Convolutional Neural Networks

M. Şafak Bilici

1 What is Convolutional Neural Networks?

Convolutional Neural Networks was proposed by Yann Lecun in 1989. CNNs are specialized kind of neural networks for processing data that has a know, gird-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought as a 2D grid of pixels. CNNs are a category of Neural Networks that have proven very effective in areas such as image recognition and classification. The vast majority of applications of CNNs focus on image data, although one can also ose these networks for all types of temporal, spatial and spatiotemporal data.(Goodfellow, Bengio, & Courville, 2016)

Early motivation for CNNs was derived from experiments by Hubel and Wiesel on a cat's visual cortex. The visual cortex has small regions of cells that are sensitive to specific regions in the visual field. In other words, if the specific areas of the visual field are excited, then those cells in the visual cortex will be activated as well. For example, vertical edges cause some neuronal cells to excited, whereas horizontal edges cause other neuronal cells to be excited. The cells are connected using layered architecture, and this discovery led to the conjecture that mammals use these different layers to construct portions of images at different levels of abstraction. From machine learning point of view, this principle is similiar to that hierarchical feature extraction. CNNs achieve something similiar by encoding primitive shapes in earlier layers, and more complex shapes in later layers (Charu C. Aggarwal 2018). Based on this biological inspirations, the earliest neural model was the Neocognitron by Kunihiko Fukushima in 1979. However, there were several differences between this model and the modern CNNs. The most prominent of these differences was that the notion of weight sharing was not used. Based on this architecture, one of the first fully convolutional architectures, referred to as LeNet-5 was developed by Yann Lecun in 1989. The later works will be discussed.

2 The Convolution Operator

In mathematics, convolution is a mathematical operation on two or more than two functions that produces a new function expressing how the shape of one is modified by the others. Usually, the convolution operation used in CNNs does not correspond precisely to the definition of convolution as used in other fields such as engineering (signal processing) or pure mathematics.
Suppose that you are James T. Kirk from Start Trek, and you are captured in a planet, you want to track the location of USS Enterprise with a laser sensor. The laser sensor provides a single output $x(t)$, the position of the Enterprise at time t. Bot x and t are real-valued, i.e., you can get a different reading from the laser sensor at any instant time. Now suppose that your laser sensor is somewhat noisy. To obtain a less noisy estimate of the Enterprise's position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this with a weighted average that gives more weight to recent measurements. If we apply such as weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of spaceship:

$$s(t) = \int x(t_0)w(t - t_0)dt_0$$

This operation called convolution, denoted as

$$s(t) = (x * w)(t)$$

Of course w needs to be a valid probability density function and w needs to be 0 for all negative arguments, or it will look into the future, which is not okay. But, usually when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. In our example, it might be more realistic to assume that our laser provides a measurement once per second. The time index t can take on only integer values. If we now assume that x and w are defined only on integer t, we can define the discrete convolution as:

$$s(t) = (x * w)(t) = \sum_{n=-\infty}^{\infty} x(n)w(t - n)$$

w is sometimes referred as the kernel and the output of convoluton operator is sometimes referred as the feature map. Oftenly, convolution operator can be over more than one axis at a time. For example if we use two-dimesional image I as our input, we probably also use a three-dimensional kernel K:

$$F(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n)$$

Denote that, the convolution operator is commutative,

$$F(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n)$$

Instead convolution, many neural network libraries implement a related functison called the cross-correlation, which is the same as convolution but without flipping the kernel,

$$F(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

Correlation is the process of moving kernel over the image and computing the sum of products at each location. Correlation is the function of displacement of the kernel. In other words, the first value of the correlation corresponds to zero displacement of the kernel, the second value corresponds to one unit of displacement, and so on. But in this note, we call cross-correlation to convolution(Goodfellow et al., 2016). Discrete convolution can be viewed as matrix multiplication. However the matrix has several elements constrained to be equal to other elements. For example, for univariate discrete convolution, each row of the matrix is constrained to be equal to the row above shifted by one element. This known as a Toeplitz Matrix. We can think cross-correlation as a Hadamard Product but after the production of input ant the kernel, output of the cross-entropy is the sum of all elements of output of the Hadamard Product. You can see the one-dimensional example:

...	128	255	123	45	4	...

$*$

1/9	4/7	-3/2	4/5	1/7

$\approx$

12

The convolution on engineering, signal processing is a very large subject to study. It will be shown more later.

2.1 Convolution With Lena

(a) Input: Colored Lena (b) Output: Edge Detected Lena

Figure 1: Just some Lena's

Efficiency of edge detection. The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all of the vertically oriented edges in the input image, which can be a useful operation for object detection. Both images have height of 512 pixels. The input image has weight of 512 pixels while the output image has weight of 511 pixels. This transformation can be described by a convolution kernel containing two element (1×2 Matrix), and requires 512×511×3 = 784896 floating point operations to compute convolution. To describe the same transformation with a matrix multiplication would take $512 \times 512 \times 511 \times 512$ floating point operations. Convolution is an extremely efficient way of describing transformations that apply the same linear transformation of a small, local region across the entire input.

3 More On Convolutional Neural Networks

CNNs combine three architectural ideas to ensure some degree of shift, scale, and distrotion invariance: local receptive fields, shared weights, and spatial or temporal sub-sampling. Classifying images can be done with an ordinary fully connected feed-forward network. But there will be problems. Firstly, typical images may contain hundreds or millions of pixels. This refers to there may be too much dimensions on the input data. A fully-connected first layer with, say one hundred hidden units in the first layer, would already contain several tens of thousand weights. Such a large number of parameters increases the capacity of the system and therefore requires a larger training set. In addition, the memory requirement to strore many weights may rule out certain hardware implementations. Secondly, a deficiency of fully-connected architectures is that the topology of the input is entirely ignored.

