Deep Learning

Hieu Nguyen

Deep Learning

Hieu Nguyen

4th October 2025

For Yiren, or anyone who is willing to listen to me ramble.

<div align="right">HIEU</div>

Contents

1

Gradients And Weight Initialization

"You lift bro?"

– Loss function weights

I am going to assume that you already have a basic understanding of the notes that I made on machine learning, and so this will probably go by really quickly. This, if anything, should be a continuation of my machine learning notes, but since deep learning in itself is a deep subject (pun intended), maybe it would be appropriate for me to make a separate set of notes. The structure of these notes will be slightly different, too. I will focus on a lot more math than I normally would in machine learning; I will actually try to prove results instead of just saying that things are true. With that being said, we will first begin talking about different gradient methods. Then after we talk about gradient methods, we will discuss how it is typically applied in deep learning via the backpropagation algorithm. Afterwards, we will talk about weight/parameter initialization.

1.1 Stochastic Gradient Descent

I assume that we are already familiar with gradient descent. We mentioned before that gradient descent is the preferred method of minimizing the loss function and this is due to its cheap cost: solving the normal equation and taking an inverse is a very expensive operation. However, as it turns out, even when the number of parameters scale to millions and billions, gradient descent can be slow. Computer scientists are cheapskates, and so will always try their best to minimize the time to compute something. On top of that, what could potentially happen is that gradient descent could trap our solution in a local minimum. Remember: the surface of the loss function may not always be convex, and it most definitely will not always be quadratic. Thus, if we were to use regular gradient descent, we could be stuck in a local minimum and never know. Gradient descent, then, is sensitive to our initial conditions; which is something that we do NOT want. We can remedy this by introducing a bit of "randomness." Instead of taking the full gradient and following the direction, we actually will randomly select a subset of the full set of weights and compute the gradient using that. What this means is that the gradient won't always be in the direction of lowest descent; it might sometimes go upwards for a while. My claim is that on average, gradient descent and stochastic gradient descent both converge to the local minima.

Stochastic Gradient Descent

1. Select a subset of the parameters you are trying to estimate $\mathscr{B}_t$ (call this the mini-batch, or sometimes just batch)
2. Compute the gradient
$$\nabla_\theta = \sum_{i\in\mathscr{B}_t} \frac{\partial L_i[\theta_t]}{\partial\theta}$$
3. $\theta_{t+1} = \theta_t - \alpha\nabla_\theta$

with $L_i[\theta_t]$ being the loss function. My claim before was that even though SGD does not always go in the direction of the minimum, it, on average, still converges to the true minimum. Unfortunately, I do not have time to prove this at the moment because I just read a proof for stochastic gradient descent and it took me like 30 minutes. I think I will include the proof in these notes on my own time. Let us try to set up an example problem and then solve it using stochastic gradient descent. Let us start with the linear regression problem, where the loss function was defined as

$$L[\boldsymbol{\theta}] = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}) = \sum_{i,j=1}(y_i - X_{ij}\theta_j)^2 \tag{1.1}$$

Then we can find the gradient as

$$\nabla_j^{(\theta)} = -2\sum_{i,k=1}(y_i - X_{ik}\theta_k)X_{ij} \tag{1.2}$$

$$\implies \nabla_\theta = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})$$

Now, instead of summing over all possible θ, we choose at random a subset. For example, we might choose the set of indices to be $i \in (1, 7, 2, 10)$. This would truncate our $\boldsymbol{X}$ matrix

