# Designing Neural Networks

(Using PyTorch)

1. Define the architecture
2. Initialize parameters
3. Forward propagation
4. Compute loss
5. Backward propagation
6. Update parameters

1. **Define the architecture:** Decide the number of layers, types of neurons, and connections between neurons in the neural network.
    a. Number of layers:
        i. **Heuristics:** based on prior successful architectures
        ii. **Empirical methods:** experiment with different architectures and evaluate performance
        iii. Domain knowledge
        iv. **Automatic methods:** employ techniques like neural architecture search(NAS) to automatically discover effective architecture
        v. **Transfer learning:** adapt architectures from pre-trained models that have performed well on similar tasks
    b. **Type of neurons:** This refers to different types of activation functions that can be used in the neuron of the network. Activations introduces non-linearity into the network, which is crucial for the network to learn complex patterns in the data.

```
import torch
from torch import nn

input = torch.randn(4)
print(input)

>> tensor([ 0.3706, -1.8396,  0.3385, -0.5086])
```

i. ReLU: $f(x) = \max(0, x)$

```
output_relu = nn.ReLU()(input)
print(output_relu)
>> tensor([0.3706, 0.0000, 0.3385, 0.0000])
```

ii. Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$

```
output_sigmoid = nn.Sigmoid()(input)
print(output_sigmoid)
>> tensor([0.5916, 0.1371, 0.5838, 0.3755])
```

iii. Tanh: $f(x) = \frac{e^{-x}-e^{+x}}{e^{-x}+e^{+x}}$

```
output_tanh = nn.Tanh()(input)
print(output_tanh)
>> tensor([ 0.3546, -0.9508,  0.3261, -0.4689])
```

iv. Softmax: $f(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$

```
output_softmax = nn.Softmax()(input)
print(output_softmax)
>> tensor([0.4011, 0.0440, 0.3884, 0.1665])
```

v. LeakyReLU: $(x) = \begin{cases} x & ,if\ x > 0 \\ \alpha x & ,if\ x \leq 0 \end{cases}$, helps to mitigate the "dying ReLU" problem in deep networks. If $\alpha$ is learned during training, it is called "Parameteric ReLU".

```
output_lrelu = nn.LeakyReLU()(input)
print(output_lrelu)
>> tensor([ 0.3706, -0.0184,  0.3385, -0.0051])
```

vi. Swish: $f(x) = x.\sigma(x) = x.\frac{1}{1+e^{-x}}$

vii. Softplus: $(x) = \ln(1 + e^x)$, smooth approximation of ReLU, used when a non-linear activation is required without dead neurons.

```
output_softplus = nn.Softplus()(input)
print(output_softplus)
>> tensor([0.8955, 0.1474, 0.8766, 0.4708])
```

viii. Gaussian Neuron: $(x) = e^{-x^2}$, used in certain contexts like radial basis function networks for function approximation.

DEMO:

```
import torch
from torch import nn

class MyNet(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer_1 = nn.Linear(4, 3)
    self.layer_2 = nn.Linear(3, 2)

  def forward(self, input):
    output = nn.ReLU()(self.layer_1(input))
    output = self.layer_2(output)

    return output

input = torch.randn(4)
net = MyNet()
output = net(input)
```

```
print(f"Output: {output}")

>> Output: tensor([-0.3010,  0.2444], grad_fn=<ViewBackward0>)
```

DYNAMIC NET:

```
class DynamicNet(nn.Module):
    def __init__(self, num_layers):
        super().__init__()
        self.linears = nn.ModuleList(
            [MyLinear(4, 4) for _ in range(num_layers)]
            )
        self.activations = nn.ModuleDict({
            'relu': nn.ReLU(),
            'lrelu': nn.LeakyReLU()
        })
        self.final = MyLinear(4, 1)

    def forward(self, x, act):
        for linear in self.linears:
            x = linear(x)
        x = self.activations[act](x)
        x = self.final(x)
        return x

dynamic_net = DynamicNet(3)
sample_input = torch.randn(4)
output = dynamic_net(sample_input, 'relu')
output

>> tensor([1.8829], grad_fn=<AddBackward0>)
```

c. **Connection between neurons:** weighted link that transmit signals from one neuron to another, defining how information flows through the network during both forward and backward passes.
   i. **Fully connected (Dense)**
   ii. **Sparse connections:** can be achieved through techniques like CNNs or through attention mechanisms in transformer models.

2. **Initialize parameters:** Initialize the weights and biases for the neurons in the network.
   a. **Random initialization:** Initialize weights randomly from a uniform or normal distribution. This method is simple and commonly used but may require careful tuning of the scale to prevent issues like vanishing or exploding gradients.
   $W \sim \text{Uniform}(-\epsilon, +\epsilon)$ , $W \sim \mathcal{N}(0, \sigma^2)$

```
def random_initialization(layer):
    torch.nn.init.uniform_(layer.weight, -0.1, 0.1)  #
    Uniform
    # or
    # torch.nn.init.normal_(layer.weight, mean=0.0,
    std=0.01)  #Normal
    torch.nn.init.zeros_(layer.bias)  # Initialize biases
    to zero
```

b. **Xavier initialization:** Also known as Glorot initialization, it scales the weights based on the number of input and output neurons. It helps in balancing the variance of gradients across layers, particularly in deep networks.