The input variables can be presented in any (fixed) order without affecting the outcome of the training. On the contrary, images have a strong 2D local structure: pixels that are spatially or temporally nearby are highly correlated. Local correlations are the reasons for the well-known advantages of extracting and combining local features before recognizing spatial or temporal objects, before configurations of neighboring pixels can be classified into a small number of categories as edges, corners etc. (LeCun, Bottou, Bengio, & Haffner, 1998)

4 Pooling

A CNN architecture consists three stages. The first one convolution, the second is the non-linear activation, such as ReLU, the third one is pooling (subsampling) to modify the output of the layer further. A pooling function replaces the output of the net at a certaion location with a summary of statistic of the nearby outputs. For example the max pooling, was proposed by Zhou and Chellappa in 1988, operation finds the maximum element within a rectangular or square neighborhood. Other popular pooling functions include the average pooling, the L^2 norm of a rectangular or square neighborhood, or weighted average pooling based on the discance from the central pixel.(Goodfellow et al., 2016)
In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by small amount, the values of most of the pooled outputs do not change. You can see a Lena example:

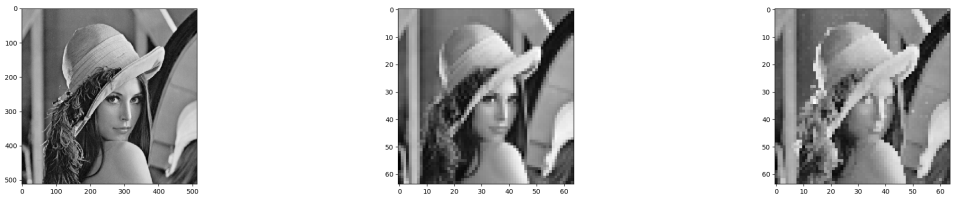


Figure 2

Left one is Lena image with size of 512×512, middle one is average pooling Lena, right one is max pooling Lena; with the kernel size of eight.

You can see the source code of the pooling layer:

```
import numpy as np
def pool2d(im, kernel, mode='max'):
    kernel0, kernel1 = kernel
    feature_map = 255 * np.ones(shape=(int(im.shape[0]/kernel0),
                                       int(im.shape[1]/kernel1)))
    for i in range(0,im.shape[0],kernel0):
```

```
for j in range(0,im.shape[1],kernel1):
    if(mode == "max"):
        feature_map[int(i/kernel0),int(j/kernel1)] = np.max(im[i:i+kernel0
                                                              ,j:j+kernel1])
    if(mode == "avg"):
        feature_map[int(i/kernel0),int(j/kernel1)] = im[i:i+kernel0
                                                        ,j:j+kernel1].mean()
```

Invarianvce to local translation can be very useful property if we care more about whether some feature is presented than exactly where it is. For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy., we just need to know that there is eyes on the left and right side of the face. In other contexts, it is more important to preserve the location of a feature.
Some theoretical work gives guidance as to which kinds of pooling one should use in various situations. It is also possible to dynamically pool features together, for example, by running a cluster algorithm on the locations of interesting features. This approach yields a different set of pooling regions for each image. Another approach is to learn a single pooling structure that is then applied to all images.
Pooling can be complicate some kinds of neural networks architectures that use top-down information, such as Boltzmann Machines and Autoencoders. You can see the visualizaed pooling in Figure 3, below.

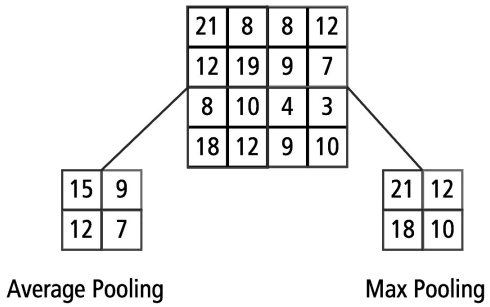


Figure 3

5 Padding

One observation is that the convolution operation reduces the size of $(q+1)$th layer in comparsion with the size of the qth layer. This type of reduction in size is not desirable in general, because it tends to lose some information along the borders of the image (or feature map in the hidden layers). This problem can be resolved by using the padding. Kernel size in the layer q denoted as K_q; in padding, one adds $(K_q - 1)/2$ "pixels" all around the borders of the image-feature map in order to maintain the spatial footprint. Note that these pixels are really feature values in the case of the padding hidden layers. The value of each of each of these padded

feature values is set to 0, irrespective of whether the input or the feature maps at the hidden layers are being padded. As a result, the spatial height and width of the input volume will both increase by $(K_q - 1)$, which is exactly what they reduce by (in the output volume) after the convolution is performed. The padded portions do not contribute to the final dot product due to their values are set to 0.

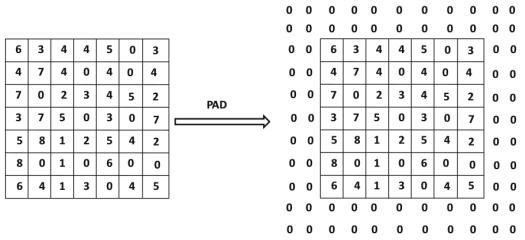


Figure 4: Padding before convolution with 5×5 kernel.

