ENPM703- Assignment-2

Part1: Fully Connected Neural Net

Architecture

Affine: Each layer, starts with an affine transformation, where the input features are multiplied by a weight matrix and a bias vector is added. This results in a linear combination of the input, transforming it into a different feature space.

RELU: The ReLU activation introduces non-linearity by passing the input directly if it's positive and outputting zero if it's negative. This non-linear activation is essential for enabling the network to learn complex, non-linear patterns in the data.

The structure of affine → RELU is repeated for the first L - 1 layers, meaning these layers share the same configuration. This repetition of layers allows the network to progressively transform and refine the data across several hidden layers, learning increasingly abstract representations at each stage.

Final Layer: For the final layer, another affine transformation is applied, followed by the softmax function. The affine layer computes the scores of each class and softmax converts the resulting scores into class probabilities, ensuring the output represents a proper probability distribution across the possible classes.

Optimizers

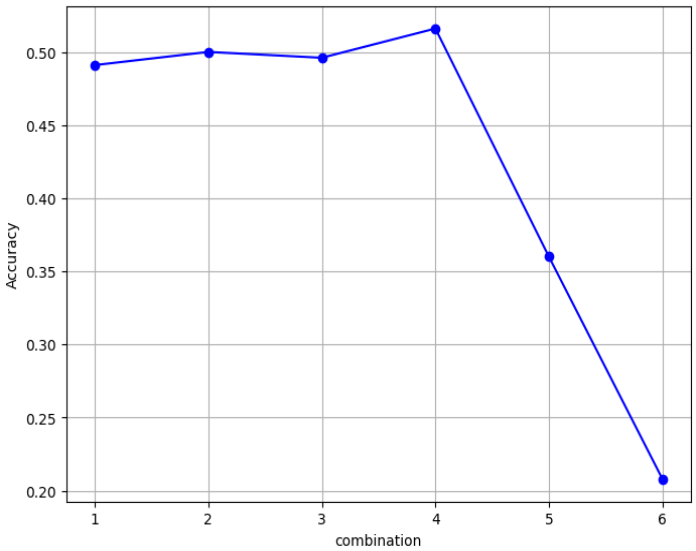
1. **SGD + Momentum**: In this method, weights are updated based on the velocity of the gradient rather than directly using the gradient. By adding a fraction of the previous update to the current one, it helps smooth the update path, preventing the optimization from getting stuck in local minima and speeding up progress in areas with small gradients.
2. **RMSProp**: RMSProp modifies the learning rate for each parameter by using a moving average of recent gradients. It maintains a moving average of squared gradients to normalize updates, which prevents large oscillations and stabilizes the training process.
3. **Adam**: Adam incorporates the benefits of both SGD with momentum and RMSProp. It tracks an exponential moving average of both the gradients (first moment) and squared gradients (second moment), enabling it to adapt the learning rate for each parameter. Additionally, it includes bias correction to address initial overshoots during the early stages of training.

Hyperparameters Tuning

Learning Rate: The learning rate dictates the magnitude of the steps taken during optimization; a higher learning rate may accelerate convergence but risks overshooting the minimum, whereas a lower learning rate allows for more precise updates but can hinder the training process.

Weight Scale: Weight scaling affects to the initialization of weights in a neural network, weights that are excessively small can delay convergence due to vanishing gradients, while

weights that are excessively large can result in extreme gradients, causing instability in training.



1: [‘lr:2e-3’, ‘weight_scale: 5e-2'],
2: [‘lr: 2e-3', ‘weight_scale: 1e-2'],
3: [‘lr: 1e-3', ‘weight_scale: 5e-2'],
4: [‘lr: 1e-3', ‘weight_scale: 1e-2'],
5: [‘lr:1e-5', ‘weight_scale: 5e-2'],
6: [‘lr:1e-5, ‘weight_scale: 1e-2']

ENPM703- Assignment-2

Part2: Batchnorm

Architecture

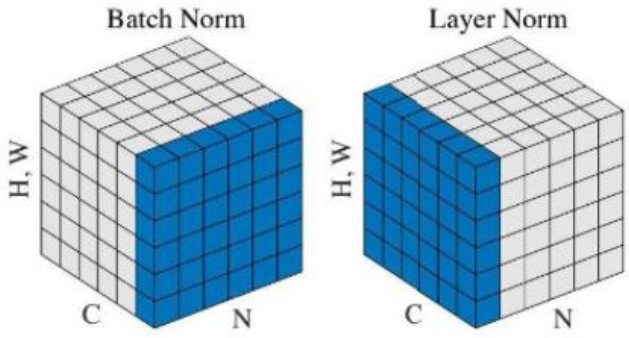
Affine: Each layer starts with an affine transformation, multiplying the input by a weight matrix and adding a bias, resulting in a linear combination that transforms the input into a new feature space.

