CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2022 Marvin Zhang **Discussion 4**

This discussion will cover CNNs, batch normalization, weight initialization, dropout, and ReLUs.

1 Convolutional Neural Networks

Convolutional neural networks[1] (CNN) are a type of neural network architecture that have become the key ingredient for state of the art modern computer vision performance.

They perform operations similar to feed-forward neural networks that we have discussed, but explicitly account for spatial structure in the data, and so are very common for computer vision tasks where inputs are images. That said, CNNs can also be applied to non-image data with similar structure in the input, such as time series or text data (in which case they're taking advantage of temporal structure).

1.1 Convolution (Cross-Correlation) Operator

At the heart of CNNs is the convolution operator. In this discussion, what we refer to as a convolution is actually the **cross-correlation** operator here instead, which is the exact same but with the indexing of the weights in $\mathbf{w}$ inverted. For example, "convolutional" layers in the deep learning library Pytorch are also actually cross-correlations instead, and homework 1 will also similarly have you implement cross-correlation instead of the actual convolution.

To motivate the use of convolutions, we will work through an example of a 1-D convolution calculation to illustrate how convolutions work over a single spatial dimension. Suppose we have an input $\mathbf{x} \in \mathbb{R}^n$, and filter $\mathbf{w} \in \mathbb{R}^k$. We can compute the convolution of $\mathbf{x} \star \mathbf{w}$ as follows:

1. Take your convolutional filter $\mathbf{w}$ and align it with the beginning of $\mathbf{x}$. Take the dot product of $\mathbf{w}$ and the $\mathbf{x}[0 : k-1]$ (using Python-style zero-indexing here) and assign that as the first entry of the output.
2. Suppose we have stride s. Shift the filter down by s indices, and now take the dot product of $\mathbf{w}$ and $\mathbf{x}[s : k-1+s]$ and assign to the next entry of your output.
3. Repeat until we run out of entries in $\mathbf{x}$.

Below, we illustrate a 1D convolution with stride 1.

$$\overbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_k \\ \vdots \\ x_n \end{bmatrix}}^{\text{Input vector } \mathbf{x} \in \mathbb{R}^n} \star \overbrace{\begin{bmatrix} w_1 \\ \vdots \\ w_k \end{bmatrix}}^{\text{Convolutional filter } \mathbf{w} \in \mathbb{R}^k} = \overbrace{\begin{bmatrix} \sum_{i=1}^k w_i x_i \\ \sum_{i=1}^k w_i x_{i+1} \\ \vdots \\ \sum_{i=1}^k w_i x_{i+n-k} \end{bmatrix}}^{\text{Output vector } \mathbf{y} \in \mathbb{R}^{n-k+1}}$$

We see that the output vector is smaller than the input vector ($\mathbb{R}^{n-k-1}$ compared to $\mathbb{R}^n$). A common way to address this is **zero-padding**, in which we append zeros on both ends of the input vector before applying the convolution (note that there are other conventions for zero-padding as well).

[1] Recommended reading: http://cs231n.github.io/convolutional-networks/

Often, we'll be dealing with multiple spatial dimensions (2 spatial dimensions in the case of images). In this case, we would need to slide our filter along all spatial dimensions to construct the output.

Problem 1: Test your know knowledge of convolution dimensions

In this problem, we will run a series of convolution-related operations to better understand how dimensions are affected by convolutions.

1. Suppose you have a $32 \times 32 \times 3$ image (a 32×32 image with 3 input channels). What are the resulting dimensions when you convolve with a $5 \times 5 \times 3$ filter with stride 1 and 0 padding?
2. What if we zero-pad the input by 2?
3. Suppose we now stack 10 of these $5 \times 5 \times 3$ filters and continue to zero pad the input by 2. What is the new shape of the output, and how many parameters are in our filters (not including any bias parameters)?
4. What would be the spatial dimensions after applying a 1×1 convolution? Think about what this does.