In a sense, what padding does is to allow the convolution operation with a portion of the filter "sticking out" from the borders of the layer and then performing the dot product only over the portion of the layer where the values are defined. This type of padding is referred to as *half-padding* because (almost) half the kernel is sticking out from all sides of the spatial input in the case where the kernel is placed in its extreme spatial position along the edges. Half-padding is designed to maintain the spatial footprint exactly.

When padding is not used, the resulting padding is also referred to as *valid padding*. Valid padding generally does not work well from an experimental point of view. Using half-padding ensures that some of the critical informatiin at the borders of the layer is represented in a standalone way. In the case of valid padding, the contributions of the pixels on the bırders of the layer will be under-represented compared to the cental pixels in the next hidden layer, which undesirable. Furthermore, this under-representation will be compounded over multiple layers. Therefore, padding is typically performed in all layers, and not just in the first layer where the spatial locations correspond to input values.

Another useful form of padding is *full-padding*, we allow almost full kernel to stick out from various sides of the input. In other words, a portion of the kernel of size $K_q - 1$ is allowed to stick out from any side of the input with an overlap at a single pixel at an extreme corner. Therefore, the input is padded with $K_q - 1$ zeros on each side. In other wors, each spatial dimesion of the input increases by $2(K_q - 1)$. Therefore, if the input dimensions in the original image are all H_q and W_q, the padded spatial dimesions in the input volumebecome $H_q + 2(K_q - 1)$ and $W_q + 2(K_q - 1)$. After performing the convolution, the feature-map dimensions in layer $q + 1$ become $H_q + K_q - 1$ and $W_q + K_q - 1$. While convolution normally reduces the spatial footprint, full-padding increases the spatian footprint.
This relationship is not a coincidence because a "reverse" convolution operation can

be implemented by applying another convolution on the fully padded output with an appropriately defined kernel of the same size.This type of "reverse" convolution occurs frequently in the back- propagation and autoencoder algorithms for convolutional neural networks. Fully padded inputs are useful because they increase the spatial footprint, which is required in several types of convolutional autoencoders.

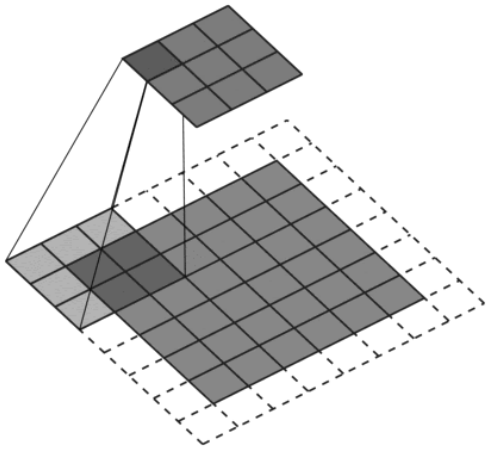


Figure 5: Convolution after padding, with kernel size 3×3.

6 Stride

There are other ways in which convolution can reduce the spatial footprint of the image. It is not necessary to perform the convolution at every spatial position in the layer. One can reduce the level of granularity of the convolution by using the notion of strides. When a stride of S_q is used in the qth layer, the convolution is performed at the locations 1, $S_q + 1$, $2S_q + 1$, and so on along both spatial dimensions of the layer. The spatial size of the output on performing this convolution has height of $(H_q - K_q)/S_q + 1$ and a width of $(W_q - K_q)/S_q + 1$.
It is most common to use a stride of 1, although a stride of 2 is occasionally used as well. It is rare to use strides more than 2 in normal circumstances. Even though a stride of 4 was used in the input layer of the winning architecture of the ILSVRC (ImageNet Large Scale Visual Recognition Challenge) competition of 2012, the winning entry in the subsequent year reduced the stride to 2 to improve accuracy. Larger strides can be helpful in memory-constrained settings or to reduce overfitting if the spatial resolution is unnecessarily high. Strides have the effect of rapidly increasing the receptive field of each feature in the hidden layer, while reducing the spatial footprint of the entire layer.

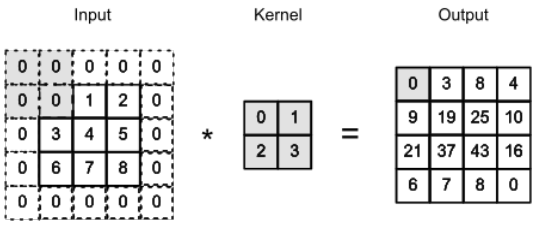


Figure 6: Padding + Convolution with 2×2 kernel size with stride 1.

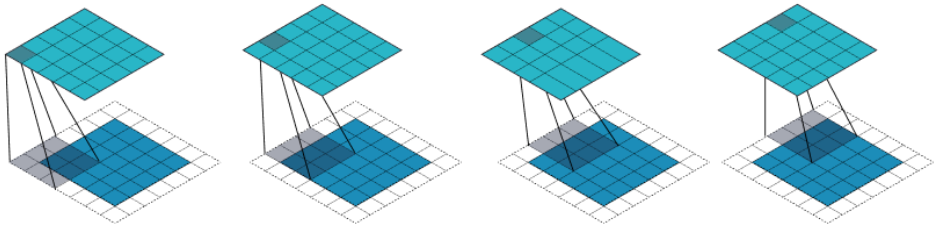


Figure 7: Padding + Convolution with 3×3 kernel size with stride 1.