BN: Normalization layers stabilize and speed up training by adjusting the inputs to each layer, making the activations more consistent. Batch Normalization normalizes over the entire mini-batch, while LayerNorm normalizes across the features of each individual data point. LayerNorm is often used when BN is not suitable, like in models with small batches.

ReLU: The ReLU activation adds non-linearity, passing positive inputs unchanged and outputting zero for negatives, allowing the network to learn complex, non-linear patterns.

This layer is repeated for the first L−1 layers, refining and abstracting the data through multiple hidden layers. The last affine layer computes class scores, and the softmax function converts them into probabilities, ensuring a proper distribution across classes.

Batchnorm



Ref: Zaki, George. Module 07B: Training Neural Networks Part 2. University of Maryland.

Batch Normalization normalizes across the whole mini-batch by calculating the mean and variance for each feature from all the samples in the batch. This helps reduce internal covariate shift. But, it relies on the batch size, which can make it less effective when the batches are small.

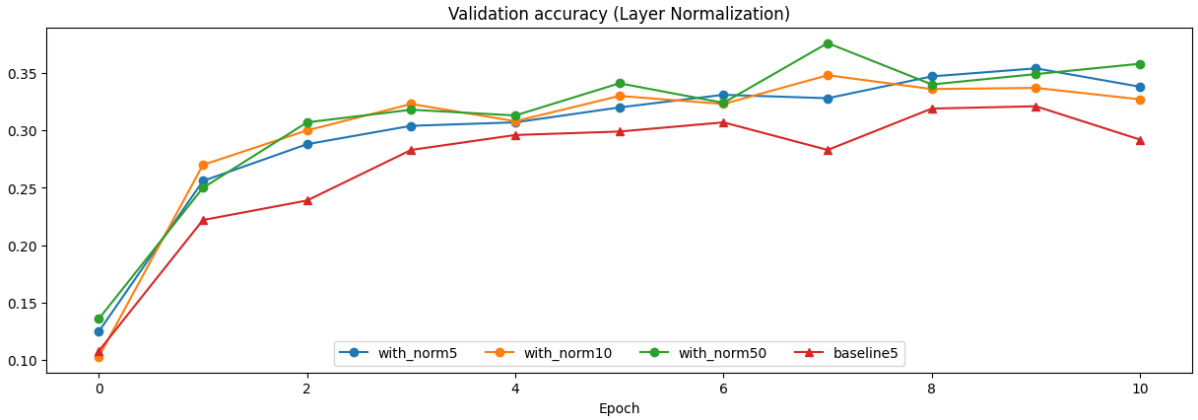
We also compare the results of vanilla batchnorm and simplified batch norm. Here, simplified batchnorm gives better results since it reduces the number of calculations and simplifies the operations required to compute gradients especially when processing large batches of data during training.

Layernorm

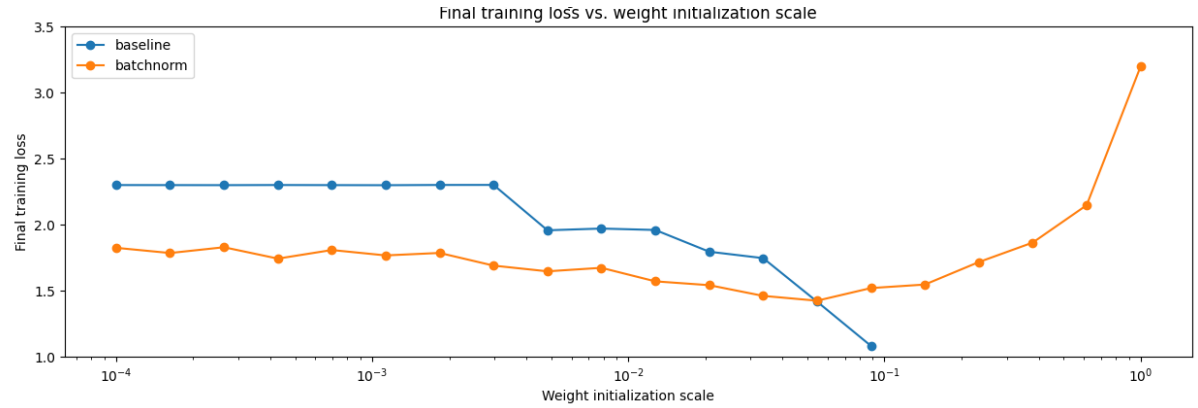
Layer Normalization normalizes the features of a single data point instead of the whole batch. This makes it independent of batch size, making it better suited for tasks when using small batches.

