MLP Notes

Overview

This document contains notes on the process of using the Multi-Layer Perceptron (MLP) model for image classification, specifically using the MNIST dataset of handwritten digits. It details the steps and concepts involved, such as data normalization, model architecture, training, and evaluation.

Data Preparation

- The MNIST dataset consists of 28×28 black and white images of handwritten digits (0 to 9).
- Data normalization is performed to reduce processing times and improve learning rate. This is done by converting pixel values to floating points, scaling them to a range of 0-1, and then normalizing each pixel by subtracting the mean and dividing by the standard deviation.

Model Architecture

- The MLP model consists of two hidden layers and a final output layer.
- The input layer has 784 neurons (one for each pixel), the first hidden layer has 250 neurons, the second hidden layer has 100 neurons, and the output layer has 10 neurons (one for each digit).
- The model uses Rectified Linear Unit (ReLU) activation functions for the hidden layers.
- The transformation from 784 → 250 → 100 → 10 is done by linear layers (fully connected or affine layers).

Training

- The training process involves passing a batch of images through the model, calculating the loss, computing the gradients of each parameter, and updating the parameters.
- The Adam algorithm is used for optimization, and the Cross Entropy Loss function is used to calculate the loss/error.
- After each epoch (full pass over the data), the model's performance is evaluated on a separate validation set.

Evaluation

- After training, the model's performance is evaluated on a test set.
- Further analysis is conducted to understand the errors made by the model.
- Principal Component Analysis (PCA) is applied to find the most important directions of variation in the data, reducing the number of features and simplifying the data for visualization.

Notes

A MLP, or Multi-Layered Perceptron is a Artificial Neural Network, which means it is designed to mimic the learn model of a human brain

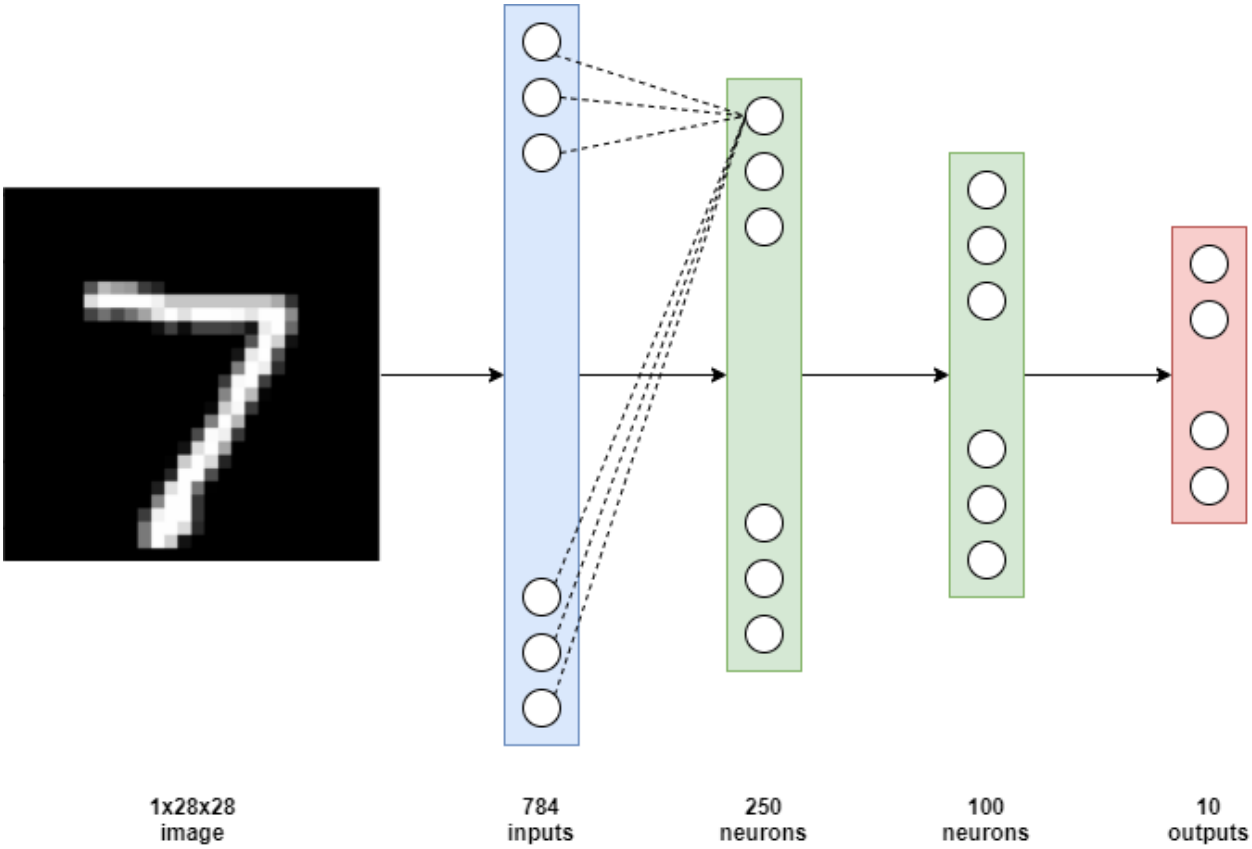
- It recognizes patterns and is trained to make choices based on that pattern matching skill.