7 Useful Practical Formulas for Implementations

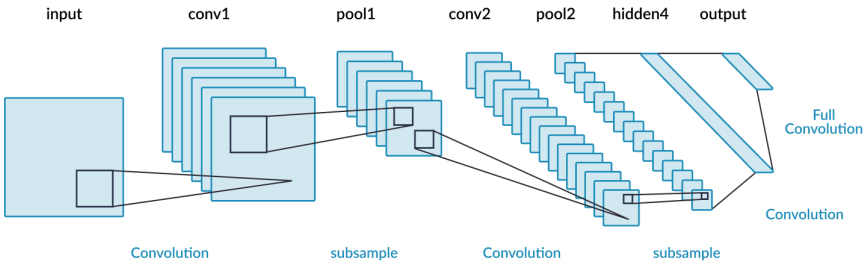


Figure 8: Architecture of CNN.

In this section, we will discuss the formulas we got in previous sections. Due to convolutions, paddings, strides and poolings; the size of the image is always changing in layers. We had showed the changing the size of the image, now we try to do some formulation.

7.1 Convolution Layer

An image accepts a volume of size $W_1 \times H_1 \times D_1$:

→ Required Hyperparameters:

- Number of Kernels K
- Kernel's Spatial Extent F
- Stride S
- Amount of Zero Padding P

→ Produces A Volume of Size $W_2 \times H_2 \times D_2$:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$
- $D_2 = K$

7.2 Pooling Layer

→ Required Hyperparameters:

- Kernel's Spatial Extent F
- Stride S

→ Produces A Volume of Size $W_2 \times H_2 \times D_2$:

- $W_2 = (W_1 - F)/S$
- $H_2 = (H_1 - F)/S$
- $D_2 = D_1$

→ If No Overlapping:

- $W_2 = W_1/F$
- $H_2 = H_1/F$
- $D_2 = D_1$

When machine learning library is implemented, this formulas have to be considered. In figure 8, we can see a full CNN architecture. We have already discussed the terms pooling and convolutiond, and this is how we connect them. Layers conv1, pool1, conv2, pool2 changes the size of the input or feature map. The layer Hidden4 changes the size of the feature map but in different way. IN this layer, the architecture flattens (fully connected layer) the feature map in the layer pool2. We will discuss it later.

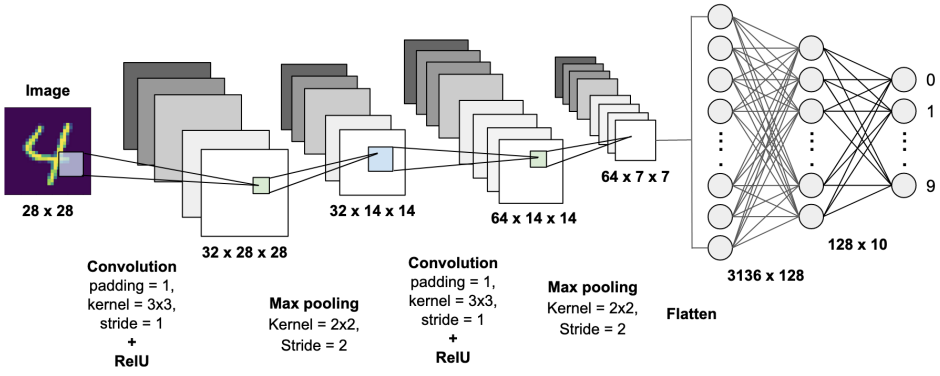


Figure 9: CNN Architecture for Digit Recognition.

8 Batch Normalization

Batch normalization is a recent method to address the vanishing and exploding gradient problems, which cause activation gradients in successive layers to either reduce or increase in magnitude. Another very imprortant problem in training a deep learning model is that of internal covariate shift. The problem is that the parameters change during training, and therefore the hidden variable activations change as well. In other words, the hidden inputs from early layers to later layers keep changing. Changing inputs from early layers to later layers causes slower convergence during training because the training data for later layers is not stable. Batch Normalization reduces this effect.(Charu C. Aggarwal, 2018)
In Batch Normalization, the idea is to add additional "normalization layers" between hidden layers that resist this type of behavior by creating features with somewhat similiar variance. We normalize the input layer by adjusting and scaling the activations. For example, when we have features from 0 to 1 and some from 1 to 1000, we should normalize them to speed up learning. If the input layer is benefiting from it, why not do the same thing also for the values in the hidden layers, that are changing all the time, and get 10 times or more improvement in the training speed. (Ioffe & Szegedy, 2015)
Furthermore, each unit in the normalization layers contain two additional parame-

ters β_i and γ_i that regulate the precise level of normalization in the ith unit; these parameters are learned in a data-driven manner. The basic idea is that the output of the ith unit will have a mean of β_i and a standard deviation of γ_i over each mini-batch of training samples. One might wonder that whether it might make sense to simply set each β_i to 0 and each γ_i to 1, but doing so reduces the representation power of the network.