$for\ \mathrm{tanh}\ and\ sigmoid\ activations(preferred): W \sim \mathcal{U}\left(-\frac{1}{\sqrt{n}}, +\frac{1}{\sqrt{n}}\right) or\ \mathcal{N}\left(0, \frac{1}{n}\right)$

$for\ \mathrm{ReLU}\ activation: W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n+m}}, +\frac{\sqrt{6}}{\sqrt{n+m}}\right) or\ \mathcal{N}\left(0, \frac{2}{n+m}\right)$ , where n is the number of neurons in the previous layer and m is the number of neurons in the current layer.

```
def xavier_initialization(layer):
    torch.nn.init.xavier_uniform_(layer.weight)  # For
    uniform
    # or
    # torch.nn.init.xavier_normal_(layer.weight)  # For
    normal
    torch.nn.init.zeros_(layer.bias)
```

c. **He initialization:** Similar to Xavier, but scales the weights differently to better handle ReLU activation functions. It is effective in networks that predominantly use ReLU or its variants. $for\ \mathrm{ReLU}\ activation(preferred): W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n}}, +\frac{\sqrt{6}}{\sqrt{n}}\right) or\ \mathcal{N}\left(0, \frac{2}{n}\right)$ $for\ other\ activations: W \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n+m}}, +\frac{\sqrt{6}}{\sqrt{n+m}}\right) or\ \mathcal{N}\left(0, \frac{2}{n+m}\right)$, where n is the number of neurons in the previous layer and m is the number of neurons in the current layer.

```
def he_initialization(layer):
    torch.nn.init.kaiming_uniform_(layer.weight,nonlinear
    ity='relu')  # For uniform
    # or
    #torch.nn.init.kaiming_normal_(layer.weight,nonlinear
    ity='relu')  # For normal
    torch.nn.init.zeros_(layer.bias)
```

d. **Orthogonal initialization:** Generate a random orthogonal matrix $\mathrm{W}$ such that $\mathrm{W}^{\mathrm{T}}W = I$. Useful for very deep networks or recurrent neural networks (RNNs). Helps maintain gradients across layers.

```
def orthogonal_initialization(layer):
    torch.nn.init.orthogonal_(layer.weight)
    torch.nn.init.zeros_(layer.bias)
```

DEMO:

```
import torch
from torch import nn

class MyNet(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer_1 = nn.Linear(4, 3)
    self.layer_2 = nn.Linear(3, 2)

  def forward(self, x):
    x = nn.ReLU()(self.layer_1(x))
    x = self.layer_2(x)
    return x

net = MyNet()

# initialization
xavier_initialization(net.layer_1)
he_initialization(net.layer_2)

# For demonstration, printing initialized weights
print(net.layer_1.weight)
print(net.layer_2.weight)
```

3. **Forward propagation:** Perform the forward pass through the network to compute the predicted output for a given input.

4. **Compute loss:** Calculate the difference between the predicted output and the actual target output using a loss function. The choice of loss function depends on the specific task and the nature of the data. Common loss functions include:

```
import torch

output = torch.tensor([0.5, 0.7])
target = torch.tensor([1.0, 0.0])
```

   a. **Mean Squared Error(MSE):** for regression tasks

   $MSE = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i)^2$

```
mse_loss = nn.MSELoss()
loss = mse_loss(output, target)
print('MSE Loss:', loss.item())
```

   b. **Mean Absolute Error(MAE):** for regression tasks

   $MAE = \frac{1}{m}\sum_{i=1}^{m}|\hat{y}^i - y^i|$

```
mae_loss = nn.L1Loss()
loss = mae_loss(output, target)
```

c. **Binary Cross-Entropy Loss:** for binary classification tasks

$Binary\ Cross\ Entropy = -\frac{1}{m}\sum_{i=1}^{m}[\,y^i.\log(\hat{y}^i) + (1-y^i).\log(1-\hat{y}^i)]$

```
bce_loss = nn.BCELoss()
loss = bce_loss(output, target)
```

d. **Categorical Cross-Entropy Loss:** for multi-class classification tasks

$Categorical\ Cross\ Entropy = -\frac{1}{m}\sum_{i=1}^{m}\sum_{c=1}^{C} y_c^i.\log(\hat{y}_c{}^i)$, where is the number of samples and C is the number of classes (for multi-class tasks)

```
ce_loss = nn.CELoss()
loss = ce_loss(output, target)
```

e. **Hinge loss:** used for "maximum-margin" classification, like SVMs

$Hinge = \frac{1}{m}\sum_{i=1}^{m}\max(0, 1 - y^i.\hat{y}^i)$

```
def hinge_loss(output, target):
    return torch.mean(torch.clamp(1 - output * target,
    min=0))

loss = hinge_loss(output, target)
print('Hinge Loss:', loss.item())
```

f. **Kullback-Leibler Divergence (KL Divergence):** Used in probabilistic models and variational inference