Layers- a large part of the MLP model. An MLP model is like the layers in a rainbow cake, and each layer is made up of ‘neurons’, processing units, to mimic the brain.

Connections- The neurons of different layers are connected through connections, they share information from one to another. Like electric synapses between real neurons . Each connection has a weight value to describe it

Learning- as the MLP model inputs and trains on real data (images or text), the model adjusts the weight of each connection to improve the model

- this project, i'll be using pytorch and torchvision to perform image classification using the MLP model. Its known as the Feedforward network
- The dataset we'll be using is the famous MNIST dataset, a dataset of 28×28 black and white images consisting of handwritten digits, 0 to 9. I'll be training a model to detect what digit is in a picture



Model of how an image will be classified using the MLP model

Goal is to

1. Process the dataset
2. Build the model
3. Train the model to be accurate

Dependencies and Jupyter notebook

- Python has a lot of libraries to do different tasks, like ML or web dev
- Python and Jupyter notebook have a virtual kernel feature where you can install the needed libraries for a project on a kernel (so you don't need to worry ab what you have installed on your computers version of python)

To check what virtual environment to use and install the pytorch libraries for this project, go to cmd and do the following:

- jupyter kernelspec list (list out kernels used in jupyter notebooks)
- python -m venv python3 (get into the kernel needed)
- .\python3\Scripts\activate
- pip install torchvision install the proper libraries for that kernel and workbook
- Found out you can also install libraries to kernel directly in Jupyter notebook
    - %pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu

With everything imported and the data pulled in, its time to normalize the data

- in the case of images, normalizing the data means reduce the potential number range to calculate for each pixel
- pixels can have a range from 0 to 255, reducing that range to 0-1 can help reduce processing times
    - updating weights based on 0-1 instead of 0-255 takes less space and improves learning rate
- it can also reduce finding a local minimum (where the solution is performing good relatively but not moving towards the best solution). Reducing the range down helps explore effectiveness better to potentially find best solution

Steps to normalize MNIST data

- convert pixel values to floating points
- The mean (average intensity) and standard deviation (spread of intensities) (only for training not testing data!)
- divide each pixel value by 255 to scale it to the 0-1 scale
- Normalize each pixel by subtracting the mean and dividing by b vgthe standard deviation
    - it centers data around 0 and ensures it is within one standard deviation

Effect:

By normalizing the MNIST data, the model can treat all digits (features) more equally, leading to faster convergence and potentially better classification accuracy. This is particularly helpful because handwritten digits can have varying thicknesses and lighting conditions, leading to variations in pixel intensity. Normalization helps account for these variations and allows the model to focus on learning the underlying patterns of the digits themselves.