What transformations does Batch Normalization apply? Consider the case in which its input $x_i^{(r)}$, corresponding to the rth element of batch feeding into the ith unit. Each $x_i^{(r)}$ is obtained by using linear transformation defined by the weight vector and biases. For a particular batch of m instances, let the values of the m activations be denoted by $x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(m)}$. The first step is to compute the mean μ_i abd the standard deviation σ_i for the ith hidden unit. These are then scaled using the parameters β_i and γ_i to create the outputs for the next layer:

Algorithm 1 Batch Normalization

1: **Input**: Values of x over a mini-batch $B = \{x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(m)}\}$
2: **Output**: $\{y_i = BN_{\gamma_i,\beta_i}(x_i^{(r)})$
3: $\mu_i \leftarrow \frac{\sum_{r=1}^{m} x_i^{(r)}}{m}$
4: $\sigma_i^2 \leftarrow \frac{\sum_{r=1}^{m}(x_i^{(r)}-\mu_i)^2}{m} + \epsilon$
5: $\hat{x}_i^{(r)} \leftarrow \frac{x_i^{(r)}-\mu_i}{\sigma_i}$
6: $a_i \leftarrow \gamma_i \cdot \hat{x}_i^{(r)} + \beta_i$

ϵ is a small value to prevent that σ_i^2 can be zero. Note that a_i is the pre-activation output of the ith node. We conceptually represent this node as BN_i that performs this additional processing. Therefore, the backpropagation algorithm has to account for this additional node and ensure that the loss derivative of layers earlier than the batch normalization layer accounts for the transformation implied by these new nodes. This type of computation is unusual for a neural network in which the gradients are linearly separable sums of the gradients respect to individual training examples. This is not quite true in this case because the batch normalization layer computes nonlinear metrics, such as its standard deviation, from the batch. Therefore , the activations depended on how the examples in a batch are related to one another, which is not common in most neural network computations. The following will describe the changes in the backpropagation algorithm caused by the normalization layer. The main point of this change is to show how to backpropagate through the newly added layer of normalization nodes. Another point to be aware of is that we want to optimize the parameters γ_i and β_i. For the gradient descent steps respect to each γ_i and β_i we need the gradients with respect to this parameters. Assume that we have already backpropagated up to the output of the BN node, and

therefore we have each $\frac{\partial L}{\partial a_i^{(r)}}$ available:

$$\frac{\partial L}{\partial \beta_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial \beta_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}}$$

$$\frac{\partial L}{\partial \gamma_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial \gamma_i} = \sum_{r=1}^{m} \frac{\partial L}{\partial a_i^{(r)}} \cdot \hat{x}_i^{(r)}$$

We also need a way to compute $\frac{\partial L}{\partial x_i^{(r)}}$. Once this value is computed, the backpropagation to pre-activation values $\frac{\partial L}{\partial \hat{y}_j^{(r)}}$ for all nodes j in the previous layer uses the straightforward backpropagation update. Therefore we have the following... Here comes the holy backpropagation:

$$\frac{\partial L}{\partial x_i^{(r)}} = \frac{\partial L}{\partial \hat{x}_i^{(r)}} \frac{\partial \hat{x}_i^{(r)}}{\partial x_i^{(r)}} + \frac{\partial L}{\partial \mu_i} \frac{\partial \mu_i}{x_i^{(r)}} + \frac{\partial L}{\partial \sigma_i^2} \frac{\partial \sigma_i^2}{x_i^{(r)}}$$

$$= \frac{\partial L}{\partial \hat{x}_i^{(r)}} \left(\frac{1}{m}\right) + \frac{\partial L}{\partial \mu_i} \left(\frac{1}{m}\right) + \frac{\partial L}{\partial \sigma_i^2} \left(\frac{2(x_i^{(r)} - \mu_i)}{m}\right)$$

and,

$$\frac{\partial L}{\hat{x}_i^{(r)}} = \gamma_i \frac{\partial L}{\partial a_i^{(r)}} \quad [since \;\; a_i^{(r)} = \gamma_i x_i^{(r)} + \beta_i]$$

and we have following,

$$\frac{\partial L}{\partial x_i^{(r)}} = \frac{\partial L}{\partial a_i^{(r)}} \left(\frac{\gamma_i}{m}\right) + \frac{\partial L}{\partial \mu_i} \left(\frac{1}{m}\right) + \frac{\partial L}{\partial \sigma_i^2} \left(\frac{2(x_i^{(r)} - \mu_i)}{m}\right)$$

So... It is time to compute the partial derivation of the loss respect to the mean and the variance.

$$\frac{\partial L}{\partial \sigma_i^2} = \sum_{j=1}^{m} \frac{\partial L}{\partial \hat{x}_i^{(j)}} \frac{\partial \hat{x}_i^{(j)}}{\partial \sigma_i^2} = -\frac{1}{2\sigma_i^3} \sum_{j=1}^{m} \frac{\partial L}{\hat{x}_i^{(j)}} (x_i^{(j)} - \mu_i) = -\frac{1}{2\sigma_i^3} \sum_{j=1}^{m} \frac{\partial L}{a_i^{(j)}} \gamma_i (x_i^{(j)} - \mu_i)$$

$$\frac{\partial L}{\partial \mu_i} = \sum_{j=1}^{m} \frac{\partial L}{\partial \hat{x}_i^{(j)}} \frac{\partial \hat{x}_i^{(j)}}{\partial \mu_i} + \frac{\partial L}{\partial \sigma_i^2} \frac{\partial \sigma_i^2}{\partial \mu_i} = -\frac{1}{\sigma_i} \sum_{j=1}^{m} \frac{\partial L}{\hat{x}_i^{(j)}} - 2 \frac{\partial L}{\partial \sigma_i^2} \frac{\sum_{j=1}^{m} (x_i^{(j)} - \mu_i)}{m}$$

$$= -\frac{\gamma_i}{\mu_i} \sum_{j=1}^{m} \frac{\partial L}{\partial a_i^{(j)}} + \left(\frac{1}{\sigma_i^3}\right) \left(\sum_{j=1}^{m} \frac{\partial L}{\partial a_i^{(j)}} \gamma_i (x_i^{(j)} - \mu_i)\right) \left(\frac{\sum_{j=1}^{m} (x_i^{(j)} - \mu_i)}{m}\right)$$

