Lab Sheet 2: Neural Networks (with Python)

Expected Lab Duration: ~2 hours

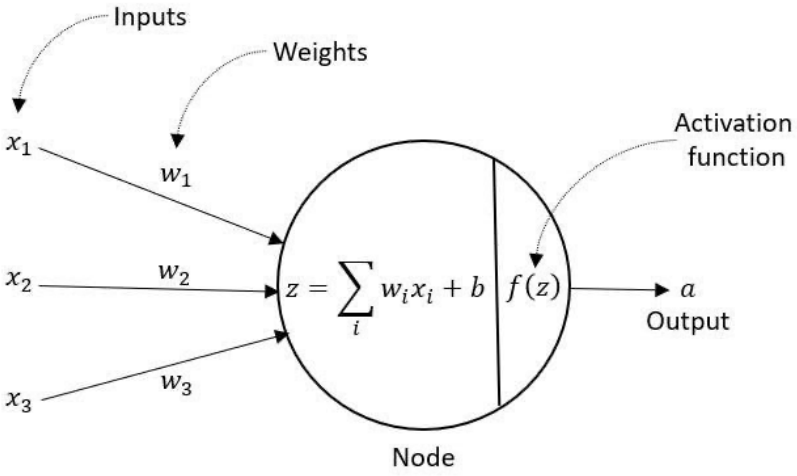
2.1 - Overview of Neural Networks with PyTorch

Introduction to Neural Networks: [You can refer to this amazing Neural Networks playlist, if you are new to this topic]

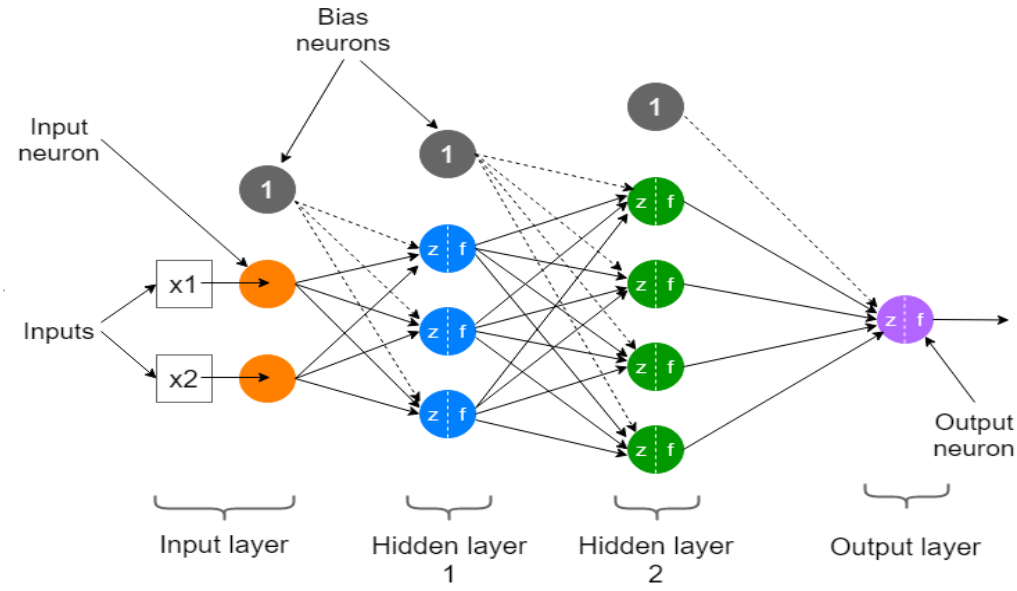
1. **Definition of Neural Networks**:
   - Neural Networks mimic the structure and function of the human brain.
   - Composed of interconnected nodes (neurons) organized in layers.

2. **Motivation for Neural Networks**:
   - Addressing complex tasks that traditional algorithms struggle with.
   - Learning from data to make predictions or decisions.

3. **Basic Components of a Neural Network**:
   - Neurons or Nodes:
     1. Basic units that process and transmit information.
     2. Receive inputs, apply weights, and produce an output.



- Layers:
    1. Input Layer: Receives initial data.
    2. Hidden Layers: Process information.
    3. Output Layer: Produces the final result.



Implementation in PyTorch:

Objective: We will implement a simple neural network using PyTorch to solidify the theoretical concepts. Ideally you should implement a Multi Layer Perceptron first, in order to get clarity about neural networks and backpropagation. However for the purpose of simplicity we will proceed with the intricacies of implementing neural networks with PyTorch.

1. **PyTorch basics**:
   PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR). It is widely used for deep learning applications and research. Following features are relevant to us.

    - **Dynamic Computational Graph**: PyTorch uses a dynamic computational graph, allowing for dynamic changes in the network architecture during runtime. This makes it easier to debug and experiment with different models.

