Report 2: Mathematical Background for CNNs and Semantic Segmentation

Adithya Narayan

Manipal, Karanataka

Abstract

This report covers in great detail the mathematics behind a majority of what CNN's have to offer. Starting from forward prop, loss functions, back-prop and gradient computations as well as optimizers used in a CNN.

Keywords: CNN, Mathematics

1. On the Mathematics of CNNs

In the last report, the basics of the structure of a CNN were gone over. Additionally, it mentioned in a very general sense why convolutions were preferred over standard multi layer preceptrons(MLPs).

To begin with, a typical MLP flattens the input data.The disadvantage of this is two-fold. One, as the image size increases the number of parameters to train increases as well. Secondly, we do not take advantage of the spatial information in the image.

Hence we arrive at CNNs. Through the process of convolving filters with the inputs, spatial information used. We train weights to understand and fit to these spatial features as opposed to pixel-wise values.

Given that CNNs are supposed to take advantage of local spatial features, CNNs must have certain features. These features are discussed in the next section.

1.1. Important properties of a CNN

1.1.1. Translational Invariance

One important feature of a CNN is that it should be translation invariant. In effect, if the CNN learns the features that make up a dog, the position

of the dog should not effect the CNN. It should be able to identify a dog anywhere in the image.

In effect, patches/clusters of pixels which describe a feature should not depend on where the patch is. So mathematically speaking, this means that the representation on the next layer of the network must be independent of the values of i and j which are used to iterate through the image channels. Hence, for a $N_h * N_w * 1$ image, if we assume $[H]$ to represent the hidden representation in the next layer, $[V]$ to represent a matrix of coefficients and $[X]$ to represent the input, we can write the expression,

$$[H]_{i,j} = \sum_a \sum_b [V]_{a,b}[X]_{i+a,j+b} \tag{1}$$

In actuality, this equation is essentially a convolution(hence why we use convolutions in a CNN) and this will be discussed in the convolutional layer in the upcoming sections of this report.

1.1.2. Locality

In addition to the property above, CNNs must posses the property of locality. This means that when we look at a patch in an image, the features relevant to that patch must be gleaned locally from the neighborhood. We should not have to look at wildly different positions in the image for the network to understand the relevant features.

To do this, we simply put mathematical constraints on equation (1). Specifically, we say that the coefficient corresponding to the pixel position $|a| > \Delta$ and $|b| > \Delta$ should be set to zero. In effect the corresponding $[V]_{a,b} = 0$. Hence, we can represent this as,

$$[H]_{i,j} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [V]_{a,b}[X]_{i+a,j+b} \tag{2}$$

Basically, what the above equation means is we convolve the image with a kernel of dimension $2\Delta + 1$. With the addition of a bias, this essentially describes the convolutional layer of a CNN. With these in mind, we now look at the layers of a CNN.

1.2. Layers Of a CNN

1.2.1. Convolutional Layer

As discussed previously, the convolutional layer is essentially responsible for emphasizing different features in an image using a kernel. While defining

the translational invariance and locality properties, we constrained the input to a single channel. This can be generalized for an image with a greater number of channels. Additionally, we also include a bias u to the equation(2).This can be then written as,

$$[H]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [V]_{a,b,c,d} [X]_{i+a,j+b,c} \tag{3}$$

This is essentially passing a kernel over an image and multiplying and adding corresponding coefficients from $[V]$ with values in $[X]$. Note that in this report the term convolution is used rather loosely and as most research would suggest, correlation is what we really perform and the term convolution is a bit of a misnomer. Anyhow, we can visualize this as shown below.

