



Module 2


Training deep models





Syllabus

Introduction, setup and initialization issues, Vanishing and exploding gradient problems, Optimization techniques - Gradient Descent (GD), Stochastic GD, GD with momentum, GD with Nesterov momentum, AdaGrad, RMSProp, Adam. Regularization Techniques - L1 and L2 regularization, Early stopping, Dataset augmentation, Parameter tying and sharing, Ensemble methods, Dropout.



Introduction

In the context of deep learning and neural networks, several challenges arise during training, especially as the complexity of models increases. Understanding these issues is critical to effectively train networks and ensure good performance.

Key concepts covered here include:

- Proper initialization of weights.
- Addressing gradient-related problems.
- Efficient optimization techniques.
- Robust regularization methods.

Setup and Initialization Issues

Symmetry Breaking

Initializing all weights to the same value leads to identical outputs and gradients.

Prevented by using random initialization.

Scale of Weights

Too small: Slow learning due to small gradients.

Too large: Instability during training.

Initialization Methods

Random Initialization: Small random values.

Zero initialization

He Initialization: For ReLU activations ($\sigma = \sqrt{2}$).

Xavier Initialization: For sigmoid/tanh ($\sigma = \sqrt{1/2}$).

Setup and Initialization Issues

Proper weight initialization in a neural network can significantly enhance the performance of the network. If the starting weights are too small, then the signal shrinks as it passes through each layer until it's too tiny to be useful. If the weights are too large, then the signal grows exponentially as it passes through each layer until it's too large to be processed. Thus, the choice of weight initialization can play a significant role in how quickly a neural network can learn and converge to the optimal solution.

Setup and Initialization Issues

There are several weight initialization techniques:

Random Initialization: Simple but can lead to poor convergence.

Zero Initialization: Ineffective as it causes symmetry problems.

Xavier Initialization (also known as Glorot Initialization): This technique is named after Xavier Glorot, one of the first authors of a paper that introduced this concept. The idea behind Xavier initialization is to make the variance of the outputs of a neuron to be equal to the variance of its inputs. Xavier Initialization sets the initial weights of the neural network by drawing them from a distribution with zero mean and specific variance. The variance is $1/n$, where n is the number of input units. This distribution can be either a uniform distribution or a normal distribution.

He Initialization: This technique was proposed in a 2015 paper by Kaiming He, hence the name. It's a variant of Xavier initialization, specifically designed for neural networks with ReLU activation functions. In He initialization, the variance of the distribution is $2/n$, where n is the number of input units. This adjustment accounts for the fact that the ReLU activation function halves the variance of the output, due to only passing through positive inputs. Like Xavier initialization, this distribution could be either a uniform distribution or a normal distribution.

Vanishing and Exploding Gradient Problems

Vanishing Gradients

Vanishing gradient problem is a phenomenon that occurs during the training of deep neural networks, where the gradients that are used to update the network become extremely small or "vanish" as they are backpropogated from the output layers to the earlier layers.

During the training process of the neural network, the goal is to minimize a loss function by adjusting the weights of the network.

The backpropagation algorithm calculates these gradients by propogating the error from the output layer to the input layer.

The vanishing gradient problem causes the gradients to shrink. But, if a gradient is small, it won't be possible to effectively update the weights and biases of the initial layers with each training session.

These initial layers are vital for recognizing the core elements of the input data, so, if their weights and biases are not properly updated, it is possible that the entire network could be inaccurate.

Gradients shrink exponentially, especially with sigmoid/tanh activations.

Solutions:

Use ReLU/Leaky ReLU activations.

Residual Networks (ResNets).

Batch Normalization

Vanishing and Exploding Gradient Problems

Exploding Gradients

On the contrary, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the exploding gradients problem.

Gradients grow uncontrollably, causing instability.

Solutions:

Gradient clipping.

Proper weight initialization.

Architectures with effective gradient flow (e.g., LSTMs).

Optimization Techniques

1. Gradient Descent (GD)

Updates weights using the entire dataset.

Pros: Converges smoothly.

Cons: Computationally expensive for large datasets.

2. Stochastic Gradient Descent (SGD)

Updates weights using a single sample per iteration.