This equations provides us a full view of the backpropagation of the loss through

the batch normalization layer corresponding to the BN node. The other aspects of bacpropagation remain similiar to traditional case. Also batch normalization enables faster inference because it prevents problems such as the exploding and vanishing gradient which cause slow learning. A natural question about batch normalization arises during prediction time. Since the transformation parameters μ_i and σ_i depend on natch, how should one compute them during testing when a single test instance is available? In this case, the values of μ_i and σ_i are computed up front using the entire population, and then treated these values as constants during testing time.
An interesting property of batch normalization is that it also acts as a regularizer. Note that same data point can cause somewhat different updates depending on which batch it is included in. One can view this effect as a king of noise added to the update process. Regularization is often achieved by adding a small amount of noise to the training data. It has been experimentally observed that regularization methods like Dropout do not seem to improve perfomance when batch normalization is used although there is not a complete agreement on this point. Batch normalization is oftenly used in CNNs.

9 The ReLU Layer

Rectified Linear Unit, also denoted as ReLU, is an commonly used activation function in Deep Learning methods. The ReLU is mathematically defined as,

$$f(x) = x^+ = max(0, x)$$

visually look like,

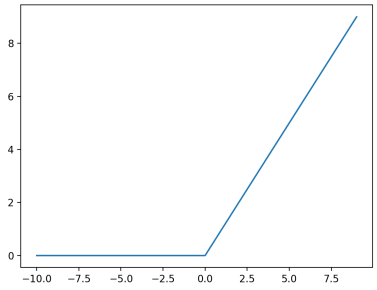


Figure 10: The ReLU.

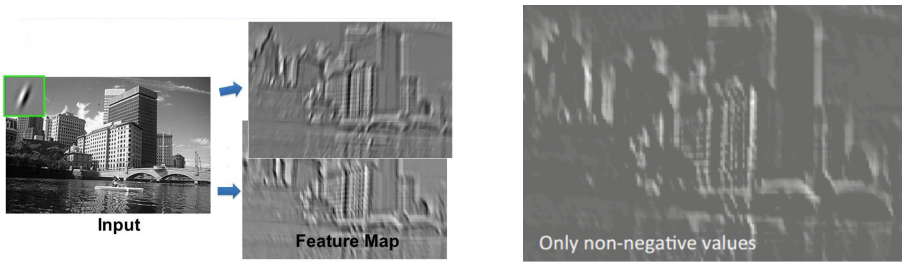
9.1 Limitations of Sigmoid and Tanh Activation Functions

Nonlinear activation functions are preferred as they allow the nodes to learn more complex structures in the data. Traditionally, two widely used nonlinear activation functions are the sigmoid and hyperbolic tangent activation functions. The sigmoid activation function, also called the logistic function, is traditionally a very popular activation function for neural networks. The input to the function is transformed into a value between 0.0 and 1.0. Inputs that are much larger than 1.0 are transformed to the value 1.0, similarly, values much smaller than 0.0 are snapped to 0.0. The shape of the function for all possible inputs is an S-shape from zero up through 0.5 to 1.0. For a long time, through the early 1990s, it was the default activation used on neural networks. The hyperbolic tangent function, or tanh for short, is a similar shaped nonlinear activation function that outputs values between -1.0 and 1.0. In the later 1990s and through the 2000s, the tanh function was preferred over the sigmoid activation function as models that used it were easier to train and often had better predictive performance. A general problem with both the sigmoid and tanh functions is that they saturate. This means that large values snap to 1.0 and small values snap to -1 or 0 for tanh and sigmoid respectively. Further, the functions are only really sensitive to changes around their mid-point of their input, such as 0.5 for sigmoid and 0.0 for tanh. The limited sensitivity and saturation of the function happen regardless of whether the summed activation from the node provided as input contains useful information or not. Once saturated, it becomes challenging for the learning algorithm to continue to adapt the weights to improve the performance of the model. *Finally, as the capability of hardware increased through GPUs' very deep neural networks using sigmoid and tanh activation functions could not easily be trained.*

9.2 Moreover About ReLU and Convolutional Neural Networks

The concolution operator is interleaved with the pooling and ReLU operations. The ReLU is not very different from how it is applied in traditional neural networks. For each of the $W_q \times H_q \times D_q$ values in a layer, the ReLU activation function is applied to it to create $W_q \times H_q \times D_q$ thresholded values. These values are then passed onto the next layer. Therefore applying ReLU does not change the dimensions of a layer because it is a simple one-to-one mapping of activation values. The purpose of applying the rectifier function is to increase the non-linearity in our images. The reason we want to do that is that images are naturally non-linear. When you look at any image, you'll find it contains a lot of non-linear features (e.g. the transition between pixels, the borders, the colors, etc.). ReLU is an element wise operation (applied per pixel) and replaces all negative pixel values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in our ConvNet, since most of the real-world data we would want our ConvNet to learn would be non-linear (Convolution is a linear operation – element wise matrix multiplication

and addition, so we account for non-linearity by introducing a non-linear function like ReLU).