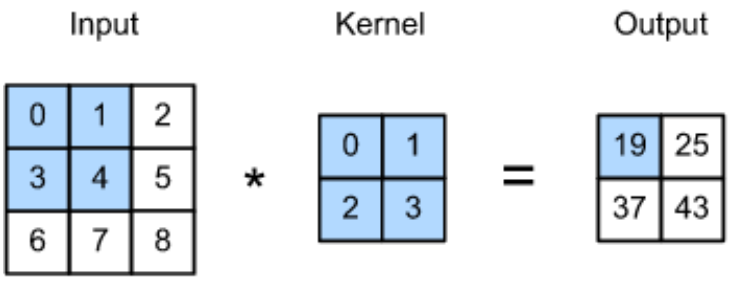

Figure 1: And example of 2D convolution

We also inspect the change in dimension as we pass a kernel over an input image $[X]$. Suppose we have a input image of dimension $N_h * N_w * N_c$. If we pass m different kernels, each of dimension $f * f * N_c$ we obtain an output with dimension of,

$$dims([H]_h) = N_h + f - 1 \tag{4}$$

$$dims([H]_w) = N_w + f - 1 \tag{5}$$

The above equations are only for 'valid' convolutions where the dimension of the output will be lower than that of the input image. However, it is also possible to include padding in the original image so as to maintain output dimensions(this is known as 'same' convolution). Additionally, the above equations are for a stride of one. We can also describe convolutions for higher strides. Assigning variable p for padding and s for stride, we can

generalize the above equation to,

$$dims([H]_h) = \left\lfloor \frac{N_h - f + 2p}{s} + 1 \right\rfloor \tag{6}$$

$$dims([H]_w) = \left\lfloor \frac{N_w - f + 2p}{s} + 1 \right\rfloor \tag{7}$$

After the convolution operation, we pass the outputs through a non-linear activation function to generate the output. Typically, you use relu for the middle layers and a sigmoid/softmax layer for the output. These will be gone into in more detail in later sections. Taking this into account and generalizing the convolution equation so, we can define the l^{th} convolutional layer with the two following equations,

$$[H]^{[l]} = [V]^{[l]} \cdot [X]^{[l-1]} + b^{[l]} \tag{8}$$

$$[A]^{[l]} = g([H]^{[l]}) \tag{9}$$

1.2.2. Pooling

The pooling layer is similar to a convolutional layer in that this layer also uses a kernel that is passed over the outputs of the convolutional layer. However, the purpose of the pooling layer is slightly different.Pooling is generally done to compress the height and width of the previous input. Typically, two types of pooling are used: max pooling and average pooling.

In max pooling, the maximum value is taken from the range specified by the kernel in the previous layer. This works under the assumption that the most important features will have the largest activation response. Hence, this feature should be an effective description of the region defined by the kernel. The output dimensions for this is consistent with equation (6) and (7) above. Typically, we use a pooling layer of $f = 2$ and $s = 2$ to downsize the image.

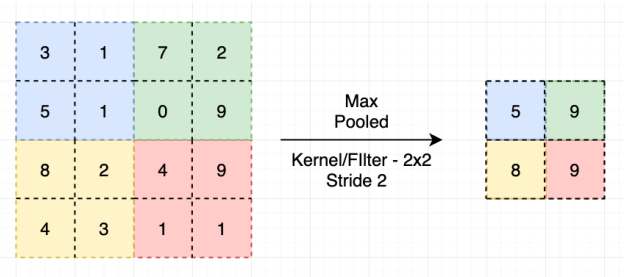

Figure 2: Max Pooling with a $2 * 2$ window

In average pooling, as opposed to finding the maximum in the given window, all values in the window are summed and divided by the number of values in the sum.

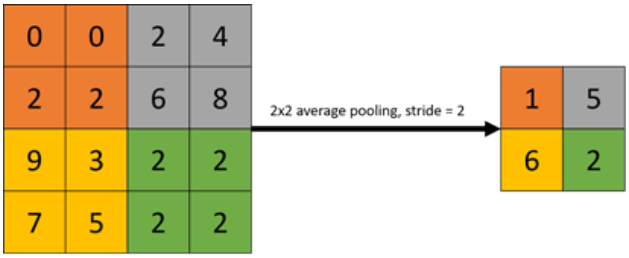

Figure 3: Average Pooling with a $2 * 2$ window

1.2.3. Fully Connected Layer

Typically at the final stages of a CNN the multi-dimensional output form a Pooling layer is flattened into a vector and passed into a Fully Connected Layer. These layers are stacked to basically form a multi-layer preceptron network. This layer follows a similar set of equations as (8) and (9) for linear mapping and non-linear activation. However in this case, the inputs are vectors and so are the outputs.