Why batch norm works?

By calculating the mean and variance for each feature across the mini-batch, BatchNorm helps keep the inputs to the activation functions at a consistent distribution which prevents problems like vanishing gradients. This normalization reduces internal covariate shift, where the distribution of inputs to the layers change during training helping the model learn better. BatchNorm also adds learnable parameters to scale and shift the normalized outputs, allowing the network to adjust the features dynamically. As a result, BatchNorm often results in fast convergence, better performance, and ability to use higher learning rates.



- Also having larger batch size corresponds to improved accuracy since it provides a better statistical representation of the data.



- A general trend is batchnorm is less sensitive to weight initialization than the baseline model.

ENPM703- Assignment-2

Part4: Dropout

Dropout

Dropout is a regularization method used to avoid overfitting, where the model fits the training data too well. By randomly turning off certain neurons during training, dropout helps the model learn better features that generalize well, improving its ability to handle unseen data.

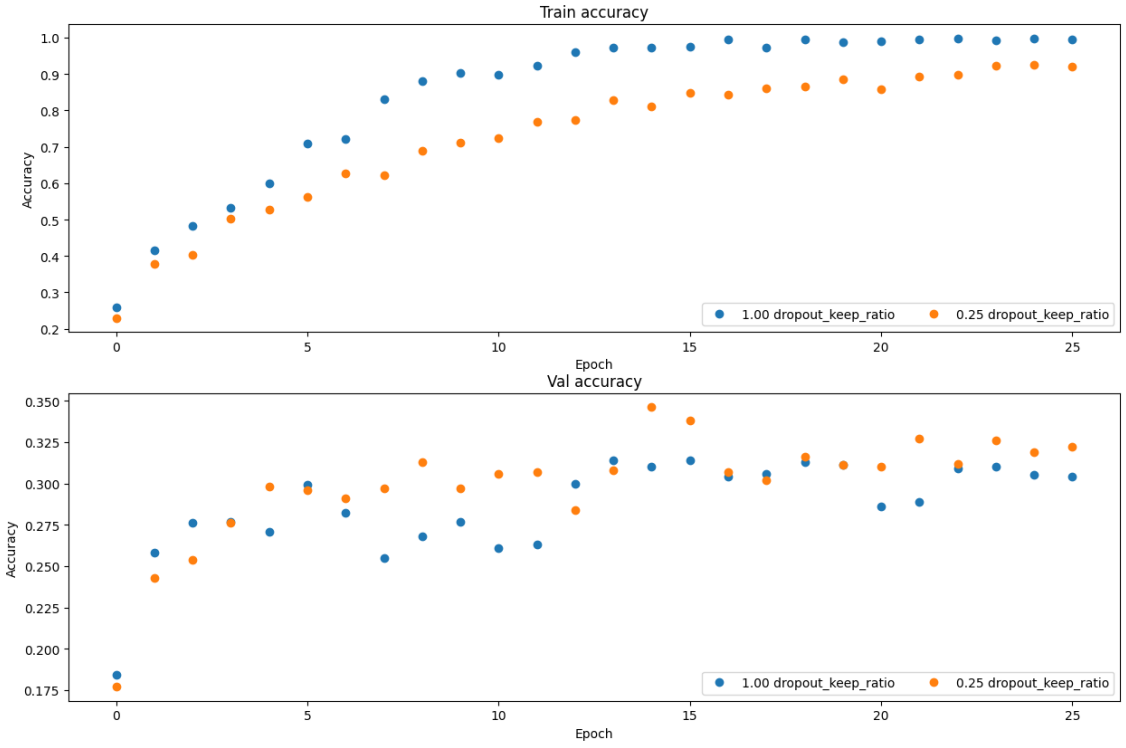
How Dropout Works

During training, dropout randomly deactivates some neurons in a layer with a certain probability 1-p, where p is the keep probability. This means only part of the neurons contribute to the forward pass, making the network less dependent on specific neurons.