- **Tensors**: The fundamental building block in PyTorch is the tensor, a multi-dimensional array similar to NumPy arrays. Tensors are the core data structure for representing inputs, outputs, and parameters in PyTorch.

- **Autograd**: PyTorch provides automatic differentiation through its autograd package. It automatically computes gradients for tensor operations, which is crucial for training neural networks using gradient-based optimization algorithms.

- **Neural Network Module**: PyTorch includes the torch.nn module for building neural networks. It provides pre-built layers, loss functions, and optimization algorithms, making it easy to construct and train deep learning models.

2. **Install PyTorch** by running the following command after you open a new Python environment or Jupyter notebook.

```
pip install torch torchvision
```

3. **Import the PyTorch library**:

```
import torch
```

4. **Creating a Simple Neural Network**:
    - Define a basic neural network using PyTorch's **nn** Module.
    - Showcase the architecture with input, hidden, and output layers. Following is a skeleton of what it would look like. (You don't have to copy this to your notebook 😛)

```
import torch
import torch.nn as nn

class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
```

```
        # Define layers

    def forward(self, x):
        # Forward pass logic
```

5. **Loss Functions:**

   In deep learning, a loss function, also known as a cost or objective function, quantifies the difference between the predicted output and the actual target values. PyTorch provides a variety of loss functions suitable for different tasks.

   Common Loss Functions:

   1. **Mean Squared Error** (MSE):

```
criterion = nn.MSELoss()
loss = criterion(outputs, labels)
```

   2. **Cross-Entropy Loss**:

```
criterion = nn.CrossEntropyLoss()
loss = criterion(outputs, labels)
```

6. **Activation Functions in PyTorch:**
   Activation functions introduce non-linearity to the neural network, allowing it to learn complex patterns. PyTorch provides a variety of activation functions that can be easily integrated into your neural network model.

   Common Activation Functions:

   1. **ReLU** (Rectified Linear Unit):

```
activation = nn.ReLU()
output = activation(input)
```

   2. **Sigmoid**:

```
activation = nn.Sigmoid()
output = activation(input)
```

3. **Tanh**:

```
activation = nn.Tanh()
output = activation(input)
```

7. **Optimizers in PyTorch**:

   Optimizers play a crucial role in training neural networks by adjusting the model parameters based on the computed gradients. PyTorch provides a variety of optimization algorithms that facilitate the convergence of the model during training.

   - Stochastic Gradient Descent (SGD):

```
optimizer = torch.optim.SGD(model.parameters(),
lr=learning_rate)
optimizer.zero_grad()  # Zero the gradients
loss.backward()  # Backpropagation
optimizer.step()  # Update weights
```

   - Adam Optimizer:

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

8. **Understanding Feedforward and Backpropagation with code**:
    - **FeedForward**: We define a simple neural network (class **SimpleNN**) with one hidden layer. The network has two fully connected layers (using **nn.Linear** function) with specified input and output features.

      The **forward** function defines the sequence of operations during the forward pass, including the ReLU activation function.

      An instance of the model is created, and a sample input tensor is defined. The input tensor is passed through the model to perform the feedforward operation. The output of the feedforward operation is printed.

      __init__ is a constructor for our SimpleNN class.

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

# Define a simple neural network architecture
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(in_features=2, out_features=5)  # Layer 1:
Number of Input features: 2, Output features: 5
        self.fc2 = nn.Linear(in_features=5, out_features=1)  # Layer 2:
Number of Input features: 5, Output features: 1

    def forward(self, x):
        x = F.relu(self.fc1(x))  # Apply ReLU activation to the output
of the first layer
        x = self.fc2(x)  # Output of the second layer
        return x

# Create an instance of the neural network
model = SimpleNN()
```

```
# Define a sample input
input_data = torch.tensor([[0.5, 0.3]])  # Input tensor with shape (1,
2)

# Perform the feedforward operation
output = model(input_data)

# Print the output
print("Input Data:", input_data)
print("Feedforward Output:", output)
```

- **Backpropagation with PyTorch:** Adding backpropagation functions to the above Neural Networks implementation will make it look like this.

The existing NN and feed-forward functions:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(in_features=2, out_features=5)  #
Input features: 2, Output features: 5
        self.fc2 = nn.Linear(in_features=5, out_features=1)  #
Input features: 5, Output features: 1

    def forward(self, x):
        x = F.relu(self.fc1(x))  # Apply ReLU activation to the
output of the first layer
        x = self.fc2(x)  # Output of the second layer
        return x
```

Time to add a function for backpropagation:

```
    def backpropagate(self, loss):
        # Initialize the optimizer (SGD in this case)
        optimizer = optim.SGD(self.parameters(), lr=0.01)

        # Zero the gradients to prevent accumulation
        optimizer.zero_grad()

        # Backward pass to calculate gradients
        loss.backward()

        # Update the weights using the optimizer
        optimizer.step()
```