1.2.4. Output Layer

Typically found at the very end of a standard CNN, this layer generates the output of a typical CNN. Usually this is a softmax/sigmoid layer depending on the number of output classes. The mathematical details of the same will be gone over in the upcoming section.

1.3. Activation Functions Used in a CNN

1.3.1. Rectified Linear Unit

The Rectified Linear Unit(ReLu) for short, is one of the non linear activations used in a CNN. This,or a version of it, is generally used as the non-linear activation in the middle layers of a CNN. The ReLu function is defines as,

$$g(z) = max(0, z) \tag{10}$$

As can be seen, the ReLu function is more computationally efficient compared to tanh and sigmoid. Additionally, since the gradient of ReLu is positive and non-zero for large values of z, it combats the problem of vanishing gradients in a neural network.

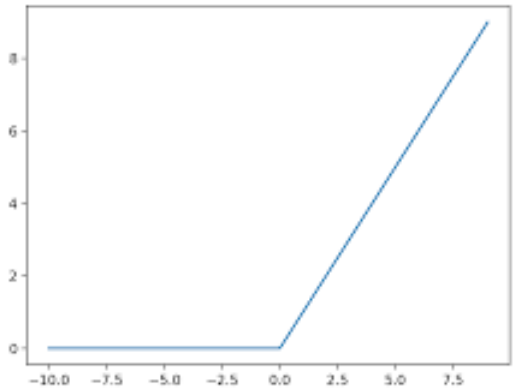

Figure 4: ReLu function

One interesting problem with a ReLu function is the problem that if any neuron ends up with a negative value of z, the non-linear activation of it becomes zero. Additionally, these 'dead' neurons are unlikely to recover since the gradient is always zero in this region. As a consequence, a large portion of your network will essentially do nothing and this is extremely inefficient.

One solution to the above problem is to use a version of ReLu with a slight positive slope in the negative region. This is known as 'Leaky Relu'. The Leaky ReLu function is given as,

$$g(z) = \begin{cases} \alpha z & \text{if } z \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{11}$$

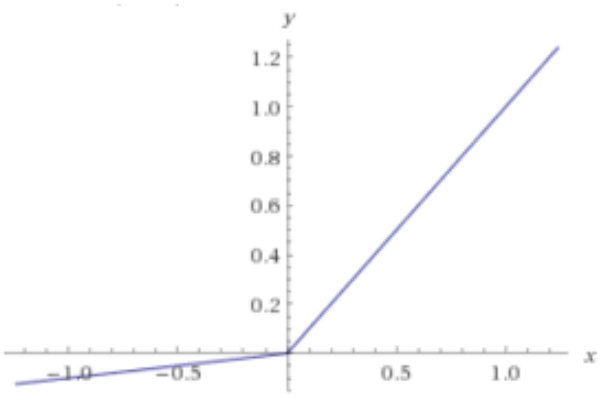

Figure 5: LeakyReLu function.

1.3.2. Sigmoid/tanh

the sigmoid and tanh functions are typically used for the final layer in a neural network. They allow for binary output classes. The Sigmoid function is given as,

$$g(z) = \frac{1}{1 + e^{-z}} \tag{12}$$

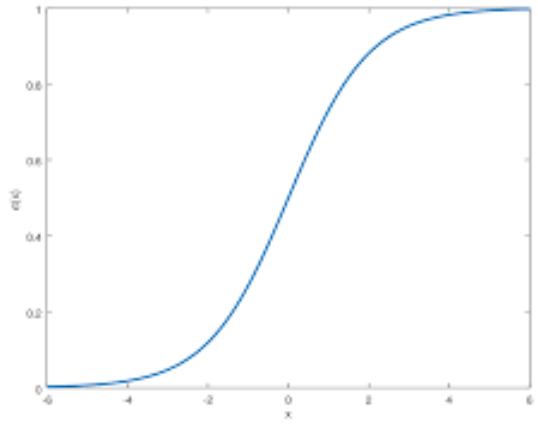

Figure 6: Sigmoid function.

