# Assignment-13 : CNNs, Learning Algorithms & Generalization

**Harshvardhan Patidar**
Department of Artificial Intelligence
Indian Institute of Technology Hyderabad
ai24btech11015@iith.ac.in

## 1 Convolutional Neural Networks

Convolutional networks (CNNs) are the neural networks which are designed specifically for processing structured data like images. They are more efficient than fully connected layers due to the following reasons:

- **Local connectivity** : each unit in the hidden layer receives input from only a small, local region of the image. It has two benefits, first it considers the adjacency of the pixels, and also it reduces the computational power requirements as the number of weights required will be a lot lesser than a fully connected network.
- **Spatial Invariance** : Images also have a spatial invariance, i.e. is that the features(like edges or textures of objects) can appear anywhere in the image. To consider this, the weights of local connections are constrained to be the same across different parts of the image. The pattern of weights that is being replicated is called a kernel. This allows the model to detect a feature in an image regardless of its location in the image. This process is termed as convolution.
- **Convolution Operation** : This operation applies a kernel, say $mathbfk$ to an input vector $mathbfx$, such that $mathbfz = mathbfx * mathbfk$, which is defined as

  $$z_i = \sum_{j=1}^{l} k_j \, x_{j+i-\frac{(l+1)}{2}}$$

  This reduces the output size in comparison to the input size. If $n$ is the size of the input vector and we apply this process with a stride of $s$, then the output layer has a size of roughly $n/s$. In two dimensions it would be $n/s_x s_y$.
- **Pooling** : It summarises a number of given inputs that are adjacent to each other, and represents them with a single value. It is accomplished commonly in 2 ways:
  - **Average Pooling** : As the name implies, it simply calculates the average of the given inputs. It is analogous to convolution with a kernel vector $\mathbf{k} = [1/l, \ldots, 1/l]$. This simply downsample the image by a factor of $l$, i.e. if it earlier consumed $10l$ pixels it will now consume 10 pixels.
  - **Max Pooling** : It simply takes the max values of the given inputs. It is thus a form of downsampling, which preserves the important feature of the image as the features are most probably there in the max value.
- **Tensor Operations** : CNNs are described by using tensors, which are multidimensional arrays. This representation gives computational efficiency. A deep learning package can generate compiled code that is highly optimised for the required operations. These when run on GPUs or TPUs can then take full advantage of their computational power for parallel processing to complete the task faster.

- **Residual Networks** : This is a successful approach to making very deep networks while avoiding the issue of vanishing gradients. This allows the layers to learn perturbations instead of completely replacing the representation of the previous layer, which was the major drawback of other models. The equation is as follows

  $$\mathbf{z}^{(i)} = \mathbf{g}_r^{(i)}\left(\mathbf{z}^{(i-1)} + f(\mathbf{z}^{(i-1)})\right)$$

  Here $\mathbf{g}_r$ is the activation function and $f$ represents the residual function. It is given by

  $$f(\mathbf{z}) = \mathbf{V}\mathbf{g}(\mathbf{W}\mathbf{z})$$

  Here, $\mathbf{V}$ and $\mathbf{W}$ are the learned matrices (with some usual weights added). The particular benefit of this type of function is that even if we set the weight matrix **W** as 0 for some layer, still, the network wouldn't cease to function. It will behave just like that particular layer never existed.

  $$\mathbf{z}^{(i)} = \mathbf{g}^{(i)}\left(\mathbf{z}^{(i-1)}\right) \quad \text{when } f \text{ disappears}$$

## 2 Learning Algorithms

Training a neural network involves adjusting the parameters of the network to minimise some sort of loss function. Gradient descent is the most commonly used technique.

**Gradient Descent** it tries to move the weights in the direction of the steepest decrease in the loss function. The update rule was :

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\nabla_{\mathbf{w}}L(\mathbf{w})$$

Here, $\alpha$ is the learning rate. In the standard gradient descent, we compute $L(\mathbf{w})$ over the entire data set, whereas in stochastic gradient descent, we compute it from a randomly chosen minibatch of $m$ members. It helps reduce computational cost and also its randomness helps avoiding local minima. Some enhancements for optimally training the model are :

- For Real world problems, the data size and the dimensionality of **w** are large, these favour the usage of SGD for updating the weights.
- SGD can take advantage of the parallelism of advanced GPUs and TPUs.
- A gradually decreasing learning rate improves the convergence of the hypothesis towards true function.
- The gradients calculated from minibatch might have high variance near some local or global minimum. It can be handled by using the idea of momentum, in which we keep a running average of the gradients.
- Special attention is also needed towards the numerical instability that might be caused by the overflow or rounding error during the calculations in the program.

