Improving Neural Network Performance

1. Vanishing Gradients / Exploding Gradients

When training deep neural networks, gradients can either become too small (vanishing gradients) or too large (exploding gradients), making training difficult. Vanishing gradients prevent early layers from learning effectively, while exploding gradients lead to instability.

a. Activation Functions

- **Sigmoid and Tanh** cause vanishing gradients due to their limited output range. Gradients become very small as they propagate backward through layers, making deep networks difficult to train.
- **ReLU (Rectified Linear Unit)** mitigates vanishing gradients by outputting zero for negative values and the identity function for positive values. However, it may suffer from dead neurons (neurons stuck at zero activation).
- **Leaky ReLU, Parametric ReLU (PReLU), and Swish** help reduce dead neurons while maintaining non-linearity, improving gradient flow.

b. Weight Initialization

- **Xavier/Glorot Initialization**: Helps balance activation distribution across layers (used with Sigmoid/Tanh) by setting weights based on the number of input and output neurons.
- **He Initialization**: Specifically designed for ReLU and its variants, it uses a slightly higher variance to prevent vanishing gradients.
- **Orthogonal Initialization**: Preserves variance across layers and improves training stability, particularly for deep networks.

2. Overfitting

Overfitting occurs when a model performs well on training data but poorly on new, unseen data. This happens when the model learns noise instead of meaningful patterns.

a. Reduce Model Complexity, Increase Data

- **Simpler architectures**: Reducing unnecessary depth/width in the model minimizes overfitting.
- **Data augmentation**: Techniques like rotation, cropping, flipping, and noise addition artificially increase dataset size, improving generalization.
- **Collecting more data**: A larger, diverse dataset helps prevent overfitting and improves real-world performance.

b. Dropout Layers

- **Dropout** randomly deactivates a percentage of neurons during training, preventing the model from relying too much on specific neurons and promoting generalization.
- Common dropout rates: **0.2 - 0.5**, depending on layer depth.

c. Regularization

- **L1 Regularization (Lasso)**: Encourages sparsity by penalizing large absolute values of weights, making some weights zero.
- **L2 Regularization (Ridge/Weight Decay)**: Penalizes large weight values, reducing overfitting while retaining all features.
- **Elastic Net**: Combines L1 and L2 regularization to leverage the benefits of both.

d. Early Stopping

- **Monitors validation loss**: Stops training when validation loss stops decreasing to prevent excessive training.
- Saves computational resources and prevents overfitting.

3. Normalization

Normalization improves training stability by ensuring that input data and activations have a controlled distribution. This helps speed up training and enhances model performance.

a. Normalizing Inputs

- **Standardization (zero mean, unit variance)** improves training stability.
- Helps in faster convergence and stable training.

b. Batch Normalization

- Normalizes activations across mini-batches, reducing internal covariate shift.
- Enables the use of higher learning rates and faster convergence.

c. Normalizing Activations

- **Layer Normalization**: Normalizes across features rather than batches, useful for NLP tasks.
- **Instance Normalization, Group Normalization**: Applied in computer vision models, especially in style transfer tasks.

4. Optimizers

Optimizers are algorithms that adjust the network's weights based on gradients to minimize loss. Different optimizers improve convergence speed and stability.

a. Momentum

- Uses exponentially weighted moving averages of past gradients to accelerate updates.
- Helps in faster convergence by reducing oscillations and stabilizing weight updates.
- Commonly used with stochastic gradient descent (SGD) to improve performance.

b. Adagrad

- Adapts learning rate per parameter based on past squared gradients, making large updates for infrequent parameters and smaller updates for frequently changing ones.
- Works well for sparse data but suffers from diminishing learning rates over time.
- Not ideal for deep networks as learning rate eventually becomes too small.

c. RMSprop

- Modifies Adagrad by introducing a moving average of squared gradients, preventing learning rate from decreasing too quickly.
- Maintains an exponentially decaying average of past squared gradients.
- Works well for recurrent networks and adaptive learning scenarios.

d. Adam (Adaptive Moment Estimation)

- Combines Momentum and RMSprop by computing first-order momentum (mean of past gradients) and second-order momentum (mean of squared gradients).
- Adaptive learning rates per parameter make it effective for noisy, sparse, and non-stationary problems.
- Default parameters ($\beta 1 = 0.9$, $\beta 2 = 0.999$) work well for most deep learning tasks.

5. Learning Rate Scheduling

Adjusting the learning rate during training helps optimize convergence. A too-high learning rate causes instability, while a too-low rate slows convergence.

- **Step Decay**: Reduces learning rate at fixed intervals.
- **Exponential Decay**: Multiplies the learning rate by a factor after every epoch.
- **Cyclical Learning Rates**: Oscillates between a minimum and maximum learning rate.
- **Warmup**: Starts with a low learning rate and gradually increases before applying decay.

6. Hyperparameter Tuning

Hyperparameters control the structure and training process of the model. Finding optimal values improves performance.

a. Number of Hidden Layers

- Deeper networks learn more complex representations.
- Too many layers can cause vanishing gradients (use skip connections like in ResNets to mitigate).

b. Nodes per Layer

- More nodes capture richer features but increase computation cost.
- Finding the optimal number requires experimentation and tuning.

c. Batch Size

- **Small batches**: More generalization but introduces noise in updates.
- **Large batches**: Faster convergence but risks poor generalization.
- **Mini-batch (32-256 samples)**: A balance between efficiency and generalization.

7. Additional Topics

a. Data Augmentation

- Improves generalization, particularly in vision tasks.
- Examples: Rotation, flipping, cropping, adding noise.

b. Transfer Learning

- Using pre-trained models like ResNet, VGG, BERT for faster convergence on new tasks.
- Fine-tuning a small part of the network can help adapt it to a specific dataset.

c. Gradient Clipping

- Limits the maximum gradient values to prevent exploding gradients.
- Useful for training recurrent neural networks (RNNs, LSTMs, GRUs).

d. Model Ensembling

- Combines multiple models to improve performance.
- **Bagging** (e.g., Random Forest), **Boosting** (e.g., XGBoost), and **Stacking** are common ensemble methods.

Summary

- **Vanishing/Exploding gradients** can be mitigated using proper activation functions and weight initialization.
- **Overfitting** can be prevented using regularization, dropout, early stopping, and data augmentation.
- **Normalization** helps stabilize training and improve convergence.
- **Optimizers and learning rate schedules** significantly impact performance.
- **Hyperparameter tuning** is necessary for model efficiency and generalization.
- **Additional techniques** like transfer learning and ensembling can further improve performance.

These techniques collectively contribute to optimizing deep neural networks for better accuracy, efficiency, and generalization.