Similarly, we have the tanh function as,

$$g(z) = tanh(z) = \frac{2}{1 + e^{-2z}} - 1 \tag{13}$$

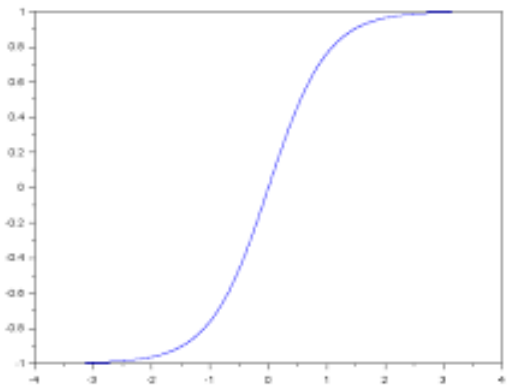

Figure 7: The tanh function.

Both these non-linear activations have the same issue. At larger values of z, both face the problem of vanishing gradients. One other interesting thing to note however, is that the choice between tanh and sigmoid depends on the network trained. Since tanh offers a steeper slope, it learns parameters faster.

1.3.3. Softmax

For CNNs with greater than two output classes, we generally use softmax. Softmax is basically a generalization of the sigmoid function. Essentially, what the softmax function does is generate an output vector of probabilities where the most likely class is taken as the answer. We define the softmax function as,

$$g(z) = softmax(z) \tag{14}$$

where,

$$g(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}} \tag{15}$$

Then to select the most probable output class, we simply look for the largest probability. Hence we perform,

$$Output = \underset{i}{\text{argmax}}(g(z_i)) \tag{16}$$

Note that we can confirm that this is a valid output distribution because $\sum_i g(z_i) = 1$. Additionally, another interesting thing about softmax is that

since the output probability if proportional to the size of the input in the range $0 \leq g(z) \leq 1$, it is possible to write,

$$\underset{i}{\text{argmax}}(g(z_i)) = \underset{i}{\text{argmax}}(z_i) \tag{17}$$

If the above is true, then it raises the interesting question of why bother to use softmax at all. Unfortunately, to optimize a neural network, it is necessary to have a differentiable function. Since argmax is non-differentiable we prefer softmax. However, note that it is general practice to use use softmax during the training of the network and to use argmax when making inferences with a trained network.

1.4. Loss Functions

When a neural network is training based on a dataset, we use a function to determine how far off the prediction is from the labels provided in the dataset. The function is known as a loss function. When summed over the entirety of the dataset, we determine the overall cost for a particular set of weights. Assuming a dataset of m training examples, this can be represented as,

$$J(W, b) = \frac{1}{m} \sum_{i}^{m} L(\hat{y}_i, y) \tag{18}$$

In general, the term loss function and cost function are used interchangeably since one depends on the other. Hence in this particular section, all loss functions discussed will be written mathematically in terms of its cost function for simplicity.

1.4.1. Mean Squared Error

One of the most basic loss functions, the Mean Squared Error loss function is typically used in regression problems. MSE is defined as,

$$L(\hat{y}, y) = (\hat{y}^{(i)} - y^{(i)})^2 \tag{19}$$

Then as per equation (18) the respective cost function would be,

$$J(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2 \tag{20}$$

In general for classification problems, we use cross-entropy loss functions. Particularly in the case of CNNs, we use the loss functions described in the upcoming sections.

1.4.2. Cross Entropy Loss

This loss is better for classification problems. In the most general form, the Cross Entropy Loss over q output classes is defined as,

$$L(\hat{y}_i, y) = -\sum_{j=1}^{q} y_j \log(\hat{y}_j) \tag{21}$$

While the classical cross-entropy loss is useful, we use a slightly different version for binary classification. In effect, instead of going over q classes, we only iterate over two. In this case, we define it as,

$$L(\hat{y}_i, y) = -\sum_{j=1}^{2} y_j log(\hat{y}_j) = -ylog(\hat{y}) - (1-y)\log(1-\hat{y}) \tag{22}$$

Since we define the cost function as per equation (18), we can then write the cost function as,

$$J(W, b) = \frac{1}{m}\sum_{i=1}^{m} -y^{(i)} log(\hat{y}^{(i)}) - (1-y^{(i)})\log(1-\hat{y}^{(i)}) \tag{23}$$

The loss model described by equation (20) did not take into account the function that actually generates the predictions $\hat{y}$. In that sense, this is the most general form of the cross-entropy loss models. In the next two sections, we go over the specific cases where the activation function used to generate $\hat{y}$ is taken into account when defining the loss. In this way, we arrive at both Categorical Cross-Entropy Loss as well as Binary Cross Entropy Loss.