Transforms

- they define how data is processed and augmented before feeding it to a model
- transforms can involve scaling, cropping, rotating, and normalizing data
- transforms.Compose: this function allows combining multiple transforms into a single pipeline, each data in list with sequentially be processed with each transform in the list being defined
- there are two types of transforms, those that happen before converting to a tensor, and then those on a tensor object
- train transforms: rotating an image by x degrees, the ‘fill’ argument is used to fill any blank spaces created by rotating
    - train cropping: it adds padding (a couple pixels here or there), then removes a random 28×28 part from image, its use to recognize digit

anywhere it positioned

- `ToTensor()`: This converts the PIL image (Python Imaging Library format) into a Pytorch tensor, a format suitable for deep learning models.
- normalize(): only occurs on tensor object, it performs normalization described above
- training transforms help create more training examples
- tensor transforms help ensure consistency in processing and don't create more data

Epoch: an iteration of the training data through the algorithm

Dataloader: its a pytorch tool which also iterating of data in dataset over batches to save memory usage

- improves efficiency of training
- **Training Set:** Used to train the model parameters. This set should be shuffled for each epoch (training iteration) to ensure the model sees examples in a random order and avoids overfitting to specific data patterns.
- **Validation Set:** Used to monitor model performance during training and potentially adjust hyperparameters like learning rate. It's helpful to keep the validation set separate from training and not shuffle it to get a consistent evaluation.
- **Test Set:** Used for final evaluation of the trained model's generalization ability on unseen data. Similar to validation, the test set should not be shuffled.

Batch size refers to the number of images processed by the model at once during training. Here, the code uses a batch size of 64.

In summary,
`DataLoader` helps manage data loading in batches for training, validation, and testing. While shuffling the training set is crucial for stochastic gradient descent, validation and test sets typically remain unshuffled for consistent evaluation. The batch size should be chosen carefully considering memory constraints and potential benefits for training speed.

- larger dbatch size improves training speed but cost memory contraints

SGD

Imagine you're lost in a big mountain range and want to find the lowest valley (minimum point) to set up camp. Stochastic gradient descent (SGD) is like a way to explore the mountains and find the valley as quickly as possible. Here's how it works in a simplified way:

1. **Blindfolded Hiker:** You're blindfolded, so you can't see the whole mountain range. You can only feel the slope of the ground you're standing on (like the steepness of the learning curve).
2. **Small Steps:** You take small steps in the direction that feels most downhill (like adjusting the model weights in the direction that reduces error).
3. **Listen for Your Guide:** At each step, someone shouts a random direction (like a random sample from your training data). You don't always follow this direction, but it helps you explore different parts of the mountain (helps prevent getting stuck in local minima).
4. **Repeat:** Keep taking small steps downhill, occasionally listening to random directions, until you feel like you've reached a pretty flat area (minimum point where the model performs well).

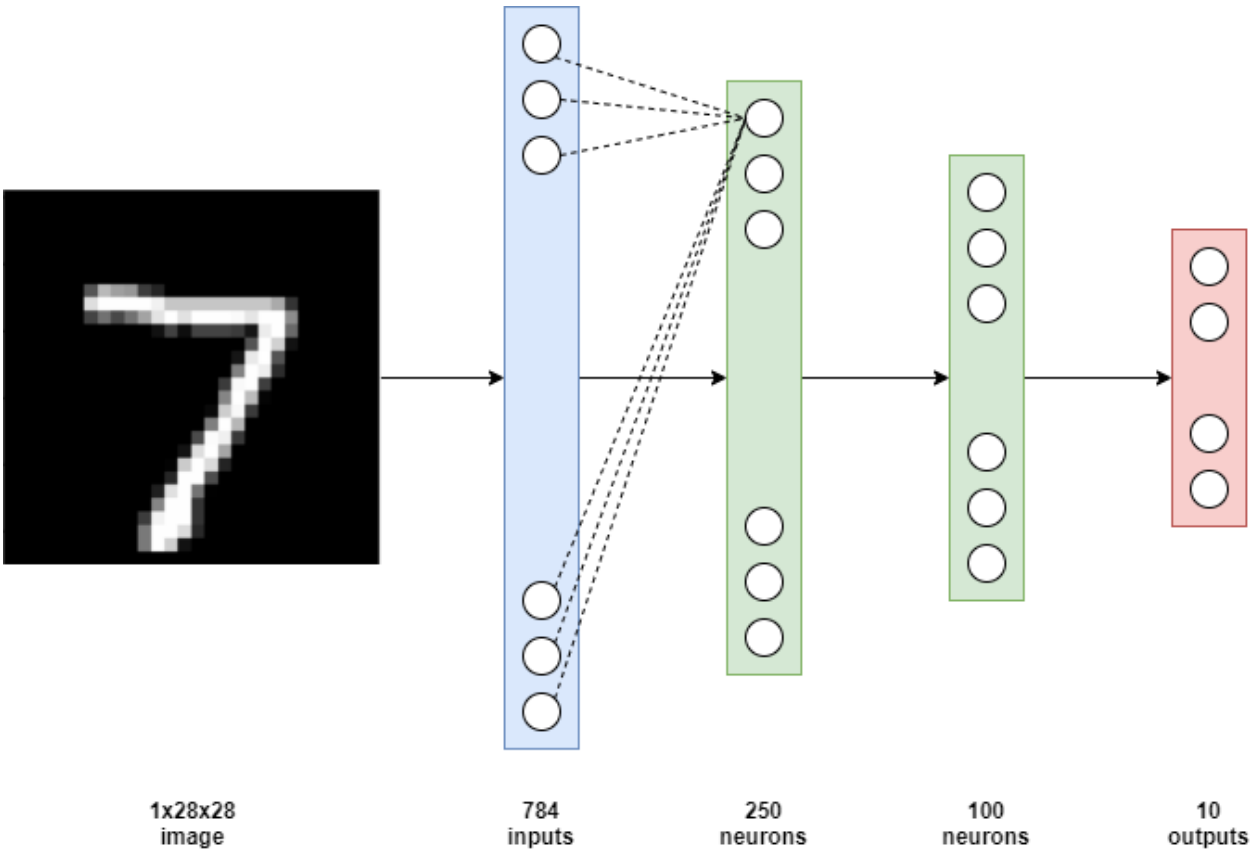
Here's why it's called "stochastic":