(a) Input and Convolved Image (b) Applying ReLU

Figure 11: The ReLU Layer

10 Fully Connected Layers and Flattening

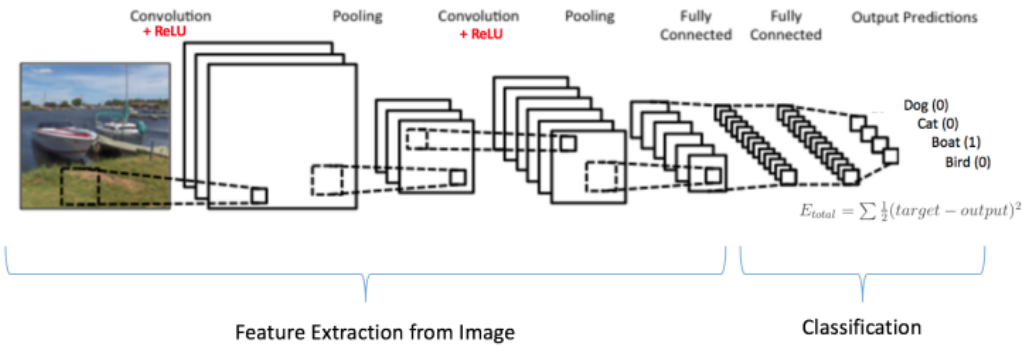


Figure 12: Feature Extraction and Classification.

Each feature in the final spatial layer is connected to each hidden state in the first fully connected layer. This layer functions in exactly the same way as a traditional feed forward network. In most cases, one might use more than one fully connected layer to increase the power of the computations towards the end. The connections among these layers are exactly structured like a traditional feed forward networks. Since the fully connected layers are densely connected, the vast majority of parameters lie in the fully connected layers. For example, if each of two fully connected layers has 4096 hidden units, the the connections between them have more than 16 million weights.
The output layer of CNN is designed in an applicatio-specific way. In the following, we will consider the representative application of classification. In a such case, the output layer is fully connected to every neuron in the penultimate layer, and has a

weight associated with it. One might use the logistic, softmax, or linear activation depending on the nature of the application.
Flattening the final feature map can be viewed as, transforming image $W \times H \times D$ matrix into $1 \times (W \times H \times D)$ array. This means, first fully connected layer has $W \times H \times D$ neurons in it. In fully connected layers, we are classifying feature extracted inputs. For example we have an data that has handwritten digits, from 0 to 9. And final convolution layer generates 10 feature map with size of 7×7. This means, we have $7 \times 7 \times 10 = 490$ neurons in first fully connected layer. And in the last fully connected layer, output layer, we have 10 neurons to classify digits from 0 to 9.

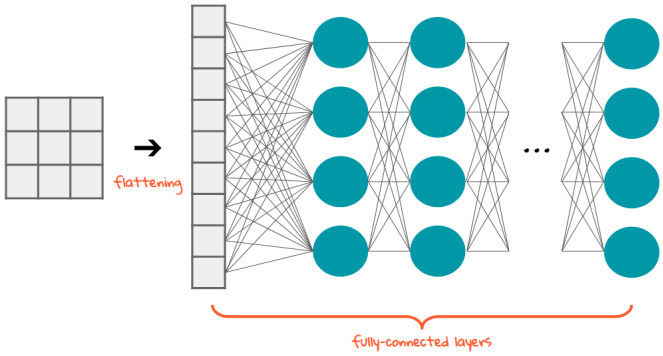


Figure 13: Flattening the image and classifying 4 classes.

11 A Practical Example: MNIST

The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. The MNIST database contains 60,000 training images and 10,000 testing images.

Figure 14: Samples from MNIST Database.

Each MNIST image is a gray scale image and has size of 28×28. So, let's define our model and architecture.

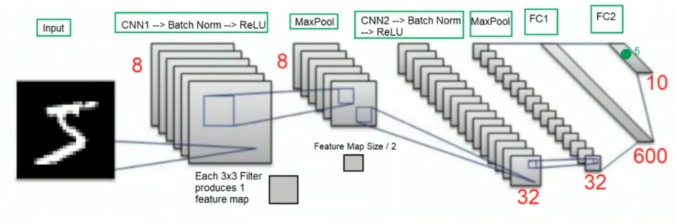


Figure 15: An architecture that classifies hand-written digits.

- In our architecture, we have two Convolution Layer and 2 Pooling Layer. In first Convolution Layer we have 8 kernels that have size of 3×3. We apply this kernels with stride of 1. Hence, first Convolution Layer produces 8 feature map. In input images, we appy zero padding on each image. The amount of zero padding is dependent on kernel size: amount_of_padding = (kernel_size - 1)/2 $\rightarrow$ (3-1)/2 = 1, as we discussed in section 5.

- So, what will be our each feature map's size? From feature_map_size = ((input_size - kernel_size + 2(padding))/(stride) +1), as we discussed in section 6, each feature map's size will be (28 -3 + 2)/1 + 1 = 28. Then we apply Batch normalization and ReLU, which do not change the size of the feature maps.

- So, we have 8 feature maps with the size of 28×28. Now, it is time to apply max-pooling with kernel size 2×2. This pooling layer affects only the widht and height of the image. After applying the max-pooling with kernel size 2×2, each feature map has a size of $28/2 \times 28/2$, which is 14×14.

- So, we have now 8 feature maps with size of 14×14 before applying second convolution operator. In second Convolution Layer, we have 32 kernels with size of 5×5. We apply this kernels with stride of 1. This kernel size means that our amount of zero padding is 1. Then this Convolutional Layer produces 32 feature map with size of (14 - 5 + 2x2)/1 + 1 = 14, 14×14. Then we apply Batch normalization and ReLU, which do not change the size of the feature maps.