Solution 1: Test your know knowledge of convolution dimensions

1. The resulting spatial dimensions are 28×28 (with one output channel).
2. The resulting spatial dimensions are 32×32, so we have preserved the same size as the input image.
3. The resulting outputs are $32 \times 32 \times 10$, with 10 output channels. There are $5 \cdot 5 \cdot 3 = 75$ parameters per filter, so with 10 filters, we have 750 parameters in this layer. Note that, if we did choose to include a bias parameter, then there would be 76 parameters per filter, and so 760 in total.
4. A 1×1 convolution does not change the spatial dimensions. For every spatial location, it performs a linear map of the the input channels pointwise over space. In practice, this is useful for changing the number of channels.

1.2 Convolutions as Matrix Multiplication

We note that convolutions are a linear operation. Recalling linear algebra, any linear map (between finite-dimensional spaces) can be expressed as a matrix, so we will see in this section how to write a convolution as a matrix multiplication.

Problem 2: Expressing convolutions as matrix multiplication

We shall again consider a 1D convolution. Consider an input $\mathbf{x} \in \mathbb{R}^4$ and filter $\mathbf{w} \in \mathbb{R}^3$. Letting $\bar{\mathbf{x}}$ denote the result of zero-padding the input by 1 on each end, what is the matrix W such that

$$\overbrace{W}^{\mathbb{R}^{4\times 6}} \quad \overbrace{\begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}}^{\text{Zero padded input } \bar{\mathbf{x}} \in \mathbb{R}^6} = \bar{\mathbf{x}} * \mathbf{w}?$$

Solution 2: Expressing convolutions as matrix multiplication

Computing the convolution, we see that

$$\bar{\mathbf{x}} * \mathbf{w} = \begin{bmatrix} x_1 w_2 + x_2 w_3 \\ x_1 w_1 + x_2 w_2 + x_3 w_3 \\ x_2 w_1 + x_3 w_2 + x_4 w_3 \\ x_3 w_1 + x_4 w_2 \end{bmatrix}.$$

Writing this out as a matrix multiplication, we obtain

$$W = \begin{bmatrix} w_1 & w_2 & w_3 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & w_3 \end{bmatrix}.$$

We can observe now that the resulting matrix will be very sparse (most entries are 0) if the filter size is much smaller than the input size, corresponding to the fact that such convolutions exploit spatial locality. We also observe that there is a lot of parameter reuse, as the convolutional filter weights are repeated many times throughout the explicit matrix.

This has several implications. First of all, this implies that convolutional layers are less expressive than fully-connected layers (as fully connected layers are represented by arbitrary matrices).

Another important implication stems from the fact that we have very optimized tools for computing matrix multiplications. While a naive implementation of a convolution will require looping over all the spatial dimensions, it will turn out that reformulating the convolution as a matrix multiplication will often be much faster due to these optimizations (for example, the Cythonized im2col function in part 4 of homework 1 essentially does this).

1.3 Backwards Pass for a Convolution

We'll consider the same 1D convolution as before, but without zero-padding for simplicity.

Problem 3: Backwards pass for convolutions

Let $\mathbf{y} = \mathbf{x} * \mathbf{w} \in \mathbb{R}^2$, where $\mathbf{w} \in \mathbb{R}^3$, $\mathbf{x} \in \mathbb{R}^4$. Let $\nabla_{\mathbf{y}} L$ denote the gradient of the loss with respect to the output of the convolution. Compute the gradients of L with respect to $\mathbf{x}$ and $\mathbf{w}$. Can you express the gradients as convolutions themselves?