In the end, we don't have confirmation that we will always reach to the global or local minimum. We might need to make a very large number of small steps to reach to the optimal solution. Going through these steps not only increases the computational cost, but may also cause the model to overfit the data.

### 2.1 Computing Gradients in Computation Graphs

In neural networks, each node computes some value and passes it to the next node. For gradient calculations, the loss information is back propagated so that gradient descent can be used to update the weights.

Each node calculates the gradient from the nodes to which it supplies data. Suppose node $h$ has a connection to $j$ and $k$. Then the gradient of $L$ w.r.t. $h$ is

$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial h_j} + \frac{\partial L}{\partial h_k}$$

Now we will use chain rule for the gradient which are to be passed back. Lets say its input are nodes $f$ and $g$ , then gradient is:

$$\frac{\partial L}{\partial f_h} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial f_h} \quad \frac{\partial L}{\partial g_h} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial g_h}$$

This process of calculating and passing back the gradient continues till we reach the input layer. Then the final gradient is used to update the weights. This process of gradient descent is computationally efficient as all the gradients for different types of computations of nodes are already predefined and can be converted into highly efficient code.

The only disadvantage of back-propagation is that it requires a large amount of memory to store the traced-back data. The total amount of memory required is proportional to the number of nodes.

### 2.2 Batch Normalization

This technique is used to improve the convergence of the SGD method. Consider some node $z$ in the network, then the value of the node on the m examples is given by $z_1, \ldots, z_m$. Batch normalization replaces each $z_i$ with a $\hat{z}_i$ :

$$\hat{z}_i = \gamma \cdot \frac{z_i - \mu}{\sqrt{\epsilon + \sigma^2}} + \beta$$

where $\mu$ is the mean value of $z$ across the minibatch, $\sigma$ is the standard deviation of $z_1, \ldots, z_m$, $\epsilon$ is a small constant added to prevent division by zero, and $\gamma$ and $\beta$ are learned parameters.

It has two benefits, first is that without it, there might be some loss of information if a layer's weights are too small, such that $\sigma$ tends to zero making it harder to learn. Batch normalization helps prevent this. The second benefit is that it remits us from the task of careful initialization of weights.

## 3 Generalization

It focuses on making the trained model to perform better on the unseen data. As the name suggests we try to 'Generalize' our model so that it better fits the new unseen data. Following are the ways to achieve this

- **Choosing a proper Network Architecture** :t is an important step for good generalization. Each neural network architecture are useful for a particular type of data depending upon their working process. Deeper neural networks usually give better generalization.

  Deep learning models can also produce some unexpected results. One such case is of the adversarial examples, where a slight modification of the input data can lead to misclassification. It is more likely in the case of high dimensional input. For example in an image of a dog, if some of the pixels are modified, not affecting the original view of the picture, the neural network might misclassify it as cat or something else. This is caused by the discontinuous nature of neural networks mapping.
- **Neural Architecture Search** : There are no fixed set of rules yet to select the best network architecture for a neural network. So, choosing the optimal network architecture is in itself a hyperparameter tuning problem. Evolutionary algorithms, hill climbing and bayesian optimization are used for neural architecture search to find the best architecture for our problem. In this process, we modify the architecture of the model by adding or removing layers, and then evaluate the performance of the modified network to get an idea of further modifications to further improve the performance of the network. The techniques used to reduce the computation cost for this type of process include training on a small number of batches and then predicting how it could improve with more batches.

  We can also run only a reduced version of the network which retains the properties of the full version. Another interesting method is to use the heuristic evaluation function, in which we train a separate model, which will first learn by creating some networks, estimating their performance, and then actually calculating their performance. In this way it will learn to evaluate the network performance, and then we can use this to evaluate our network's performance and then improve it.

- **Weight Decay** : It is a type of regularisation that aids in generalization. It controls the size of weights by adding a penalty to the loss function for bad weights which might reduce the performance of the network. The penalty is given by:

  $$\lambda \sum_{i,j} W_{i,j}^2$$

  Here again $\lambda$ acts as a hyperparameter which controls the extent of regularization. $\lambda = 0$ means weight decay is not being applied, whereas higher values of $\lambda$ tend to smaller weights. When ReLU is the activation function, weight decay works by keeping the activation function near the linear region of the ReLU instead of the flat part where ReLU = 0. This prevents the problem of vanishing gradients.

- **Dropout** : is another technique which enhances the generalization of the neural network. It deactivates a subset of neurons during training. At test time the full network is used without dropout

  $$\text{Dropout formula: output} = \begin{cases} 0, & \text{with probability } 1 - p \\ \frac{\text{original output}}{p}, & \text{with probability } p \end{cases}$$

  If we take the most effective value p=0.8, then it creates a network with around half as much of the original neurons, called a thinned network. Dropout makes the model robust against noise by introducing some noise in the training set during training.