- **Stochastic** means "random." In SGD, the random directions you get are like random samples from your training data. This helps explore the whole mountain range and avoid getting stuck on small hills (local minima).

Validation data

- The validation set is a hold-out set used to monitor model performance on unseen data during training.
- It's crucial not to use the test set for validation, as the test set is for final evaluation after training.
- The validation set is created by splitting a portion (e.g., 10%) from the original training set.

- This ensures the validation data comes from the same distribution as the training data.



The neural network (MLP) will have two hidden layers in green. The first layer is 784 inputs for each pixel and the final layer is the 10 outputs we are training for to decide which digit is it.

- we flatten the 1×28×28 image (one color channel, and 28×28 pixels) into a 784 element vector, and we say there are 784 features (columns).
- MLPs cant handle 2 or 3d data so we flatten it.
  - the 784 dimensional input is then passed through the first hidden layer to transform it into 250 dimensions
  - then it is passed through another filter which transforms it to 100 dimensional vector

- the final vector layer transforms the output to a 10 dimensional vector which is 0-9 to output the inferred digit
- The transformation from 784 → 250 → 100 → 10 is done by Linear layers (or fully connected or affine layers).
- in these layers, every element in one layer is connected to every element in the next layer. each element is a neuron. This is based on the milions of interconnected neurons in the brain
- each connection between one layer of the brain and the next has a weight value associated with it
- the input to one neuron is the sum of the weights of every connection to neurons in the previous layer, plus a weighted bias (bias value is always 1)
    - the neuron implements an activate function on this weighted sum, where the function is nonlinear and allows the neural network to learn non-linear functions between inputs and outputs
    - In an MLP, each neuron receives weighted inputs from the previous layer, sums them, and applies an activation function to the sum.
    - The activation function determines the output of the neuron based on the weighted sum.
        - by introducing non-linearity, the model opens possiblity for finding more complex patterns
        - theres diff type of activation functions and it impacts the performance of the model

    But how does non-linearity help here?

    - think about a straight line, can find any trends from it. Adding shapes, curves, and slopes would allow for uniqueness and help decipher different points in line
    - reaal world dat has intricate relations between features, non-linearity lets models caputre this!

        - addes nuance: ex: For example, imagine predicting house prices based on size and location. A linear model might struggle to capture the non-linear relationship between price and factors like distance to amenities or presence of desirable features.
    - nin-linear functions can help id complex relationships and identify them as “new features” for the next layer
    - MLPs with multiple hidden layers and non-linear activations can learn intricate patterns by building hierarchies of features, where each layer transforms the previous one to uncover complex relationships in the data.
    - Activation functions allow MLPs to create curved decision boundaries, enabling them to effectively classify data that isn't easily divided with straight lines.