Solution 3: Backwards pass for convolutions

Let $\delta_i = \frac{\partial L}{\partial y_i}$. We can explicitly write out the partial derivatives with respect to each entry of $\mathbf{x}$.

$$\begin{aligned} \frac{\partial L}{\partial x_1} &= w_1 \delta_1 \\ \frac{\partial L}{\partial x_2} &= w_2 \delta_1 + w_1 \delta_2 \\ \frac{\partial L}{\partial x_3} &= w_3 \delta_1 + w_2 \delta_2 \\ \frac{\partial L}{\partial x_4} &= w_3 \delta_2 \end{aligned}$$

We recognize this as convolution where we zero pad δ by 2 on each end, and convolve with the filter $\tilde{\mathbf{w}}$, where $\tilde{\mathbf{w}}$ reverses the entries of the filter $\mathbf{w}$. (Draw this out for students, explain why sliding the

filter along means that we should convolve the output derivative with $\tilde{\mathbf{w}}$ instead of $\mathbf{w}$).
Now, we can similarly compute the partial derivatives for $\mathbf{w}$

$$\frac{\partial L}{\partial w_1} = \delta_1 x_1 + \delta_2 x_2$$
$$\frac{\partial L}{\partial w_2} = \delta_1 x_2 + \delta_2 x_3$$
$$\frac{\partial L}{\partial w_3} = \delta_1 x_3 + \delta_2 x_4$$

We see that $\frac{\partial L}{\partial \mathbf{w}} = \mathbf{x} \star \boldsymbol{\nabla}_{\mathbf{y}} L$ with no zero-padding.

2 Batch Normalization

The main idea behind Batch Normalization is to transform every sampled batch of data so that they have $\mu = 0$, $\sigma^2 = 1$. Using Batch Normalization typically makes networks significantly more robust to poor initialization. It based on the intuition that it is better to have unit Gaussian inputs to layers at initialization. However, the reason for why batch normalization works is not entirely understood, and there are conflicting views between whether Batch Normalization reduces covariate shift, improves smoothness over the optimization landscape, or other reasons.

In practice, when using batch normalization, we add a BatchNorm layer immediately after ecah FC or convolutional layer, either before or after the non-linearity. The key observation is that normalization is a relatively simple differentiable operation, so we do not add too much additional complexity in the network.

Noticeably, Batch Normalization proceeds by first computing the empirical mean and variance of some mini-batch B of size m from the training set.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} a_i$$
$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(a_i - \mu_B)^2$$

Then, for a layer of network, each dimension, $a^{(i)}$ is normalized appropriately,

$$\bar{a}_i^{(k)} = \frac{a_i^{(k)} - \mu_B^{(k)}}{\sqrt{\left(\sigma_B^{(k)}\right)^2 + \epsilon}}$$

where ϵ is added for numerical stability.

In practice, after normalizing the input, we squash the result through a linear function with learnable scale γ and bias β, so, we have,

$$\bar{a}_i^{(k)} = \frac{a_i^{(k)} - \mu_B^{(k)}}{\sqrt{\left(\sigma_B^{(k)}\right)^2 + \epsilon}}\gamma + \beta$$

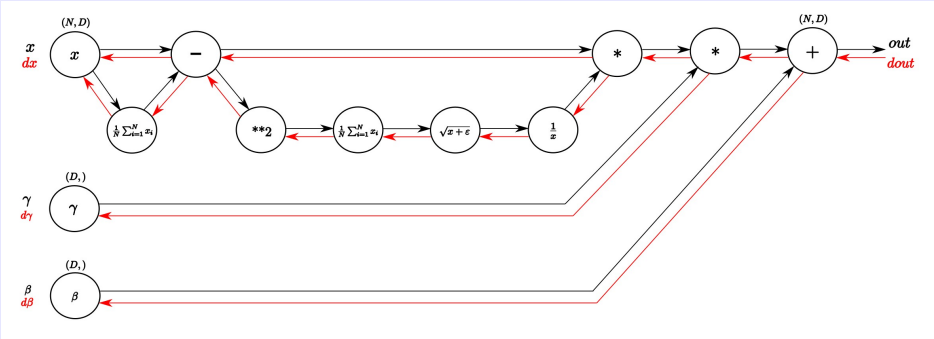
Intuitively, γ and β allows us to restore the original activation if we would like, and during training, they can learn other distribution that would be better initialization than standard Gaussian.