Pros: Faster updates, works well for large datasets.

Cons: Noisy convergence.

Optimization Techniques

stochastic gradient descent takes a much more noisy path than the gradient descent algorithm when addressing optimizers in deep learning. Due to this, it requires a more significant number of iterations to reach the optimal minimum, and hence, computation time is very slow. To overcome the problem, we use stochastic gradient descent with a momentum algorithm.

What the momentum does is helps in faster convergence of the loss function. Stochastic gradient descent

oscillates between either direction of the gradient and updates the weights accordingly. However, adding a

fraction of the previous update to the current update will make the process a bit faster. One thing that

should be remembered while using this algorithm is that the learning rate should be decreased with a high momentum term.

Optimization Techniques

3. GD with Momentum

Gradient descent with momentum is an optimization algorithm that helps accelerate the convergence of gradient descent by adding a momentum term to the weight update. The momentum term is based on the previous weight update and helps the algorithm build momentum as it descends the loss function.

Gradient descent with momentum address disadvantages of GD. It works by adding a fraction of the previous weight update to the current weight update so that the optimization algorithm can build momentum as it descends the loss function. This can help the algorithm escape from local minima and saddle points and can also help the algorithm converge faster by avoiding oscillations.

Momentum helps to,

- Escape local minima and saddle points

- Aids in faster convergence by reducing oscillations

- Smooths out weight updates for stability

- Reduces model complexity and prevents overfitting

- Can be used in combination with other optimization algorithms for improved performance.

Optimization Techniques

Here is the math behind momentum-based gradient descent:

Let's say we have a set of weights w and a loss function L , and we want to use gradient descent to find the weights that minimize the loss function. The standard gradient descent update rule is:

$$w = w - \alpha * \text{gradient}$$

Where α is the learning rate, and the gradient is the gradient of the loss function to the weights.

To incorporate gradient descent with momentum into this update rule, we can add a momentum term v that is based on the previous weight update:

$$v = \rho * v + \alpha * \text{gradient}$$

$$w = w - v$$

Where ρ is the momentum hyperparameter (typically set to 0.9).

The momentum term v can be interpreted as the "velocity" of the optimization algorithm, and it helps the algorithm build momentum as it descends the loss function. This can help the algorithm escape from local minima and saddle points and can also help the algorithm

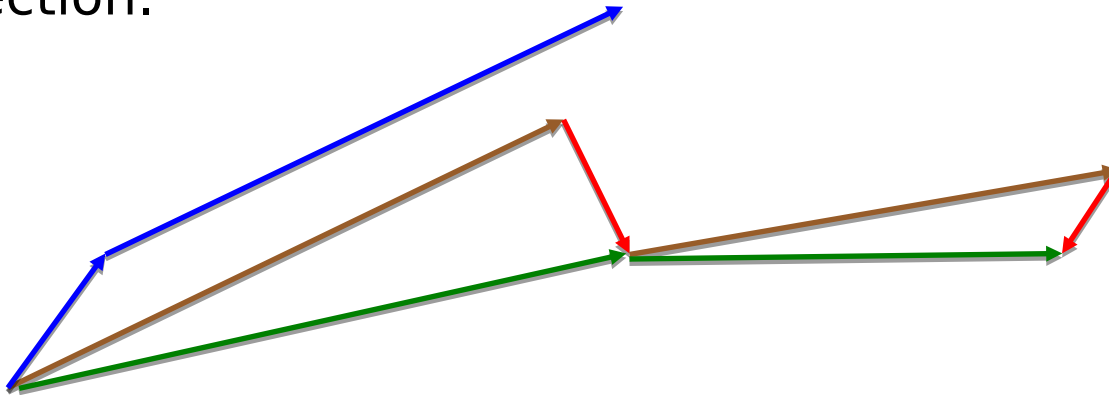
Optimization Techniques

4. GD with Nesterov Momentum

- The standard momentum method **first** computes the gradient at the current location and **then** takes a big jump in the direction of the updated accumulated gradient.
- Ilya Sutskever (2012 unpublished) suggested a new form of momentum that often works better.
 - Inspired by the Nesterov method for optimizing convex functions.
- **First** make a big jump in the direction of the previous accumulated gradient.
- **Then** measure the gradient where you end up and make a correction.
 - Its better to correct a mistake **after** you have made it!