What the code does is take the input batch of images and flatten them, and then pass them to the linear layers. Theres three linear layers (two hidden and the final one)

- we pass it thru the first linear layer using the input_fc function, which calculates the weighted sum of each input and then applys the rectified linear unit activation function elementwise.
- this result is passed through another linear layer (hidden_fc), again applying the same function element wise.
- finally we pass the function trhough the final linear layer, output_fc
    - here we return the output layer, but the last linear layer for analysis

The Rectified Linear Unit (ReLU) function runs max(0, x), where x is the weighted sum of inputs for that neuron

- if the weighted sum is positive, ReLU tracks it, if negative, it ignores it
- there can be negative numbers bc biases might be negative and the weights them selves be might negative

    - Even with ReLU in all layers, negative outputs are possible mathematically. The magic lies in the interplay between weights (multipliers) and biases (added values) assigned to connections between neurons. If, in a later layer, a positive value from a ReLU layer is multiplied by a negative weight, the output can become negative.
- we shouldnt use activation function directly on the input or output
- PyTorch combines activation functions to be applied on the output with the functions which calculate the *loss*, also known as *error* or *cost*, of a neural network. This is done for numerical stability.

- Why do we use two hidden layers? Why are we using 250 and 100 feature layers? This is decidedby programmer and theres no magic formula to decide these values

General idea:

- neural network extract features from data
- - layers closer to input learn to extract general features (lines, curves, edges)
- later layers combine the features extracted from previous layers into high level features (intersection of 2 lines making a cross, multiple curves making a circle)
- we force the neural network model to learn these features by reducing the neurons in each layer
- this way, the model extracts useful features by learning compressing info to only essewntial parts,
- we want neural network to use multiple layers to learn info compression

Training the model

A 100,000 ft view of Neural Network training is:

- pass a big batch of data through the model
- calculate the loss of your batch by comparing the models predicitions vs the actual labels
- calculate the gradient of eacxh of your parameters with respect to the loss
    - gradient: a vector which points in the direction of steepest ascent of a function
    - in context of loss, it tells you how much the loss changes for small parm change in model
    - if a landscape represents loss value, the gradient points downhill to direction of greatest decrease in loss
    - Loss measures how well a model performs (lower loss is better).
- By analyzing gradients, we can adjust the model's parameters to minimize the loss and improve its performance.
- update each of the parmeters by subtracting their gradient multipled by a small learning rate parmeter

In this project, we use the Adam algorithm wuth default parameters to update our model .

- there any many types of optimizer algorithms, but here we use the Adam as a starting off point

Criterion: Pytorchs name for a loss/error function

- this function takes in models predictions with actual labels and then computes for loss/error with its current parameters
- Cross Entropy Loss both computes the softmax activation function on supplied predictions as well as actual loss via negative log likeihood

magine you're playing a guessing game with your MLP model. There are 10 different things you can guess (represented by the 10 numbers the model outputs).

1. **The Model's Guesses:** The model gives you 10 numbers, but these aren't just guesses - they represent confidence levels. Higher numbers mean the model is more confident something is the answer.
2. **Turning Guesses into Probabilities:** But you need real probabilities (numbers between 0 and 1 that add up to 1) to calculate the score. That's where softmax comes in. It takes those 10 confidence levels and transforms them into a proper probability distribution.
3. **Negative Log Likelihood (NLL): The Scoring System:** Now, you tell the model the correct answer (the label). NLL scores how bad the model's guess was for that answer. The higher the model's confidence in the correct answer (reflected in the probability), the lower the score (better guess).

Why Probabilities?

Think of NLL as points you lose for wrong guesses. Imagine the model guesses something completely wrong (really low probability for the right answer). In that case, it gets a high NLL score (lots of points lost). But if the model is confident about the right answer (high probability), it gets a low NLL score (fewer points lost). This way, NLL encourages the model to learn to output high probabilities for the correct answers.