Problem 4: Examining the BatchNorm Layer

1. Draw out the computational graph of the BatchNorm layer
2. Given some derivatives $\frac{\partial f}{\partial y_i}$, the derivative of the output of the BatchNorm layer, compute the derivatives with respect to parameters γ, β

Solution 4: Examining the BatchNorm Layer

Note: Students should already have derived this in homework.

1. 

2. For convenience of notation, for $i = \{1, \ldots, m\}$ where m represents the batch size, let

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

and

$$y_i = \gamma \hat{x_i} + \beta$$

First, we calculate the derivative with respect to γ,

$$\begin{aligned}\frac{\partial f}{\partial \gamma} &= \frac{\partial f}{\partial y_i} \cdot \frac{\partial y_i}{\partial \gamma} \\ &= \sum_{i=1}^{m} \frac{\partial f}{\partial y_i} \cdot \hat{x_i}\end{aligned}$$

We sum from $i = 1$ to m since we have m items in the batch.

Second, we calculate the derivative with respect to β. Similar to our previous calculation,

$$\begin{aligned}\frac{\partial f}{\partial \beta} &= \frac{\partial f}{\partial y_i} \cdot \frac{\partial y_i}{\partial \beta} \\ &= \sum_{i=1}^{m} \frac{\partial f}{\partial y_i}\end{aligned}$$

Problem 5: (Challenge) Examining the BatchNorm Layer

From the previous question, given some derivatives $\frac{\partial f}{\partial y_i}$, the derivative of the output of the BatchNorm layer, compute the derivatives with respect to input x_i.
Please note the derivation can be tedious, but it is still a very good exercise!

Solution 5: (Challenge) Examining the BatchNorm Layer

We calculate the derivative with respect to x_i. To do this, we note,

$$\frac{\partial f}{\partial x_i} = \frac{\partial f}{\partial \hat{x_i}} \frac{\partial \hat{x_i}}{\partial x_i} + \frac{\partial f}{\partial \mu} \frac{\partial \mu}{\partial x_i} + \frac{\partial f}{\partial \sigma^2} \frac{\partial \sigma^2}{\partial x_i}$$

Let us compute the individual pieces,

$$
\begin{aligned}
\frac{\partial f}{\partial \hat{x_i}} &= \frac{\partial f}{\partial y_i} \cdot \frac{\partial y_i}{\partial \hat{x_i}} \\
&= \frac{\partial f}{\partial y_i} \cdot \gamma \\
\frac{\partial \hat{x_i}}{\partial x_i} &= \frac{1}{\sqrt{\sigma^2 + \epsilon}} \\
\frac{\partial f}{\partial \sigma^2} &= \frac{\partial f}{\partial \hat{x}} \cdot \frac{\partial \hat{x}}{\partial \sigma^2} \\
&= \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x_i}} \cdot \frac{\partial \hat{x_i}}{\partial \sigma^2} \\
&= \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x_i}} \cdot (x_i - \mu) \cdot (-0.5) \cdot (\sigma^2 + \epsilon)^{-1.5} \\
&= -0.5 \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x_i}} \cdot (x_i - \mu) \cdot (\sigma^2 + \epsilon)^{-1.5} \\
\frac{\partial \sigma^2}{\partial x_i} &= \frac{2(x_i - \mu)}{m} \\
\frac{\partial f}{\partial \mu} &= \frac{\partial f}{\partial \hat{x}} \cdot \frac{\partial \hat{x}}{\partial \mu} + \frac{\partial f}{\partial \sigma^2} \cdot \frac{\partial \sigma^2}{\partial \mu} \\
&= \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x_i}} \cdot \frac{-1}{\sqrt{\sigma^2 + \epsilon}} + \frac{\partial f}{\partial \sigma^2} \cdot \frac{1}{m} \sum_{i=1}^{m} -2(x_i - \mu) \\
&= \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x_i}} \cdot \frac{-1}{\sqrt{\sigma^2 + \epsilon}} + \frac{\partial f}{\partial \sigma^2} \cdot (-2) \cdot (0) \\
&= \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{x_i}} \cdot \frac{-1}{\sqrt{\sigma^2 + \epsilon}} \\
\frac{\partial \mu}{\partial x_i} &= \frac{1}{m} \\
\frac{\partial f}{\partial x_i} &= \frac{\partial f}{\partial \hat{x_i}} \cdot \frac{1}{\sqrt{\sigma^2 + \epsilon}} + \frac{\partial f}{\partial \mu} \cdot \frac{1}{m} + \frac{\partial f}{\partial \sigma^2} \cdot \frac{2(x_i - \mu)}{m}
\end{aligned}
$$