Vanila vs Inverted Dropout

- **Vanilla dropout** is the basic form of dropout where, during training, some neurons in a layer are randomly set to zero based on the dropout probability 1-p. In vanilla dropout, the output values of the active neurons aren't changed during training. During prediction, the outputs need to be scaled by p to match what the model saw during training.
- **Inverted dropout** improves on vanilla dropout by scaling the active neurons' outputs by 1/p during training. This keeps the output level consistent whether or not dropout is applied. During prediction, no scaling or dropout is needed because the training already adjusted the outputs. This makes implementation easier and quicker during prediction.

Regularization Experiment



Without dropout, keeping the ratio at 1.0, the model quickly reaches high training accuracy, close to 1.0. However, the validation accuracy is relatively low and increases slowly, indicating overfitting. The model learns the training data well but struggles to generalize to new data.

With dropout, keeping the ratio at 0.25, training accuracy increases more slowly and remains lower, as dropout encourages the model to depend less on specific neurons, preventing overfitting. The validation accuracy is generally higher than without dropout, showing that the model generalizes better to new data when dropout is used, even if training accuracy is lower.

In summary, dropout enhances generalization by reducing overfitting, as shown by higher validation accuracy despite lower training accuracy.

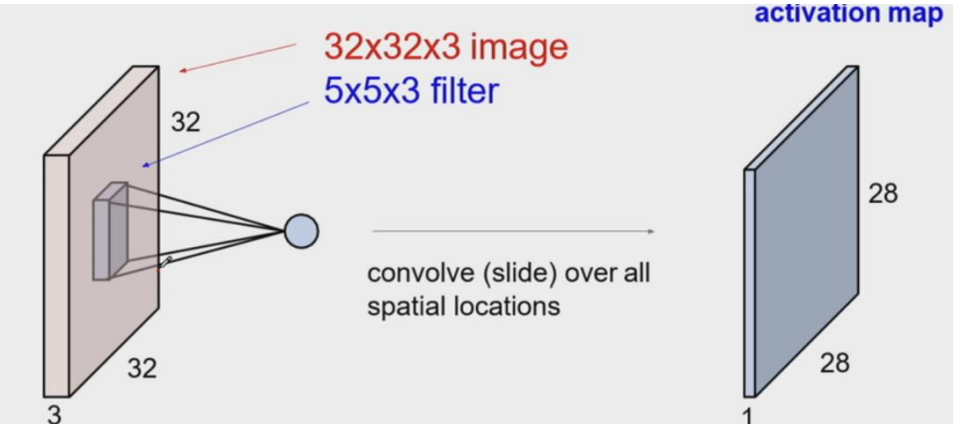
Effects of Dropout

- Dropout reduces overfitting by forcing the network to learn using different paths, making the model more robust and less dependent on individual neurons. This encourages redundancy and prevents neurons from co-adapting.
- However, if the dropout rate is too high (low p), the model may underfit, meaning it won't learn effectively. It's important to tune p to find the right balance between reducing overfitting and keeping enough learning capacity.

ENPM703- Assignment-2

Part3: Convolution NN

Convolution NN Architecture



Ref: Zaki, George. module06-convolution-layers. University of Maryland.

A Convolutional Neural Network (CNN) is a deep learning model that use convolutional layers to automatically detect and learn spatial feature hierarchies from an input image.

Convolution Layer: The convolution layer uses filters (kernels) that move over the input image, computing dot products between the filter and local slices of the image. This operation captures important spatial features like edges, textures, and shapes.

Activation Function: Following convolution, an activation function (mostly ReLU) is applied to introduce non-linearity.

The implementation of convolution layer is not efficient since the computation is done by 4 nested for loops, but is compared with an optimized implementation which is approximately 500x faster.

Maxpooling

Max Pooling is a form of down-sampling used in CNNs to reduce the spatial dimensions of feature maps. This operation is applied independently across each channel in the feature map. Max pooling divides the input into non-overlapping regions (2x2) and selects the maximum value from each region. For example, a 2x2 pooling operation would reduce a 4x4 input to a 2x2 output.

Max pooling reduces the number of parameters and computational load, while preserving the most essential features. By keeping only the maximum value, max pooling increases the network's robustness to small translations or distortions in the input.

Spatial Normalization

Spatial Normalization (Batch Normalization for spatial data) normalizes activations across channels within a mini-batch, enhancing both training stability and speed in CNNs. For each channel, the mean and variance are calculated across the spatial dimensions and normalize the data. Here the vanilla version of batch normalization function is used by reshaping the inputs accordingly.