1.4.3. Categorical Cross-Entropy Loss

Also called the softmax loss model, in this loss model, we assume the function predicting the output classes is a softmax function. In this case, we can define the loss as,

$$L(\hat{y}_i, y) = -\sum_{j=1}^{q} y_j \log(\hat{y}_j) \tag{24}$$

Where $\hat{y}_j$ is defined as given z is the linear output,

$$\hat{y}_j = \frac{e^{z_i}}{\sum_k e^{z_k}} \tag{25}$$

Hence, by substituting we simply get,

$$L(\hat{y}_i, y) = -\sum_{j=1}^{q} y_j \log(\frac{e^{z_i}}{\sum_k e^{z_k}}) \tag{26}$$

In general, this model is preferred for multi-class classification problems. For the specific binary case, since sigmoid is used instead of softmax, we define yet another loss function decribed in the next section.

A quick and rather interesting question however, is what is the difference between sparse categorical cross-entropy loss and categorical cross-entropy loss? Simply put, the only difference is way the true labels are defined. In categorical cross-entropy, we use one-hot encoding for the output labels. On the other hand, for the sparse variation, we use integers instead. For example, for the output labels of three samples, the below equations represent the categorical labels and sparse categorical labels.

$$y_c = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}; y_{sc} = \begin{bmatrix} 2 & 0 & 2 \end{bmatrix} \tag{27}$$

1.4.4. Binary Cross-Entropy Loss

Unlike the last loss mode, this model uses assumes the activation function is a sigmoid function as opposed to a softmax function. In this case, we define the loss as,

$$L(\hat{y}_i, y) = -ylog(\hat{y}) - (1-y)\log(1-\hat{y}) \tag{28}$$

Where we define $\hat{y}_j$ as,

$$\hat{y}_j = \frac{1}{1+e^{-z_j}} \tag{29}$$

This loss model is generally used when there are two output classes for the CNN.

1.5. Backpropagation in a CNN

Now that the losses for the dataset can be calculated, the obvious next question is how do we compute the gradients for optimization? Much like in the case of a MLP, we use back-propagation to compute the gradients.

To simplify the process of finding the gradients, we start by drawing a computational graph for a convolutional layer. A typical convolutional layer

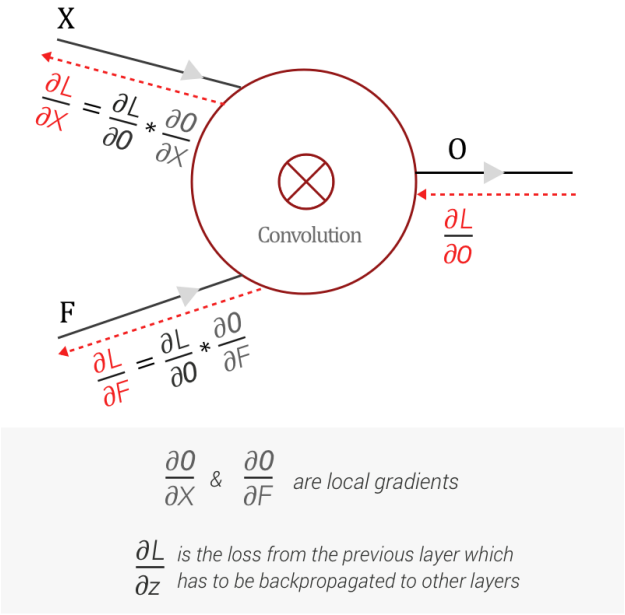

Figure 8: The computational graph for the Convolutional Layer

has two inputs, X the input matrix and F the filter applied to X. Using these, we can draw,

Here, the terms to compute are $\frac{\partial L}{\partial F}$ and $\frac{\partial L}{\partial X}$. We need the first term for updating the filter weights and the second to act as the loss gradient for the layer before the current layer being considered.