3 Dropout

Definition 1 (Dropout)**.** *Dropout is a popular technique for regularizing neural networks by randomly removing nodes with probability* $1 - p_{keep}$ *in the forward pass. However, the model is unchanged at test time.*

Intuition Dropout can be thought of as representing an ensemble of neural networks, since each forward pass is effectively a different neural network, since random nodes are removed.

Activation Scaling A caveat about dropout is that we must divide the activation by p, since we do not change the model at test time, but we notice that none of our dimensions will then be forced to 0. Below is sample code to demonstrate how Dropout works in practice for a 3-layer network.

```
import numpy as np
from scipy.special import softmax

def dropout_train(X, p):
    """
    Forward pass for a 3-layer network.
    NOTE: For simplicity, we do not include backwards pass or parameter update

    X: Input
    p: Probability of keeping a unit active (e.g., higher p leads to less dropout)
    """
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p
    H1 *= U1 # Drop the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p
    H2 *= U2 # Drop the activations
    out = np.dot(W3, H2) + b3
    return out

def predict(X):
    """ Forward pass at test time """
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
    return out
```

Problem 6: Dropout Review

Explain why Dropout could improve performance and when we should use it

Solution 6: Dropout Review

Dropout removes random activations during training, which prevents the model from overfitting to specific features, and have redundant representations. Dropout should be used to make network more robust and lower variance.

4 Weight Initialization

One of the reasons for poor model performance can be attributed to poor weight initialization. In class, we discussed ways to ensure activations are initialized reasonably and do not grow or shrink much in later layers (for example, Gaussian random weights or Xavier initialization). In practice, using batch normalization or layer normalization makes weight initialization not as critical to get perfect, and these techniques combined with Xavier initialization are often enough.

Problem 7: Deriving Xavier Initialization

Assume we are training a fully-connected neural network with identity activation functions (i.e. there are no non-linearities after each linear layer, so the entire network is a linear function), We furthermore assume that weights and inputs are i.i.d. and centered at zero, and biases are initialized as zero. We would like the magnitude of the variance to remain constant with each layer. Derive the Xavier Initialization, which initializes each weight as,

$$W_{ij} = \mathcal{N}\left(0, \frac{1}{D_a}\right)$$

where D_a is the dimensionality of a

Solution 7: Deriving Xavier Initialization

Note that, since we assume bias is initialized at 0,

$$z_i = \sum_{j=1}^{D_a} W_{ij} a_j$$

We can then compute $\mathsf{Var}(z_i)$,

$$\begin{aligned}
\mathsf{Var}(z_i) &= \mathsf{Var}\left(\sum_{j=1}^{D_a} W_{ij} a_j\right) \\
&= \sum_{j=1}^{D_a} \mathsf{Var}(W_{ij} a_j) && \text{Variance of independent sum} \\
&= \sum_{j=1}^{D_a} \mathbb{E}[W_{ij}]^2 \mathsf{Var}(a_j) + \mathbb{E}[a_j]^2 \mathsf{Var}(W_{ij}) + \mathsf{Var}(W_{ij})\mathsf{Var}(a_j) \\
&= \mathsf{Var}(W_{ij})\mathsf{Var}(a_j) D_a \\
&\iff \mathsf{Var}(W_{ij}) = \frac{1}{D_a}
\end{aligned}$$

Note that if our neural network used tanh activations, this derivation may still approximately hold, because tanh approximates the identity for small inputs.