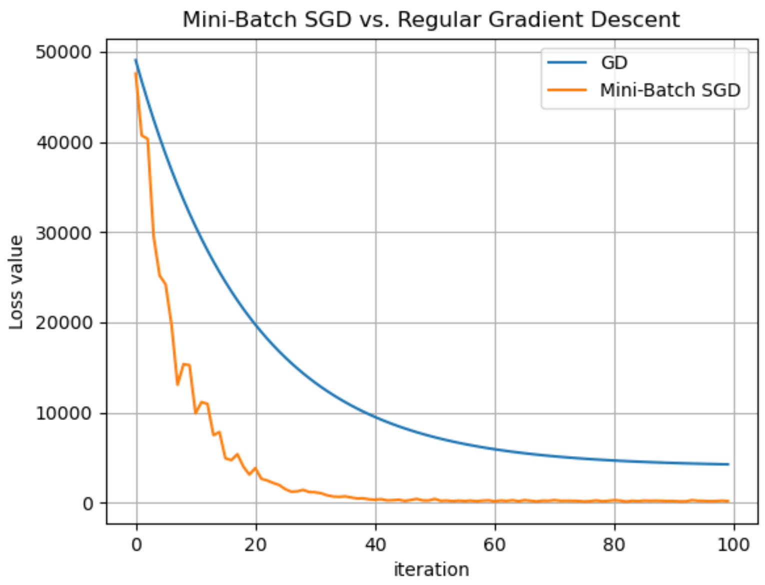


Figure 1.1: A plot of the loss function of mini batch stochastic gradient descent vs. full batch gradient descent. As we can see, SGD actually approaches the minimum faster.

as well as our $\boldsymbol{y}$ vector. If we wanted to write it out explicitly, it would be

$$\nabla_j^{(\theta)} = (y_1 - X_{1k}\theta_k)X_{1j} + (y_7 - X_{7k}\theta_k)X_{7j} + (y_2 - X_{2k}\theta_k)X_{2j} + (y_{10} - X_{10k}\theta_k)X_{10j}$$

Where we have assumed that the index k is being summed over, of course. From there, we can update our θ according to bullet point 3 of the stochastic gradient descent algorithm. We will include a code snippet for the stochastic gradient descent algorithm in python

```python
import numpy as np

# Generate synthetic data
x = np.random.uniform(0,6,1000)
epsilon = np.random.normal(0,2,1000) ## the noise term
y = 3*x+1 + epsilon ## The model

## Now implement stochastic gradient descent. With only 2 weights, it probably won
    't be very different
## Since we know we have 1000 data points, let's truncate the feature matrix to
    only 50 data points.
num_rows = 1000

## 1 batch is 50 indices, and 1 epoch is when 20 different batches have been used
```

```python
epochs = 500
batchsize = 100
w = np.array([0,-6.]).reshape(-1,1) ## Don't forget to initialize the weights!
alpha = .05 ## This is the step size
lossSGD = [] ## a list to keep track of the loss function at each step
wsteps = [] ## a list to keep track of the weights at each step

## Begin the for loop
for epoch in range(epochs):

    batchnumber = 0 ## initialize to 0 as we run through all of the indices
    random_indices = np.random.choice(num_rows, size=1000, replace=False) ## random indices

    for batch in range(int(num_rows/batchsize)):
        batchnumber += 50
        minibatchindex = random_indices[int(batchnumber-50):int(batchnumber)]

        truncatedX = X[minibatchindex, :] ## truncating the feature matrix using the random indices
        truncatedy = y[minibatchindex, :] ## truncating the output vector using the random indices

        grad = 1/n * (2*truncatedX.T)@(truncatedX@w - truncatedy) ## compute the gradient
        w += -alpha * grad ## update the gradient

        wsteps.append(w.flatten())
        lossSGD += [(truncatedy-truncatedX@w).T @ (truncatedy-truncatedX@w)]

    print('epoch:',str(epoch+1)+"/"+str(epochs)," loss="+str(lossSGD[-1]))
```

And when we test this on linear regression data

```python
print('the weight vector is:',w)
```

In practice, we would like stochastic gradient descent to have a learning rate schedule. This is so that when the number of iterations is small, the algorithm will explore the parameter space (large α). When we approach the minimum, we would like it to stay in the vicinity of the minimum, and thus desire small α.

Problem 1.

Show that regular gradient descent CANNOT escape a local minimum

Solution:
Suppose that $f(x)$ is a function that admits at least 1 local minimum. If gradient descent were to enter such an area, the gradient near the local minimum would $\to 0$. If α is small, then there would be no gradient update and θ_i cannot change. If α is large, then the algorithm would force the θ_i to oscillate back and forth around the minimum.

