# tensors

## Creating a tensor

- `torch.tensor(data)`
- `torch.from_numpy(np_array)`
- `torch.ones_like(x_data)`
- `torch.rand(shape)`
- `torch.ones(shape)`
- `torch.zeros(shape)`

</hackerspace>

# tensors

## Attributes of a tensor

- `tensor.shape : Returns shape of tensor`
- `tensor.dtype : Datatype of tensor`
- `tensor.device : Device (CPU/GPU) tensor is stored on`

</hackerspace>

# tensors

## Operations on Tensor

- `Matrix multiplication`
  - `t1 @ t2`
  - `t1.matmul(t2)`
- `Addition/ Subtraction is straightforward`
  - `t1 + t2`
  - `t1 - t2`

</hackerspace>

# tensors

## Operations on Tensor

- In-place operations
    - Add a suffix ‘_’, basically shorthand for t1 = t1 + 2
    - t1.add_(2)
- Useful : Converting tensor to python item
    - If t1 is a single element tensor
    - a = t1.item()

# back propagation

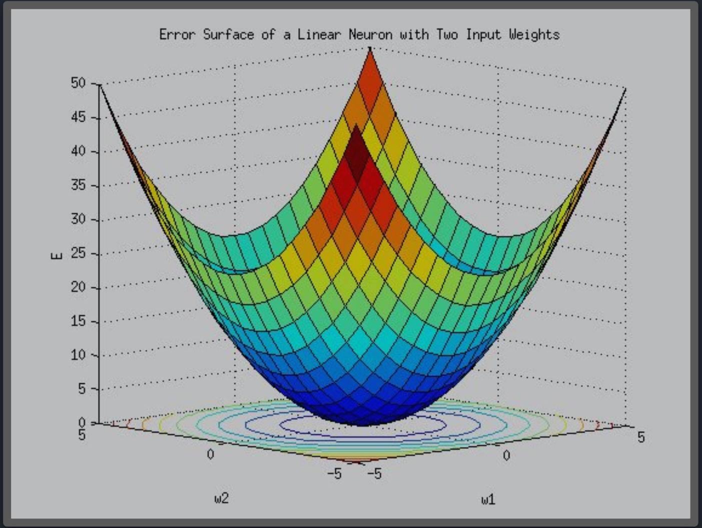
## Conceptual

# back propagation

## Loss functions

- MSE : ( Y - Y~)$^2$

- Cross entropy
- Hingle loss
- Absolute error ...




</hackerspace>

# back propagation

## Partial derivatives - A quick glance at the equations

### Finding gradient

$$\frac{\delta w}{\delta y} = \frac{\delta w}{\delta z} * \frac{\delta z}{\delta y}$$

$$\frac{\delta b}{\delta y} = \frac{\delta b}{\delta z} * \frac{\delta z}{\delta y}$$

### Updating param

$$w = w - \eta(\frac{\delta w}{\delta y})$$

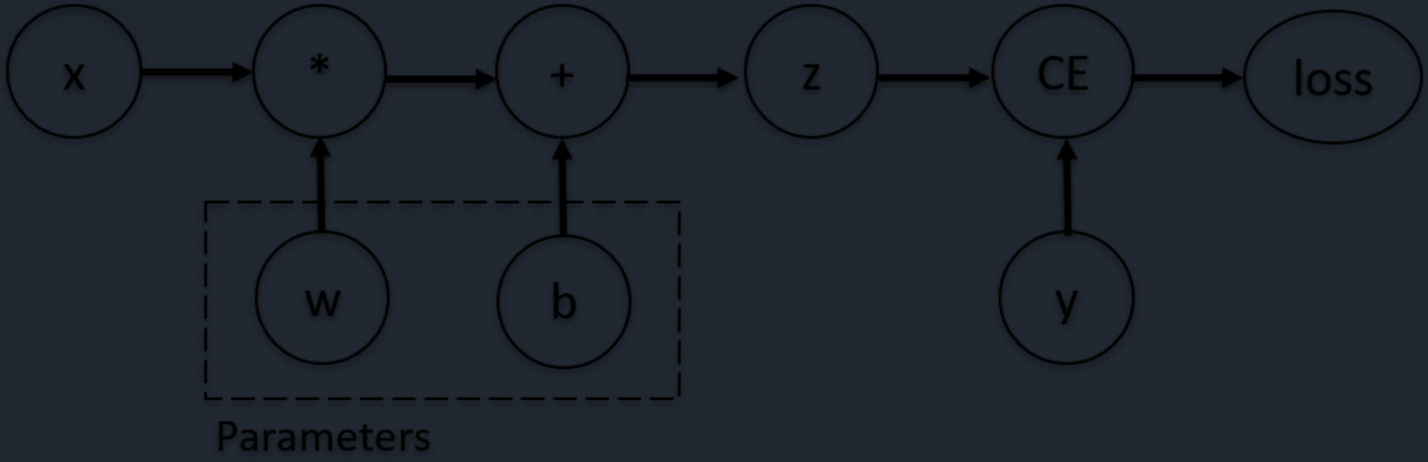
$$b = b - \eta(\frac{\delta b}{\delta y})$$

z = x * w + b

y = activation_function ( z )

</hackerspace>

# autograd




z = x * w + b

y = activation_function ( z )