Why CNN is better?

CNN are great for image processing tasks due to their ability to capture spatial feature using convolutional and pooling layers.

- Through pooling layers, CNN gain robustness to shifts in feature positions allowing them to recognize elements like edges or objects regardless of their location within an image. This property is essential for consistent and reliable image processing.
- CNN build feature representations at varying abstraction levels within their layered framework. Initial layers detect basic features such as edges and colors, while deeper layers identify more complex structures like shapes, textures, and complete objects.
- CNN are particularly suited for processing high dimensional inputs, like large images by analyzing spatial regions instead of treating each pixel individually. This design enables CNN to handle images more efficiently, as they don't need to connect every pixel to each node in subsequent layers, unlike traditional neural networks, thus making them highly scalable.

ENPM703- Assignment-2

Part5: Pytorch Implementation

Pytorch

PyTorch is a free, open-source machine learning library created by Facebook's AI Research team. It is based on tensors, which are similar to NumPy arrays, but with extra features for using GPUs, making it ideal for deep learning projects. PyTorch provides a strong framework for building deep learning applications, known for being user-friendly, having dynamic computation graphs, and focusing on performance and flexibility. Its API offers both low-level and high-level tools, making it suitable for a variety of uses.

Sample Functions

Functions in torch.nn.functional:

1. **relu():** Applies the Rectified Linear Unit (ReLU) activation function to each element, adding non-linearity to the model.
2. **cross_entropy():** Calculates the cross-entropy loss between predicted outputs and true labels, often used for classification tasks.
3. **conv2d():** Performs a 2D convolution on input data, applying the weights of a convolutional layer to the input tensor.

Functions in NN Module API:

1. **Linear(in_features, out_features):** Creates a fully connected layer that transforms the input data using a linear function, taking the number of input and output features.
2. **Conv2d(in_channels, out_channels, kernel_size):** Sets up a 2D convolutional layer used in image processing, requiring the number of input channels, output channels, and kernel size.
3. **BatchNorm2d(num_features):** Applies batch normalization to the output from a convolutional layer, helping to stabilize and speed up training by normalizing activations.

Advantages of Pytorch

1. **Flexibility and Ease of Use:**
   - PyTorch has dynamic computation graphs that allow for easy model building and debugging. It features a user-friendly interface and straightforward syntax, making it accessible for beginners and efficient for experienced users. Its flexible APIs support both low-level adjustments and high-level quick development.
2. **Performance and Efficiency:**
   - With built-in GPU support, PyTorch allows for faster training of deep learning models. It also makes it easy to save and load models, simplifying the process of deploying and sharing them.

NN Module API and Sequential API

The **NN Module API** is part of PyTorch that offers classes and functions for creating neural networks. It includes a variety of pre-defined layers, such as convolutional, recurrent, and linear layers, as well as tools for setting up loss functions and optimizers.

The **Sequential API** provides an easy way to build models by stacking layers in order, making it simple to construct models without needing to create a custom forward method unless necessary.

Open Ended Model

- Model includes two convolutional layers, each followed by batch normalization, ReLU activation, and max pooling, which help to extract and downsample features from the input images.
- The first convolutional layer has 64 channels, with a kernel size of 5, and includes padding of 2. The second layer has 32 channels, a kernel size of 3, and padding of 1.
- After extracting features, the model flattens the output and feeds it through three fully connected layers with 256, 128, and 10 output units, respectively, using ReLU activation and dropout to help prevent overfitting.
- The model is trained using the Adam optimizer with a learning rate of 1e-3 and a weight decay of 1e-4 to enhance generalization.

Tunning the Model

- **Lr:** 1e-3 was the sweet spot of getting optimal results in optimal time. increasing the value made the accuracy to dip down.
- **Weight Decay:** Values of range 1e-4 to 5e-4 produced optimal accuracies. Having this parameter will prevent overfitting of networks.
- **Filter Size:** Values of channel_1 = 64, channel_2 = 32 means more channels which help the model capture a wider range of features, prevent vanishing gradients, and retain more information during backpropagation, improving generalization on unseen data.
- **No. of Filters:** Both size (3,1) and (5,3) produced similar results but combination (5,1) performed poorly. In later, large filter size and a small filter size might not effectively extract useful details.
- **Fcc Dimension:** Fcc output dimensions 128->64->10 and 256->64->10 produced similar results but dimension 256->128->10 performed great. The added layer might help the model discover more useful patterns and enhance its ability to generalize to new data.