- So, we have 32 feature maps with the size of 14×14. Now, it is time to apply max-pooling with kernel size 2×2 again. This pooling layer affects only the widht and height of the image. After applying the max-pooling with kernel size 2×2, each feature map has a size of $14/2 \times 14/2$, which is 7×7.

- Now, it is time to flatten the all the feature maps. We currently have 32 feature maps with the size of 7×7. If we flatten this feature maps, number of node of first fully connection layer will be $7 \times 7 \times 32 = 1568$. We choose the number of node of the second fully connected layer as 600. And the output layer must have 10 node.

The pseudo-code of this process will be more helpful to imagine:

Algorithm 2 Initialization

```
cnn1 ← Conv2d(in_channels=1,out_channes=8,kernel_size=3,stride=1,padding=1)
batch_norm1 ← BatchNorm2d(8)
relu ← ReLU()
maxpool ← MaxPool2d(kernel_size = 2)
cnn2 ← Conv2d(in_channels=8,out_channels=32,kernel_size=5,stride=1,padding=2)
batch_norm2 ← BatchNorm2d(32)
fc1 ← Linear(1568,600)
fc2 ← nn.Linear(600,10)
```

Algorithm 3 Forward Propagation(x)

```
out ← cnn1(x)
out ← batch_norm1(out)
out ← relu(out)
out ← maxpool(out)
out ← cnn2(out)
out ← batch_norm2(out)
out ← relu(out)
out ← maxpool(out)
out ← out.flatten(batch_size,1568)
out ← fc1(out)
out ← relu(out)
out ← fc2(out)
return out
```

12 Dropout

Dropout provides us that a computationally inexpensive but powerful method of regularizing a broad family of models. To a first approximation, dropout can be thought of as a method of making bagging practical for ensembles of very many large neural networks. Bagging involves training multiple models, and evaulating

multiple models on each test example. This seems impractical when each model is a large neural network, since training and evaulating such networks costly in terms of runtime and memory. It is common to use ensembles of five to ten neural networks but more than this rapidly becomes unwidely. Dropout provides an inexpensive approximation to training and evaulating a bagged ensemble of exponentially many neural netwoks.

Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network. In most modern neural networks, based on a series ofaffine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero. This procedure requires some slight modification for models such as RBF Networks. Here, we present the dropout algorithm in terms of multiplication by zero for simplicity, but it can be trivially modified to work with other operations that remove a unit from the network.

Recall that to learn with bagging, we define k different models, construct k different datasets by sampling from the training set with replacement, and then train model i on dataset i. Dropout aims to approximate this process, but with an exponentially large number of neural networks. Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all of the input and hidden units in the network. The mask for each unit is sampled independently from all of the others. The probability of sampling a mask value of one is a hyperparameter fixed before training begins. It is not a function of the current value of the model parameters or the input example. Typically, an input unit is included with probability 0.8 and a hidden unit is included with probability 0.5. We then run forward propagation, back-propagation, and the learning update as usual.

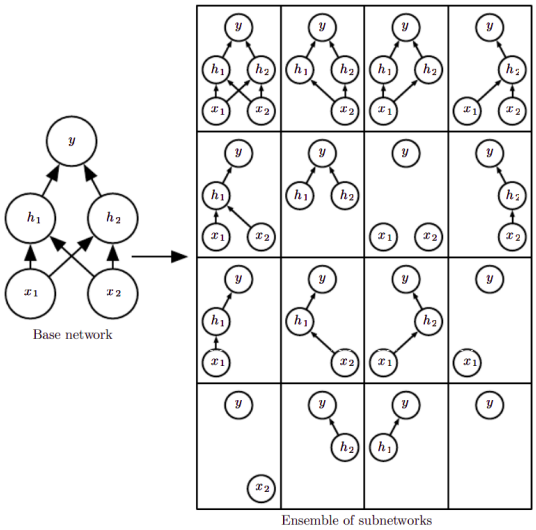


Figure 16: Bagging Approximation.

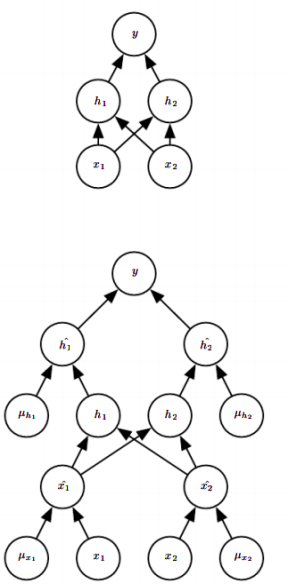


Figure 17: In this example, feedforward network is used. To perform the forward propagation with dropout, we randomly sample vector μ with one entry for each input or hidden unit in the network. Each unit in the network is multiplied by the corresponding mask, and then forward propagation continues.

More mathematically, suppose we have a mask vector μ specifies which units to include, and $L(\theta, \mu)$ is the loss function of the model with parameters θ and mask μ. Then dropuout training consists in minimizing $\mathbb{E}_\mu L(\theta, \mu)$ (Goodfellow et al., 2016)

References

Goodfellow, I. J., Bengio, Y., & Courville, A. (2016). *Deep learning.* Cambridge, MA, USA: MIT Press. (`http://www.deeplearningbook.org`)

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Charu C. Aggarwal (2018). *Neural Networks and Deep Learning.* Springer International Publishing.

https://cs231n.github.io/