</hackerspace>

# autograd

For optimizing

1. Set loss function
   - `loss_fn = torch.nn.MSELoss()`
2. Get Loss
   - `loss = loss_fn(out, exp_out)`
3. Get the gradients (partial derv)
   - `loss.backwards()`

$$\frac{\delta w}{\delta y} = \frac{\delta w}{\delta z} * \frac{\delta z}{\delta y}$$
$$\frac{\delta b}{\delta y} = \frac{\delta b}{\delta z} * \frac{\delta z}{\delta y}$$

4. Update the weights and biases via the optimizer
   - `optimizer = torch.optim.SGD(model.parameters(), lr=0.05)`
   - `optimizer.step()`

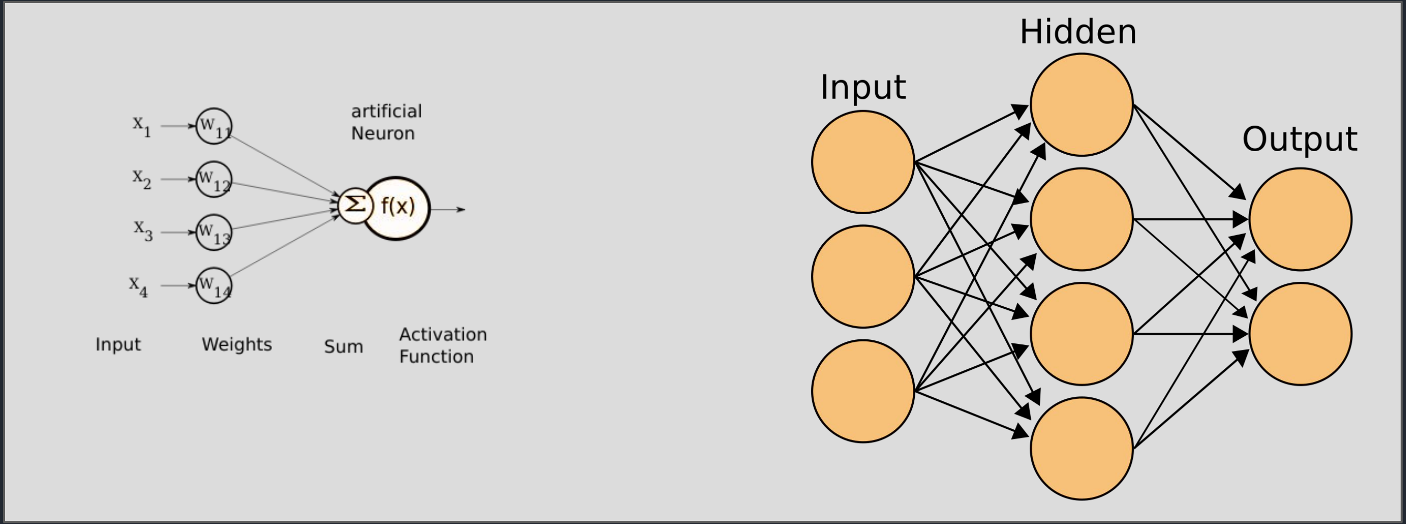
$$w = w - \eta(\frac{\delta w}{\delta y})$$
$$b = b - \eta(\frac{\delta b}{\delta y})$$

Note:
optimizer.zero_grad()
Is used to zero the gradients.

</hackerspace>

# neural networks

- Inspired by how the brain function
- Basic building block is the Neuron
- Neurons are connected together to form a Neural network



# neural networks

1. Get Data
    a. Divide to test and train
2. Create a Neural Network
    a. Choose optimiser and loss function
    b. Define a ‘forward’ function
3. (Optimizing)
    a. Training: Go through the data and at each step find gradient and update the parameters
    b. Testing: Get accuracy every epoch on test data

# neural networks

## Data loaders

- Datasets in pytorch usually in form of (inp, label)
- We can either use inbuilt dataset, or create our own
- Data loaders are an incredibly useful utility tool

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64,
shuffle=True)

test_dataloader = DataLoader(test_data, batch_size=64,
shuffle=True)
```

</hackerspace>

# neural networks

## Data loaders

- Dataloader obj is iterable
- Each call returns a batch of data containing (inp, labels)
- Ex: if batch size is 16, we get an array of 16 data elements at each call

</hackerspace>

# neural networks

## Review of classes

Important points

- Classes have __init__() function
- super() function makes the child class inherit all the methods and properties from its parent.

# neural networks

## Using GPU

GPU support is extensive in PyTorch

- If device = ‘cuda’ => GPU
- If device = ‘cpu’ => CPU

How to check and assign

- `torch.cuda.is_available()` returns true if GPU is available

Next steps

- Move the test, train data and the model *to* the GPU
- Using `x.to_device(device)`

Some misc functions

- `torch.cuda.current_device()`
- `torch.cuda.device_count()`
- `torch.cuda.get_device_name(0)`

# neural networks

## First steps

- Make a Neural network class
- Define a forward() function.
    - It is used to do forward propagation

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

</hackerspace>

# neural networks

## optimising

- Enumerate through the data loader
- Get prediction
- Find loss
- Find gradients
- Update parameters

```
for batch in enumerate(d)

Out = model(inp)

Loss = loss_fn(Out, ex_Out)

Loss.backward()

optimizer.step(0
```

</hackerspace>

# next steps

- Look at documentation
- Do small projects
- Wait for the subsequent session on PyTorch by us. Yay!

</hackerspace>