5 Aside: ReLU Activations and its Relatives

Definition 2 (ReLU Activation)**.** *ReLU (Rectified Linear Unit) Activation is defined as,* $ReLU(x) = \max(0, x)$*, and is a popular activation function.*

For $x < 0$, ReLU is equal to the zero function. This can have its down-side for training: the gradient is zero for $x < 0$, meaning we make no update when $x < 0$. Other variants of ReLU replace the zero function with a different function for $x < 0$. For example:

- **Leaky ReLU**. Leaky ReLU replaces the $x < 0$ piece of ReLU with a linear function with a small slope.

$$\text{Leaky ReLU}(x) = \begin{cases} \alpha x & x < 0 \\ x & x = 0 \end{cases},$$

  where α is chosen to be small.

- **ELU**. The ELU (Exponential Linear Unit) replaces the $x < 0$ piece of RELU with an exponential function.

$$\text{ELU}(x) = \begin{cases} \alpha(e^x - 1) & x < 0 \\ x & x = 0 \end{cases},$$

  where α is a chosen hyperparameter.

Please note we did not cover the above explicitly in lecture, but they are good knowledge to have.

Problem 8: (Review) Forward and Backward Pass for ReLU

Compute the output of forward pass of a ReLU layer with input x as given below:

$$y = \text{ReLU}(x)$$

$$x = \begin{bmatrix} 1.5 & 2.2 & 1.3 & 6.7 \\ 4.3 & -0.3 & -0.2 & 4.9 \\ -4.5 & 1.4 & 5.5 & 1.8 \\ 0.1 & -0.5 & -0.1 & 2.2 \end{bmatrix}$$

With the gradients with respect to the outputs $\frac{dL}{dy}$ given below, compute the gradient of the loss with respect to the input x using the backward pass for a ReLU layer:

$$\frac{dL}{dy} = \begin{bmatrix} 4.5 & 1.2 & 2.3 & 1.3 \\ -1.3 & -6.3 & 4.1 & -2.9 \\ -0.5 & 1.2 & 3.5 & 1.2 \\ -6.1 & 0.5 & -4.1 & -3.2 \end{bmatrix}$$

Solution 8: (Review) Forward and Backward Pass for ReLU

Applying the ReLU treats every entry independently, zeroing it out if the entry is less than 0.

$$y = \begin{bmatrix} 1.5 & 2.2 & 1.3 & 6.7 \\ 4.3 & 0 & 0 & 4.9 \\ 0 & 1.4 & 5.5 & 1.8 \\ 0.1 & 0 & 0 & 2.2 \end{bmatrix}$$

Similarly, the backwards pass zeros out entries of the same entries of the gradient that were zeroed

out in the forward pass.

$$\frac{dL}{dx} = \begin{bmatrix} 4.5 & 1.2 & 2.3 & 1.3 \\ -1.3 & 0 & 0 & -2.9 \\ 0 & 1.2 & 3.5 & 1.2 \\ -6.1 & 0 & 0 & -3.2 \end{bmatrix}$$

Problem 9: ReLU Potpourri

1. What advantages does using ReLU activations have over sigmoid activations?
2. ReLU layers have non-negative outputs. What is a negative consequence of this problem? What layer types were developed to address this issue?

Solution 9: ReLU Potpourri

1. (1) Computing ReLU and its gradient is more computationally efficient than Sigmoid. (2) It reduces the likelihood of vanishing gradient problems, since the derivative of the sigmoid function is always less than 1, and multiplying gradients over multiple layers will lead to quick convergence of sigmoid function to 0. However, this is less of an advantage, given that one can use Batch Normalization to centralize inputs
2. ReLU suffers from the Dying ReLU problem, where this unit always outputs 0, no matter what the input. Once the ReLU ends up in this state, it is unlikely to recover, since its gradient is also 0, and GD methods will not alter its weights. Other layer types that were developed include Leaky-ReLU, ELU.