To compute $\frac{\partial L}{\partial F}$, we can write the expression,

$$\frac{\partial L}{\partial F_i} = \sum_{k=1}^{M} \frac{\partial L}{\partial O_k} * \frac{\partial O_k}{\partial F_i} \tag{30}$$

Interestingly, since the convolution operation is taking the product between the weights and the corresponding inputs, $\frac{\partial O_k}{\partial F_i}$ simply returns the corresponding input value.

Expanding the above expression and substituting the values of the inputs in, it becomes apparent that the gradient is simply a convolution operation between the derivative of the outputs and the inputs for the current layer. In effect,

$$\frac{\partial L}{\partial F_i} = Conv(\frac{\partial L}{\partial O}, X) \tag{31}$$

Similarly, to find the gradient with respect to X, we perform a full convolution with a 180^o rotated version of the filter F with $\frac{\partial L}{\partial O}$. Both this and the previous gradient computation are summed up in the figure below,

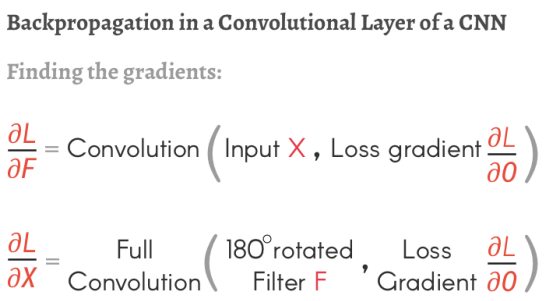

Figure 9: Summary for Backprop in a CNN

1.6. Optimizers

Once the gradients have been computed using backpropagation, we need to update the weights and using the cost function as a reference, find the weights with the minimal overall cost. To do this, we use various optimization algorithms.

1.6.1. Gradient Descent

Perhaps the most commonly used and simplest optimization algorithm, this algorithm simply computes the loss for every data-point in the training set. Assuming W represents the matrix of weights and b represents the bias, gradient descent updates gradients using the following,

$$W := W - \alpha \frac{\partial L}{\partial W} \tag{32}$$

$$b := b - \alpha \frac{\partial L}{\partial b} \tag{33}$$

Here α acts as the hyper-parameter used to control the learning rate of gradient descent.

1.6.2. Mini-Batch Gradient Descent and Stochastic Gradient Descent

In the previous case, we computed and summed the loss for the entire dataset. The weight and bias matrices were only updated after processing the entire training set. Typically, this converges on the minima quicker at the cost of adding noise to the process.

On the other hand, if we update the weights and bias matrices after each training example, it is called stochastic gradient descent.

1.6.3. Gradient Descent With Momentum

While the previous gradient descent algorithms work well, there is the issue of noise, particularly when using mini-batches. The issue with this is that, depending on the learning rate of the algorithm, we may never converge on the correct answer. As a result, it is necessary to generate a method, that reduces the noise in the gradients that are computed from back-propagation.

One method to do this is using Exponential Weighted Averages(or adding a momentum term) to the gradient descent process. This smooths the gradient over the iterations and hence reduces the noise in the gradient. Assuming W represent the weights and b represents the bias, gradient descent with momentum can be formulated as,

$$V_{dw} = \beta V_{dw} + (1 - \beta)\frac{\partial L}{\partial W} \tag{34}$$

$$V_{db} = \beta V_{db} + (1 - \beta)\frac{\partial L}{\partial b} \tag{35}$$

The above represents the momentum terms used to smooth the gradients. The updates to the weights and bias is then computed using,

$$W := W - \alpha V_{dw} \tag{36}$$

$$b := b - \alpha V_{db} \tag{37}$$

1.6.4. RMSProp

Similar to Gradient Descent with Momentum, RMSProp serves as a way to dampen the noise during gradient descent. Additionally, it also addresses one problem with the momentum method. In the momentum method, if the gradients due to one of the features is higher, gradient descent has a tendency to take a much longer route to find the minima. RMSProp addressed this issue by damping the learning rate in different amounts along different feature axis.