Initialize your network:

```
# Create an instance of the neural network
model = SimpleNN()

# Define a sample input and target
input_data = torch.tensor([[0.5, 0.3]])  # Input tensor with
shape (1, 2)
target = torch.tensor([[0.8]])  # Target tensor with shape (1,
1)

# Perform the feedforward operation
output = model(input_data)
```

Use loss functions from the **nn** library:

```
# Calculate the Mean Squared Error (MSE) loss
criterion = nn.MSELoss()
loss = criterion(output, target)
```

Print the initial state:

```
print("Initial State:")
print("Input Data:", input_data)
print("Target:", target)
print("Initial Output:", output)
print("Initial Loss:", loss.item())
```

Perform backpropagation and update weights:

```
model.backpropagate(loss)
```

Perform another feedforward operation after backpropagation:

```
output_after_backprop = model(input_data)
loss_after_backprop = criterion(output_after_backprop, target)
```

Print the updated state:

```
print("\nState After Backpropagation:")
print("Updated Output:", output_after_backprop)
print("Updated Loss:", loss_after_backprop.item())
```

2.2 - Training Neural Networks with PyTorch

Training a neural network involves iterative optimization of model parameters to minimize a defined loss function. PyTorch simplifies this process through its dynamic computation graph and user-friendly APIs.

Steps for Training:

1. **Define the Model**:

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        # Define layers and architecture

    def forward(self, x):
        # Define forward pass
        return x
```

2. **Splitting the Dataset**:

   Before training, split the dataset into training and test sets. This ensures an unbiased evaluation of the model's performance.

```
from sklearn.model_selection import train_test_split

# Assuming 'data' contains features and 'labels' contains
corresponding labels
X_train, X_test, y_train, y_test = train_test_split(data,
labels, test_size=0.2, random_state=42)
```

3. **Choose Loss Function**:

   Select an appropriate loss function based on the task (e.g., CrossEntropyLoss for classification, Mean Squared Error for regression).

```
criterion = nn.CrossEntropyLoss()
```

4. **Choose Optimizer**:

   Select an optimizer that will be used to update the model parameters based on gradients.

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

5. **Training Loop**:

   Refer to code below where the model is trained `num_epochs` number of times with the selected loss function and optimisers.

```
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    for inputs, labels in train_loader:
        optimizer.zero_grad()  # Zero the gradients
        outputs = model(inputs)  # Forward pass
        loss = criterion(outputs, labels)  # Compute the loss
        loss.backward()  # Backpropagation
        optimizer.step()  # Update weights
```

6. **Testing**:

```
#Testing the model
model.eval()
all_test_predictions = []
all_test_labels = []

with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        predictions = torch.argmax(outputs, dim=1)
        # add other steps
```

7. **Validation Loop (Optional):**

```
model.eval()  # Set the model to evaluation mode
with torch.no_grad():
    for inputs, labels in val_loader:
        outputs = model(inputs)
        # Evaluate performance on validation set
```

8. **Adjustments and Hyperparameters:**

   Fine-tune hyperparameters such as learning rate, batch size, and model architecture based on validation performance. This activity is generally done iteratively while we compare the model performance on a given set of hyperparameters.

Monitoring Training Progress:

1. **Loss Visualization:**
   Plot the training and validation loss over epochs to monitor convergence.
   #TODO (1/2) : Figure out how this can be implemented.

2. **Accuracy or Other Metrics:**

   Track additional metrics like accuracy for classification tasks or mean squared error for regression.
   `#TODO (2/2)` : Figure out how this can be implemented.

This section provides a comprehensive guide to training neural networks using PyTorch. It covers essential steps, including defining the model, choosing loss functions and optimizers, setting up training loops, and monitoring training progress.

(Exercises on next page 💀)

2.3 - Exercises

Let's solve a simple exercise to apply the feedforward and backpropagation using a well-known dataset such as the Iris dataset. The goal is to train a neural network to classify iris flowers into different species based on their features.

Create a Jupyter Notebook and solve the following exercises using the same flow of steps for training a neural network that we have discussed above. Make sure you get habituated to adding descriptions to your code in notebook files.

1. Load the Iris dataset from a library like scikit-learn (import from **sklearn.datasets**).

2. Normalize the features and encode the target variable. This involves two main preprocessing tasks: normalization of the feature values and encoding of the target variable (usually categorical) into a numerical format. (You need to figure this part out yourself. **Hint**: explore **StandardScaler**, **LabelEncoder** in sklearn library)
3. Split the dataset into training and testing sets.
4. Implement a feedforward neural network with at least one hidden layer using PyTorch.
5. Train the neural network using the training set and backpropagation.
6. Evaluate the trained model on the testing set and calculate the accuracy.
7. Experiment with different hyperparameters.
    - Train model with different optimisers (**SGD / Adams**)
    - Train model with different number of hidden units/layers.
    - Train model with different learning rate, test/training splits etc.

    Write a conclusion about how changing all these parameters affect the performance of your model.