A picture of the Nesterov method

- **First** make a big jump in the direction of the previous accumulated gradient.
- **Then** measure the gradient where you end up and make a correction.



brown vector = jump,
accumulated gradient

red vector = correction,

green vector =

blue vectors = standard momentum

Optimization Techniques

5. AdaGrad

Adaptive gradient descent algorithm

Adapts the learning rate for each parameter based on past gradients.

it uses different learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training.

Benefit: Works well for sparse features.

Issue: Learning rate becomes too small over time.

$$w_t = w_{t-1} - \eta'_t \frac{\partial L}{\partial w(t-1)}$$

the $\alpha(t)$ denotes the different learning rates at each iteration, n is a constant

$$\eta'_t = \frac{\eta}{\text{sqrt}(\alpha_t + \epsilon)}$$

Optimization Techniques

6. RMSProp (Root Mean Square)

RMSProp is an adaptive learning rate optimization algorithm designed to improve the performance and speed of training deep learning models. It is a variant of the gradient descent algorithm, which adapts the learning rate for each parameter individually by considering the magnitude of recent gradients for those parameters.

$$v_t = \text{decay_rate} * v_{t-1} + (1 - \text{decay_rate}) * \text{gradient}^2$$

$$\text{parameter} = \text{parameter} - \text{learning_rate} * \text{gradient} / (\text{sqrt}(v_t) + \text{epsilon})$$

Benefit: Solves AdaGrad's diminishing learning rate issue.

7. Adam

Adam (Adaptive Moment Estimation) is an optimization algorithm that computes adaptive learning rates for each parameter by combining the advantages of two other methods: Adagrad and RMSprop.

The algorithm maintains exponentially decaying averages of past gradients (first moment) and past squared gradients (second moment), which are used to compute adaptive learning rates. The update rule includes bias correction terms to adjust for initialization effects.

Adam's advantages include fast convergence, robustness to sparse gradients, and suitability for non-stationary objectives.

It is commonly used in various deep learning tasks, including computer vision, natural language processing, and reinforcement learning, due to its efficiency and effectiveness.

Regularization Techniques

Regularization is a technique used to address overfitting by directly changing the architecture of the model by modifying the model's training process. The following are the commonly used regularization techniques:

Early Stopping

Dropout regularization

Data Augmentation

L1 regularization

L2 regularization

Early Stopping

One way to stop a neural network from overfitting is to halt the training process before the network has thoroughly memorized the data set. This widely used technique is called early stopping.

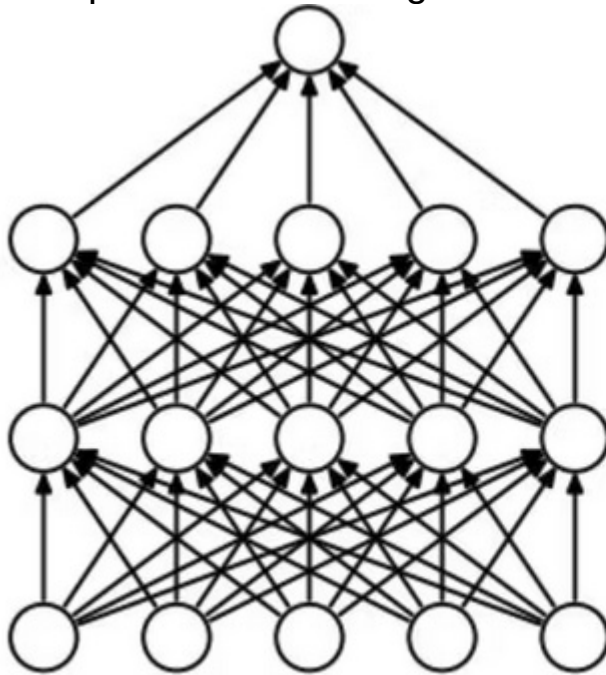
Recall that the process of optimizing a network to find the correct parameters is iterative. If you were to evaluate a model's error on the training and dev set after every training epoch.