So, RMSProp is formulated as,

$$S_{dw} = \beta S_{dw} + (1-\beta)(\frac{\partial L}{\partial W})^2 \tag{38}$$

$$S_{db} = \beta S_{db} + (1-\beta)(\frac{\partial L}{\partial b})^2 \tag{39}$$

$$W := W - \alpha \frac{\frac{\partial L}{\partial W}}{\sqrt{S_{dw}} + \epsilon} \tag{40}$$

$$b := b - \alpha \frac{\frac{\partial L}{\partial b}}{\sqrt{S_{db}} + \epsilon} \tag{41}$$

Note that in the above equations, the squaring operator represents an element-wise squaring. One disadvantage of this approach is that this method is somewhat slower than Gradient Descent with Momentum.

1.6.5. AdaM Optimizer

AdaM combines the two previously mentioned methods for a fast, robust and fairly general optimization algorithm which can be used in most cases. Additionally, while bias correction is often ignored in the previous two optimizers, it is not the case for this optimizer. The AdaM optimizer can be formulated as,

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)\frac{\partial L}{\partial W} \tag{42}$$

$$V_{db} = \beta_1 V_{db} + (1-\beta_1)\frac{\partial L}{\partial b} \tag{43}$$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)(\frac{\partial L}{\partial W})^2 \tag{44}$$

$$S_{db} = \beta_2 S_{db} + (1-\beta_2)(\frac{\partial L}{\partial b})^2 \tag{45}$$

After defining these, we perform bias correction as follows,

$$V_{dw}^c = \frac{V_{dw}}{1-\beta_1^t}; V_{db}^c = \frac{V_{db}}{1-\beta_1^t} \tag{46}$$

$$S_{dw}^c = \frac{S_{dw}}{1-\beta_1^t}; S_{db}^c = \frac{S_{db}}{1-\beta_1^t} \tag{47}$$

Once the bias correction is completed, we can define the weight updates as the following,

$$W := W - \alpha \frac{V_{dw}^c}{\sqrt{S_{dw}} + \epsilon} \tag{48}$$

$$b := b - \alpha \frac{V_{db}^c}{\sqrt{S_{db}} + \epsilon} \tag{49}$$

Note that all of these updates are performed at the end of each mini-batch since we get faster convergence for larger datasets.

2. The Mathematics Of Semantic Segmentation

In typicall CNNs, as we go pass each cnovolutional layer, the dimensions of the image decrease while the channel depth increases. This works fine for classification problems since the convolutional layers are connected to a dense layer at the end. However, when it comes to the probelm of segmentation, this is not sufficient.

During segmentation, we need to know where the object is in the frame. Typically, this is done using a pixel wise classification of the input image. For this to be useful and accurate, we need the output mask containing the segmented image to be of the same size as the input image. One way to do this would be to only perform 'same' convolutions throughout the network. Another and far more popular approach is to use a decoder. So what is a decoder? Essentially, after being downsampled, we need a method to upsample the image to it's original resolution. The below section discusses the methods to do this.

2.0.1. Up-sampling

One approach is to use typical upsampling methods such as nearest neighbour or bed of nails. In nearest neighbour, depending on the output size, we assume all values near the value to be upsampled in the same. On the other hand, 'bed of nails' assumes all neighbouring values to be zero.

Another approach is the idea of max/average unpooling. In max unpooling, we save the position of the max value from prior convolutional layers. This data is then used while upsampling the data in later layers. On the other hand, in average pooling, we redistribute the value to all positions within the window from which the average was computed. A summary of these methods can be seen below.

