pytorch 2.0!

```
model = torchvision.models.resnet50()
compiled_model = torch.compile(model)
```

StatQuest - machine learning - neuralnetworks, diagrams + Lightning

neural network = popular algorithm in machine learning ... activation functions

each time our optimization code sees all of the training data is called an epoch

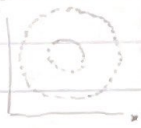
Loss function • gradient descent • stocastic gradient descent • backpropogation

ReLu • ArgMax • SoftMax • Cross Entropy

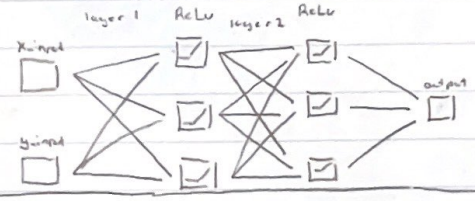
CNN • reccrent neural networks • Long short-term memory • Word Embedding and Word2Vec

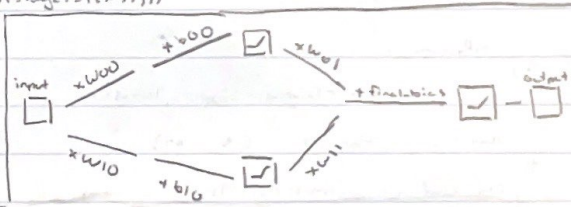
```
import torch, torch.nn as nn, torch.nn.functional as F ... from torchmetrics import SCID ... import lightning as L ... from torch.utils.data import TensorDataset, DataLoader
```

02-B pytorch classification : neural networks are just the combination of linear and nonlinear functions trying to draw patterns from data.




```
from torch import nn
class CircleModelV2(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features=2, out_features=3)
        self.layer_2 = nn.Linear(in_features=3, out_features=3)
        self.layer_3 = nn.Linear(in_features=3, out_features=1)
        self.relu = nn.ReLU()
    def forward(self,x):
        return self.layer_3(self.relu(self.layer_2(self.relu(self.layer_1(x)))))
```





L.LightningModule

```
class BasicNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(                                      )
        self.w01 = nn.Parameter(                                      )
        self.w10 = nn.Parameter(                                      )
        self.b10 = nn.Parameter(                                      )
        self.w11 = nn.Parameter(                                      )
        self.final_bias = nn.Parameter(torch.tensor(-16), requires_grad=False)
```

```
    def forward(self, input):
        a = input * self.w00 + self.b00
        b = F.relu(a)
        c = b * self.w01
        d = input * self.w10 + self.b10
        e = F.relu(d)
        f = e * self.w11
        g = c + f + self.final_bias
        output = F.relu(g)
        return output
```

3

1. //assuming weights and biases were not optimized and requires_grad=True, we could train model to fit graph:

2. inputs = torch.tensor([0.0,0.5,1]) ... labels = torch.tensor([0,1,0])

4. optimizer = SGD(model.parameters(), lr=0.1)

```
for epoch in range(100):
    total_loss=0
    for iteration in range(len(inputs)):
        input_i= inputs[iteration]
        label_i=labels[iteration]
        output_i = model(input_i)
        loss = (output_i-label_i)**2
        loss.backward()      // calculates derivative of loss function w.r.t parameters we want to optimize
                                and accumulates loss from every iteration in loop
        total_loss+= float(loss)
    if (total_loss<0.0001): break
    optimizer.step()
    optimizer.zero_grad()
    print("Step: "+str(epoch)+" Final Bias: " + str(model.final_bias.data)+"\n")
```

1. self.learning_rate=0.01

2. dataset = TensorDataset(inputs,labels)

dataloader = DataLoader(dataset)

3. def configure_optimizers(self):
        return SGD(self.parameters(), lr=self.learning_rate)

```
def train_step(self, batch, batch_idx):
    input_i, label_i = batch
    output_i = self.forward(input_i)
    loss = (output_i -label_i)**2
    return loss
```

4. trainer=L.Trainer(max_epochs=34)

lr_find_results = trainer.tuner.lr_find(model, dataloader, 0.001, 1.0, None)

new_lr=lr_find_results.suggestion()

trainer.fit(model, train_dataloaders=dataloader)