**Slide1:**

While we are getting started, I would love to know what brings you to this workshop?

**Those were some interesting answers.**

**Schedule:**

0:05 - 0:30 About Neural Networks

0:30 - 1:20 Hands-on with Jupyter Notebook

1:20 - 1:30 Wrap-up and Discussion

**Slide 2:**

But before going into more details I would like to begin by acknowledging that I .

**Slide 3:**

Active participation makes the session so much fun and gives me and your peers much more energy. Your voices and perspectives enlivens the session. We encourage you to engage with each other and us.

The participants window lists everyone in the session and click the icons at the bottom to communicate with the us.

You can also use the Chat windows to comment or ask questions at any time. It is also a good place to share problems with your audio connection.

**Slide 4:**

This workshop is eligible for the Canadian Certificate in Digital Humanities <https://ccdhhn.ca/>. This is a program that allows you to claim non-credit workshops and training toward a certificate. If you are interested in claiming 2 hours from this workshop please fill out this form: <https://ubc.ca1.qualtrics.com/jfe/form/SV_cwiqRxedypNPhA2>. This is how we track attendance for the certificate, it will only be shared during the workshop and will not be emailed to you after."

**Slide 5:**

So, to touch various viewpoints of machine learning regression, we have the following learning objectives for this workshop:

**Slide 6:**

For hands-on exercises, we will use [Python](https://www.python.org/) on [Jupyter Notebooks](https://jupyter.org/). You don’t need to have Python installed. Please make sure that you have a [UBC Syzygy](https://ubc.syzygy.ca/) or a [Google Colaboratory](https://colab.research.google.com/) account. (You will need a CWL login to access Syzygy.) hands-on exercises, programming tools and libraries, such as [Python] and [scikit-learn] prior familiarity with Python programming is recommended, we do not study the codes in detail

**Slide 7:**  
What is an Artificial Neural Network?

* "A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates."
* Ask about brain cells?

**Slide 8:**

* Input Layer: Receives data. Like our senses perceive the environment.
* Hidden Layers: Process data. The 'thinking' happens here.
  + Depth Over Width: Hidden layers allow neural networks to be "deep", giving depth to the learning process.
  + Feature Transformation: Hidden layers help transform inputs in a way that makes them separable by the time they reach the output layer.
  + Capturing Complexities: The more hidden layers and neurons, the more complexities a network can capture. However, this also increases the risk of overfitting.
* Output Layer: Provides the result.
* Weights & Biases: These determine the strength and direction of the signal between neurons. Adjusted during learning.
* Activation Function: Determines if a neuron should be activated or not.

**Slide 9:**

A typical network is made of multiple such neurons.

**Slide 10:**

* **Forward Propagation**: Input data is passed through the network to get an output.
* **Loss Calculation**: The difference between the predicted output and the actual output (known as the 'loss' or 'error').
* **Backpropagation**: The process of adjusting weights and biases to reduce the error.
* **Iterative Process**: Repeated many times until the network performs satisfactorily.

**Slide 11:**  
**Overfitting**: When the network is too specific to the training data and performs poorly on new, unseen data.

* **Data Requirements**: Neural networks need a lot of data to train effectively.
* **Computationally Intensive**: Can require powerful hardware.
* **Interpretability**: Often seen as "black boxes" - it's challenging to understand why they make certain decisions.

**Slide 12:**Let’s dive deeper   
  
**Slide 13:**

End

**Bias:**

**Definition**: In artificial neural networks, a bias is an extra input to each neuron, in addition to the regular inputs from the previous layer. It's used to offset the output of the neuron and is adjusted in the learning process alongside the weights.

**Purpose**: The bias allows the activation function to be shifted to the left or right, which can be essential for the neuron to fire under the right conditions. It can be thought of as the y-intercept in a linear equation, though neural networks often work in higher dimensions and are non-linear.

**Example**: If you're modeling a simple linear relationship

y=mx+c*y*=*mx*+*c*

, 'm' is analogous to the weight and 'c' to the bias.

**Activation function:**

### 1. What does an activation function do?

An activation function introduces non-linearity to the output of a neuron. It takes the output from a neuron (a weighted sum of its inputs plus a bias) and processes it to produce the neuron's final output, which is passed onto subsequent layers.

### 2. Why do I need an activation function?

* Introducing Non-linearity: Without non-linearity, even a deep neural network would behave like a single-layer linear model, which means it would not capture complex patterns or relationships in the data. Non-linearity helps in representing more intricate decision boundaries.
* Activation Threshold: Activation functions can set a threshold, determining whether or not a neuron should be activated. This helps to make the network's behavior more dynamic and expressive.