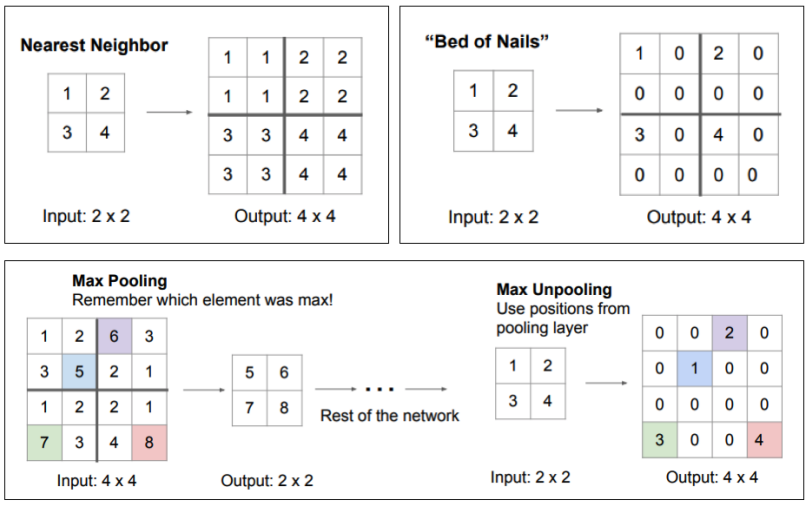

Figure 10: Different methods for upsampling

The methods of up-sampling discussed so for only describe how upsampling is done with no learnable parameters. In the next section, we discuss a more popular method of upsamling that uses transpose convolutions.

2.0.2. Transpose Convolutions

This is by far a more popular method for upsampling because these have learnable parameters. In this method, we multiply the filter with the weight described in the layer to be de-convolved and then we copy these weighted filter values into the respective regions in the upsampled layer.

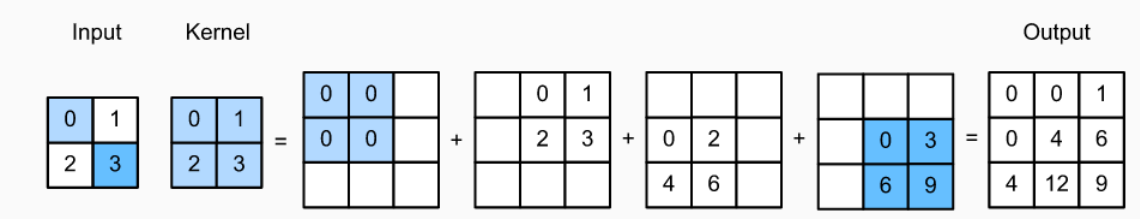

Figure 11: How transpose convolution is performed

The advantage of this method is that you can include trainable weights in

this method. Obviously, since this is being fed into a CNN, this is particularly useful.

2.1. Metrics for Semantic Segmentation

And important aspect of determining the performance of a CNN is it's performance metrics. Generally, you would metrics such as accuracy, recall, F1 score, etc. However, in the case of semantic segmentation, the metrics used are slightly different. These metrics are discussed below.

2.1.1. IOU/Jaccard Index

This is a metric used in semantic segmentation to determine the percentage overlap between our target mask and our prediction output. In effect, it finds the ratio between the number of pixels overlapping in the ground-truth and the prediction and the total number of pixels common to both classes. This can be represented as,

$$IOU = \frac{target \cap prediction}{target \cup prediction} \tag{50}$$

2.1.2. Pixel Accuracy

Pixel accuracy is somewhat more straight forward. Here, from the values computed using the confusion matrix, we determine the pixel-wise accuracy as,

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{51}$$

References

[1] Albawi, Saad Abed Mohammed, Tareq ALZAWI, Saad. (2017). Understanding of a Convolutional Neural Network. 10.1109/ICEngTechnol.2017.8308186.

[2] https://d2l.ai/chapter_convolutional-modern/batch-norm.html

[3] https://medium.com/analytics-vidhya/optimization-algorithms-for-deep-learning-1f1a2bd4c46b

[4] https://towardsdatascience.com/understanding-2d-dilated-convolution-operation-with-examples-in-numpy-and-tensorflow-with-d376b3972b25

[5] http://www.cs.umd.edu/ djacobs/CMSC733/CNN.pdf

[6] https://www.jeremyjordan.me/semantic-segmentation/

[7] https://cs.nju.edu.cn/wujx/teaching/15_CNN.pdf

[8] https://www.jeremyjordan.me/evaluating-image-segmentation-models/

[9] https://web.stanford.edu/class/cs231a/lectures/intro_cnn.pdf

[10] https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

[11] https://cs231n.github.io/convolutional-networks/

[12] https://deepai.org/machine-learning-glossary-and-terms/softmax-layer