Early stopping is a cross-validation strategy in which we keep one part of the training set as the validation set. When we see that the performance on the validation set is getting worse, we immediately stop the training on the model.

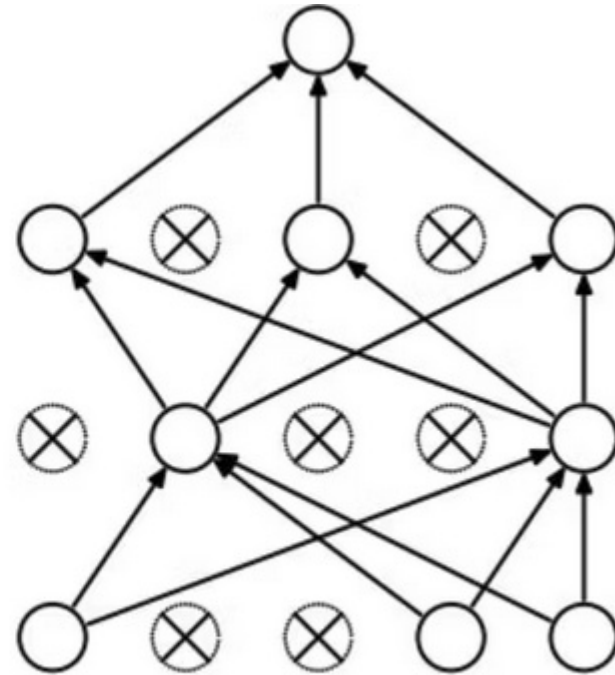


Drop Out

Dropout regularization is the technique in which some of the neurons are randomly disabled during the training such that the model can extract more useful robust features from the model. This prevents overfitting.



(a) Standard Neural Net



(b) After applying dropout.

Drop Out

The following are the characteristics of dropout regularization:

Dropout randomly disables some percent of neurons in each layer. So for every epoch, different neurons will be dropped leading to effective learning.

Dropout is applied by specifying the 'p' values, which is the fraction of neurons to be dropped.

Dropout reduces the dependencies of neurons on other neurons, resulting in more robust model behavior.

Dropout is applied only during the model training phase and is not applied during the inference phase.

When the model receives complete data during the inference time, you need to scale the layer outputs 'x' by 'p' such that only some parts of data will be sent to the next layer. This is because the layers have seen less amount of data as specified by dropout.

L1 Regularization

Essentially, the L1 regularizer searches for parameter vectors that minimize the parameter vector's norm (the length of the vector). The main issue here is how to best optimize the parameters of a single neuron, a single layer neural network generally, and a single layer feed-forward neural network specifically.

Since L1 regularization offers sparse solutions, it is the favored method when there are many features. Even so, we benefit from the computational advantage since it is possible to omit features with zero coefficients.

The mathematical represent

$$\text{Cost function} = \text{Loss} + \frac{\lambda}{2m} * \sum ||w||$$

Here the lambda is the regularization parameter. Here we penalize the absolute value of the weights and weights may be reduced to zero. Hence L1 regularization techniques come very handily when we are trying to compress the deep learning model.

L2 Regularization

By limiting the coefficient and maintaining all the variables, L2 regularization helps solve problems with multicollinearity (highly correlated independent variables). The importance of predictors may be estimated using L2 regression, and based on that, the unimportant predictors can be penalized.

The mathematical representation for the L2 regularization is:

$$\text{Cost function} = \text{Loss} + \frac{\lambda}{2m} * \sum ||w||^2$$

The regularization parameter, in this case, is lambda. The value of this hyperparameter is generally tweaked for better outcomes. Since L2 regularization leads the weights to decay towards zero (but not exactly zero), it is also known as weight decay.

L1 vs L2

L1 regularization can add the penalty term to the cost function by taking the absolute value of the weight parameters into account. On the other hand, the squared value of the weights in the cost function is added via L2 regularization.

In order to avoid overfitting, L2 regularization makes estimates for the data mean instead of the median as is done by L1 regularization.

Since L2 is a square of weight, it has a closed-form solution; however, L1, which is a non-differentiable function and includes an absolute value, does not. Due to this, L1 regularization requires more approximations, is computationally more costly, and cannot be done within the framework of matrix measurement.