### 3. Different Activation Functions:

* Sigmoid:
  + Range:
  + (0,1)(0,1)
  + Commonly used in the output layer for binary classification tasks.
  + Limitations: Can suffer from vanishing gradient problem, especially in deep networks.
* Tanh (Hyperbolic Tangent):
  + Range:
  + (−1,1)(−1,1)
  + Like a scaled sigmoid.
  + Can also suffer from the vanishing gradient problem but less so than sigmoid.
* ReLU (Rectified Linear Unit):
  + Formula:
  + f(x)=max(0,x)f(x)=max(0,x)
  + Most popular and widely used in hidden layers of deep networks.
  + Helps mitigate the vanishing gradient problem.
  + Limitations: Neurons can sometimes "die" and stop updating if they always output zero.
* Leaky ReLU:
  + Allows a small gradient for negative values to prevent neurons from "dying".
* Softmax:
  + Used in the output layer for multi-class classification problems.
  + Converts output to probability distributions for each class.

**Cuda cell:**  
This code is related to PyTorch, a popular deep learning framework. It is checking for the availability of a GPU (Graphics Processing Unit) and setting the device to use for subsequent computations

GPU helps with parallel computing.   
  
  
S**et Random Seed**:

* manualSeed is set to 1234. This is done to ensure reproducibility of results. When you set a seed, the random numbers generated by various libraries will be the same every time the code is run.
* Seeds are set for Python's random, numpy, and torch. Additionally, if a GPU is available, a seed for CUDA operations is also set.
* **ChunkSampler Class**:
  + A custom ChunkSampler class is defined. This sampler is designed to sample a chunk of data sequentially from some starting offset. It's useful to manually split data, e.g., for creating training and validation datasets from a larger dataset.
  + The sampler will start from the specified start index and will pick num\_samples consecutive samples.
* **Download and Prepare MNIST Dataset**:
  + The MNIST dataset is loaded for training (train=True) and testing (train=False). It's stored in the ../data directory.
  + Images in the dataset are transformed into tensors using transforms.ToTensor().  
      
      
    Tensors are multi-dimensional arrays that can represent a wide variety of data types and shapes
* **Split the Dataset**:
  + The MNIST training dataset is split into a training set and a validation set.
  + 80% of the training dataset is used for training (num\_train\_samples), and the remaining 20% is used for validation (num\_valid\_samples).
* **Print Dataset Sizes**:
  + The number of samples in the training, validation, and test datasets are printed.
* **Create DataLoaders**:
  + DataLoader objects are created for training, validation, and test datasets.
  + The custom ChunkSampler is used to split the training dataset into training and validation parts.
  + batch\_size is set to 100, which means that the neural network will be trained using 100 samples at a time (i.e., minibatch gradient descent with batches of size 100).
  + Shuffling of data is turned off (shuffle=False). This is because the custom sampler already determines the order of the samples.
* **Scales the Image**: By multiplying the pixel values by 255, the range is transformed back to [0, 255], which is the standard range for grayscale images. This step converts the floating-point numbers back into their original grayscale intensity values.
* **Inverts the Image**: By subtracting the scaled pixel values from 255, the image is color-inverted. Pixels with values close to 0 (black) become close to 255 (white), and pixels with values close to 255 (white) become close to 0 (black).

The scaling step is necessary to ensure that when the image is plotted using matplotlib, it correctly interprets the pixel intensities. If the pixel values were left as floating-point numbers in the range [0.0, 1.0], matplotlib would still plot them, but the intensities would be off if you didn't specify the correct color mapping and intensity range. Scaling ensures that the pixel values are in a standard format and range, making it easier to work with and visualize using various tools.

Training   
  
S**etting Hyperparameters**:

* + input\_size: The size of the input layer, which is 784. This corresponds to the number of pixels in a flattened 28x28 MNIST image.
  + hidden\_size: The size of the hidden layer, set to 500.
  + num\_classes: The number of output classes, set to 10. This is because MNIST has 10 digit classes (0 to 9).
* **Define the MLP Class**:
  + The neural network is defined as a subclass of nn.Module.
  + In the \_\_init\_\_ method:
    - self.hidden\_layer: Defines the hidden layers of the MLP. It has two linear (fully connected) layers, each followed by a ReLU activation function.
    - self.output\_layer: Defines the output layer of the MLP, which is a linear layer.
  + In the forward method:
    - This method defines the forward pass of the neural network. Data (x) is passed through the hidden layer, and then through the output layer. The final output is returned.
* **Model Initialization**:
  + model = MLP(input\_size, hidden\_size, num\_classes): An instance of the MLP class is created, initializing the neural network.
  + model = model.to(device): The model is transferred to the specified device (either CPU or GPU, based on earlier code you shared).
* **Print the Model**:
  + The architecture of the initialized neural network is printed.
* **Count and Print Parameters**:
  + The number of trainable parameters in the model is computed and printed. This gives insight into the model's size and capacity.
* **Backup Initial Weights**:
  + init\_model\_wts = copy.deepcopy(model.state\_dict()): The initial weights of the model are saved using deepcopy. This might be useful later if you want to revert the model back to its initial state (e.g., for re-training or comparing against other initializations).
* **Set Learning Parameters**:
  + learning\_rate: Specifies the learning rate for the optimization algorithm.
  + criterion = nn.CrossEntropyLoss(): The loss function is defined as cross-entropy loss, commonly used for classification tasks.
  + optimizer = torch.optim.SGD(model.parameters(), lr=learning\_rate): The stochastic gradient descent (SGD) optimization algorithm is chosen for training the model. The model's parameters and the specified learning rate are passed as arguments.

In summary, this code defines a simple MLP neural network with one hidden layer for classifying MNIST images, initializes it, sets up a loss function, and chooses an optimization method. The model is ready for training on the MNIST dataset.

* **import Required Modules**:
  + time: Used for tracking how long the training process takes.
  + Variable from torch.autograd: This is an older construct from PyTorch that was used to differentiate tensors. In recent versions of PyTorch, tensors are differentiable by default, so Variable is often not used.
* **Reset Model to Initial State**:
  + model.load\_state\_dict(init\_model\_wts): Resets the model's weights to its initial state using the saved weights from before.
* **Initialize Time Tracking**:
  + since = time.time(): Records the current time for tracking the total training duration.
* **Training Settings**:
  + num\_epochs: Number of times the model will see the entire dataset.
  + train\_loss\_history and valid\_loss\_history: Lists to track the average loss for each epoch during training and validation, respectively.
* **Training Loop**:
  + For each epoch:
    - **Training Phase**:
      * model.train(): Puts the model in training mode. This is necessary because certain layers like dropout or batch normalization behave differently during training and evaluation.
      * Loop over each batch of data in train\_loader:
        + Load images and labels to the defined device (GPU if available).
        + Flatten the images so they can be input to the MLP.
        + Zero out any old gradients.
        + Perform a forward pass to compute predictions.
        + Compute the loss between predictions and true labels.
        + Perform a backward pass to compute gradients.
        + Update model weights using the optimizer.
        + Track training loss.
    - **Validation Phase**:
      * model.eval(): Puts the model in evaluation mode.
      * Loop over each batch of data in valid\_loader:
        + Similar to the training loop, but without the backpropagation and weight updating steps. This is because we're just evaluating the model, not training it.
        + Track validation loss.
    - Compute and print average losses for this epoch for both training and validation data.
    - Store average losses in their respective history lists.
* **Print Training Duration**:
  + Compute the total time taken for training and print it.

In summary, the code is a typical training loop for neural networks in PyTorch. The model is trained for a specified number of epochs, and for each epoch, the model is trained using the training dataset and evaluated using the validation dataset. The performance (loss) of the model on both datasets is tracked and printed.

**Receptive field**  
Imagine you're using binoculars to look at a painting. The small circle you see through the binoculars is like the "receptive field" in a CNN - it's the part of the image the model is focusing on at any given moment.

In a standard neural network, it's like you're looking at the whole painting all at once. But in a CNN, we focus on small patches of the painting one at a time, then combine those patches to understand the whole image.

Now, here's a cool trick CNNs use:

Instead of just using one big lens to look at a large part of the painting, CNNs use a series of smaller lenses to look at smaller parts, then combine them. So, the first lens might see a small patch, then the next lens looks at a slightly bigger patch that includes the first one, and so on.

Why do this? Two main reasons:

* **Fewer Parts Needed**: Using a series of small lenses requires fewer parts than one big lens. In our neural network, this means we need fewer "parameters", which helps the model learn faster and more efficiently.
* **Better Understanding**: By using smaller lenses one after another, we can focus on the details and then gradually piece them together. This adds depth to our understanding of the image.

Importing Lib  
Defining the Architecture   
Implementing the model   
Seeing the results