 values for the 100 images).This is called an integer encoding.

📭 Task 4: Use a pytorch function (or make your own) that converts from integer encoding to one hot encoding to convert the picture labels to one hot encoding. Then compute the log-loss of your estimated output and the actual output.

Hint: the following may be helpful: \

- torch.nn.functional.one_hot
- torch.nn.functional.nll_loss

In [ ]:

```
########################
## YOUR CODE STARTS HERE
########################
# Convert integer labels to one-hot encoding
# Assuming the maximum label value is less than 4 since the network outputs 4 values
num_classes = 4  # Number of output classes
one_hot_labels = F.one_hot(picture_labels.to(torch.int64), num_classes=num_classes)

# Convert network outputs to probabilities
probabilities = F.softmax(sigmoid_output, dim=1)

# Compute log probabilities
log_probabilities = torch.log(probabilities + 1e-5)  # Adding a small value to prevent l

# Compute the log-loss
# nll_loss expects inputs in log form; however, it is designed for use with log_softmax,
# Convert one-hot labels to probabilities by multiplying with log_probabilities and summ
loss = -torch.sum(one_hot_labels * log_probabilities) / one_hot_labels.shape[0]
########################
## YOUR CODE ENDS HERE
########################
```

In [ ]:

```
'''
Implemented a simple neural network in PyTorch for processing 100 images. The network ha

Steps
Preprocessing: Flattened and concatenated image tensors to create a 100x9 input tensor.
Network Construction: Initialized weights for connections between layers. Used ReLU for
Output Processing: Applied softmax to convert network outputs to probabilities.
One-Hot Encoding and Loss: Converted integer labels to one-hot encoding and computed log
Conclusion
This process involved data preparation, network setup, prediction adjustment, and accura
'''
```