$D_{KL}(P||Q) = \sum_i P(i).\log\left(\frac{P(i)}{Q(i)}\right)$

```
kl_loss = nn.KLDivLoss(reduction='batchmean')
loss = kl_loss(output, target)
print('KL Divergence Loss:', loss.item())
```

5. **Backward propagation (Backpropagation):** Propagate the error backward through the network to compute gradients of the loss function with respect to the weights and biases.
   a. **Gradient calculation:** Compute the gradient of the loss function $J$ with respect to each parameter $\theta$ (weights and biases) using chain rule of calculus $\frac{\partial J}{\partial \theta} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L^i}{\partial \theta}$, $L^i$ is the loss function for $i^{th}$ sample.
   b. **Chain rule application:** The gradients are computed layer by layer, starting from the output layer and moving backward through the network.
   c. **Update rule:** $\theta := \theta - \alpha\frac{\partial J}{\partial \theta}$, where $\alpha$ is the learning rate.

6. **Update parameters:** Use an optimization algorithm (e.g., Gradient Descent, Adam) to update the weights and biases of the network based on the computed gradients.

```
import torch.optim as optim
```

a. **Gradient Descent:** It updates parameters in the opposite direction of the gradient of the loss function with respect to the parameters: $\theta := \theta - \alpha \frac{\partial J}{\partial \theta}$

```
optimizer_gd = optim.SGD(model.parameters(), lr=learning_rate)
```

b. **Momentum:** Momentum-based optimization methods improve upon standard gradient descent by adding a momentum term that accelerates gradients in the relevant direction and dampens oscillations.

$v_t = \beta . v_{t-1} + (1-\beta) \frac{\partial J}{\partial \theta}$ ,

$\theta := \theta - \alpha . v_t$

```
optimizer_momentum = optim.SGD(model.parameters(), lr=learning_rate,
momentum=0.9)
```

c. **AdaGrad:** Adagrad adapts the learning rate for each parameter based on the history of gradient updates. It scales the learning rate inversely proportional to the square root of the sum of the squared gradients.

$\theta := \theta - \frac{\alpha}{\sqrt{G_t + \epsilon}} \frac{\partial J}{\partial \theta}$ , here $G_t$ is the sum of squares of past gradients up to time step $t$.

```
optimizer_adagrad = optim.Adagrad(model.parameters(), lr=learning_rate)
```

d. **AdaDelta:** AdaDelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. It introduces a decay factor and uses a running average of recent gradients to update parameters.

$\theta := \theta - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} g_t$ ,

$E[g^2]_t = \rho . E[g^2]_{t-1} + (1-\rho) . g_t^2$ ,

$\Delta\theta_t = \frac{\sqrt{E[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} g_t$

```
optimizer_adadelta = optim.Adadelta(model.parameters(), lr=learning_rate)
```

e. **RMSProp (Root Mean Square Propagation):**

$v_t = \beta . v_{t-1} + (1-\beta) \left(\frac{\partial J}{\partial \theta}\right)^2$,

$\theta := \theta - \alpha \frac{\frac{\partial J}{\partial \theta}}{\sqrt{v_t + \epsilon}}$ , where $v_t$ is running average of the squared gradients, $\beta$ is decay rate(typically close to 1)

```
optimizer_rmsprop = optim.RMSprop(model.parameters(), lr=learning_rate)
```

f. **Adam (Adaptive Moment Estimation):** Adam combines aspects of RMSProp and Momentum methods and adapts the learning rate for each parameter based on estimates of first and second moments of the gradients.

$m_t = \beta_1 . m_{t-1} + (1-\beta_1) \frac{\partial J}{\partial \theta}$ ,

$v_t = \beta_2 . v_{t-1} + (1-\beta_2) \left(\frac{\partial J}{\partial \theta}\right)^2$ ,

$\widehat{m}_t = \frac{m_t}{1-\beta_1^t}$ ,

$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$,

$\theta := \theta - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$ , $\beta_1$ and $\beta_2$ are momentum terms (exponential decay rates, typically set close to 1, $\beta_1 = 0.9, \beta_2 = 0.999$), $m_t$ and $v_t$ areestimates of the first moment (mean) and second moment (uncentered variance) of the gradients respectively.

```
optimizer_adam = optim.Adam(model.parameters(), lr=learning_rate)
```

g. **AdaMax:** AdaMax is a variant of Adam that is more stable for very large gradient updates, where the $l^\infty$ norm of the gradient is used instead of the $l^2$ norm in the update rule.

$m_t = \beta_1 . m_{t-1} + (1-\beta_1). g_t$ ,

$u_t = \max(\beta_2, u_{t-1}, |g_t|)$ ,

$\theta_t = \theta_{t-1} - \frac{m_t}{u_t + \epsilon}$ .

```
optimizer_adamax = optim.AdaMax(model.parameters(), lr=learning_rate)
```

DEMO:

```
for epoch in range(num_epochs):
    optimizer_adam.zero_grad()  # Clear previous
    gradients
    outputs = model(inputs)       # Forward pass
    loss = loss_function(outputs, targets)  # Compute
    loss
    loss.backward()             # Backward pass
    optimizer_adam.step()        # Update weights
```