Analogy Breakdown:

- Your Model's Guesses: Confidence levels (like saying "I'm 80% sure it's number 3")
- Softmax: Turns those guesses into real probabilities (like percentages that add up to 100%)
- Label: The correct answer you reveal
- Negative Log Likelihood (NLL): Points you lose for bad guesses (based on how confident the model was in the wrong answer)

By minimizing NLL (getting a lower score), the model learns to become better at guessing the right answer (increasing the probability for the correct class).

Briefly, the softmax function is:

$$\text{softmax}\left(\mathbf{x}\right) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

This turns out 10 dimensional output, where each element is an unbounded real number, into a probability distribution over 10 elements. That is, all values are between 0 and 1, and together they all sum to 1.

Why do we turn things into a probability distribution? So we can use negative log likelihood for our loss function, as it expects probabilities. PyTorch calculates negative log likelihood for a single example via:

$$\text{negative log likelihood}\left(\hat{\mathbf{y}}, y\right) = -\log\left(\text{softmax}(\hat{\mathbf{y}})[y]\right)$$

$\hat{\mathbf{y}}$ is the $\mathbb{R}^{10}$ output, from our neural network, whereas y is the label, an integer representing the class. The loss is the negative log of the class index of the softmax. For example:

$$\hat{\mathbf{y}} = [5, 1, 1, 1, 1, 1, 1, 1, 1, 1]$$

$$\text{softmax}\left(\hat{\mathbf{y}}\right) = [0.8585, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157]$$

If the label was class zero, the loss would be:

$$\text{negative log likelihood}\left(\hat{\mathbf{y}}, 0\right) = -\log(0.8585) = 0.153\ldots$$

If the label was class five, the loss would be:

$$\text{negative log likelihood}\left(\hat{\mathbf{y}}, 5\right) = -\log(0.0157) = 4.154\ldots$$

So, intuitively, as your model's output corresponding to the correct class index increases, your loss decreases.

Now, the training loop is:

This will:

- put our model into `train` mode
- iterate over our dataloader, returning batches of (image, label)
- place the batch on to our GPU, if we have one
- clear the gradients calculated from the last batch

- pass our batch of images, `x`, through to model to get predictions, `y_pred`
- calculate the loss between our predictions and the actual labels
- calculate the accuracy between our predictions and the actual labels
- calculate the gradients of each parameter
- update the parameters by taking an optimizer step
- update our metrics

Some layers act differently when training and evaluating the model that contains them, hence why we must tell our model we are in "training" mode. The model we are using here does not use any of those layers, however it is good practice to get used to putting your model in training mode.

The evaluation loop is similar to the training loop. The differences are:

- we put our model into evaluation mode with `model.eval()`
- we wrap the iterations inside a `with torch.no_grad()`
- we do not zero gradients as we are not calculating any
- we do not calculate gradients as we are not updating parameters
- we do not take an optimizer step as we are not calculating gradients

`torch.no_grad()` ensures that gradients are not calculated for whatever is inside the `with` block. As our model will not have to calculate gradients, it will be faster and use less memory.

After training the model we focus on evaluating its accuracy, and perform an exploritory analysis on it:

- where does the model go wrong? Are they believable and reasonable mistakes?

- This helps us gain confidence in the process instead of focusing on just the results

Imagine you have a bunch of friends and you want to take a group photo, but your room is messy and cluttered. Here's how PCA (Principal Component Analysis) can help:

1. **Messy Data:** Think of your messy room as your data. You have information about the location of toys, books, clothes (all features), but it's all jumbled up.
2. **Finding the "Main Messiness":** PCA acts like a super organized friend. It analyzes the mess (data) and figures out the main direction of clutter (the direction with the most spread-out stuff).
3. **Taking a Simplified Photo:** Now, imagine taking a picture of the room from the side that shows the most clutter. This simplified picture captures the "main messiness" (the biggest trend in the data).
4. **Multiple Messy Directions:** If your room is messy in many ways (toys everywhere, clothes on the floor, books on chairs), PCA might find multiple "main messiness" directions. It would take several simplified pictures from these different directions to capture the overall messiness.

Key Points:

- PCA helps you find the most important directions of variation in your data (like the "main messiness" in the room).
- It reduces the number of features by focusing on these important directions, creating a simplified version of your data.
- This can be useful for visualization (like the simplified pictures) or for machine learning models that work better with less cluttered data.

Remember: PCA doesn't throw away information, it just focuses on the most important parts!