1.2 Momentum

The speed ups that we get from stochastic gradient descent can be improved; and the improvement comes from adding memory to the algorithm. Both regular and stochastic gradient descent are memory-less in the fact that where you go tomorrow does not depend on where you were yesterday. The problem that could theoretically arise from this is that SGD will zig-zag towards the minimum, taking more time. Instead of allowing the algorithm to be "memory-less," we allow the gradient update an "inertia" term; that is, we compute the next step in the gradient, but instead of going there immediately, the **momentum** from the previous iteration carries through.

$$\theta_{t+1} = \theta_t - \alpha \left(\beta \boldsymbol{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial L_i[\theta_t]}{\partial \theta} \right) \tag{1.3}$$

The $\beta \boldsymbol{m}_t$ term is the carry-over momentum from the previous step and the derivative term is still the gradient. The β parameter is the parameter that controls how "smooth" the path is.

Problem 2.

Show that the momentum term $\boldsymbol{m}_t$ is an infinite weighted sum of the gradients at the previous iterations and derive an expression for the coefficients (weights) of that sum.

Solution:
If we declare the term in the parenthesis of (1.3) as $\boldsymbol{m}_{t+1}$

$$\boldsymbol{m}_{t+1} = \beta \boldsymbol{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial L_i[\theta_t]}{\partial \theta}$$

We can find the previous iteration of the momentum $\boldsymbol{m}_t$

$$\boldsymbol{m}_t = \beta \boldsymbol{m}_{t-1} + (1-\beta) \sum_{i \in \mathscr{B}_t} \frac{\partial L_i[\theta_{t-1}]}{\partial \theta}$$

We can take the above term and recursively add it back to $\boldsymbol{m}_{t+1}$

$$\boldsymbol{m}_{t+1} = \beta \left(\beta \boldsymbol{m}_{t-1} + (1-\beta) \sum_{i \in \mathscr{B}_t} \frac{\partial L_i[\theta_{t-1}]}{\partial \theta} \right) + (1-\beta) \sum_{i \in \mathscr{B}_t} \frac{\partial L_i[\theta_t]}{\partial \theta}$$

If we repeat this recursion an infinite number of times, we see that we get

$$\boldsymbol{m}_{t+1} = \beta^t \boldsymbol{m}_0 + \sum_{n=0}^{t} \beta^{t-n} (1-\beta) \sum_{i \in \mathscr{B}_t} \frac{\partial L_i[\theta_n]}{\partial \theta}$$

We can assume without loss of generality that $\boldsymbol{m}_0 = 0$ and therefore we get

$$\boldsymbol{m}_{t+1} = \sum_{n=0}^{t} \beta^{t-n} (1-\beta) \sum_{i \in \mathscr{B}_t} \frac{\partial L_i[\theta_n]}{\partial \theta} \tag{1.4}$$

which is an infinite weighted sum of the gradients, with the weight being $\beta^{t-n}(1-\beta)$.

We can implement the momentum term in python. Suppose we have the usual linear regression problem. We will visualize what SGD does and what it does with the momentum term. We remember that the mini-batch gradient can be denoted as $\nabla_j^{(\theta)}$. We would need to initialize

$$\boldsymbol{m}_{t+1}^j = \beta \boldsymbol{m}_t^j + (1-\beta) \nabla_t^j(\theta)$$

where the j is the index for the j-th parameter. The code for the momentum SGD function is below

```python
## Define the momentum SGD function
def momentumSGD(X, y,epochs,batchsize,alpha,beta):
    """This is the SGD algorithm with momentum. It takes in a series of parameters
    that define the algorithm (like number of epochs, batchsize, etc) and returns
    the final weights, the loss function at each step, and the values of weights
    at each step"""

    num_rows = len(y)
    lossSGD = [] ## a list to keep track of the loss function at each step
    wsteps = [] ## a list to keep track of the weights at each step
    w = np.array([0,-6.]).reshape(-1,1) ## Don't forget to initialize the weights!
    m = np.ones(len(w)).reshape(-1,1)
```

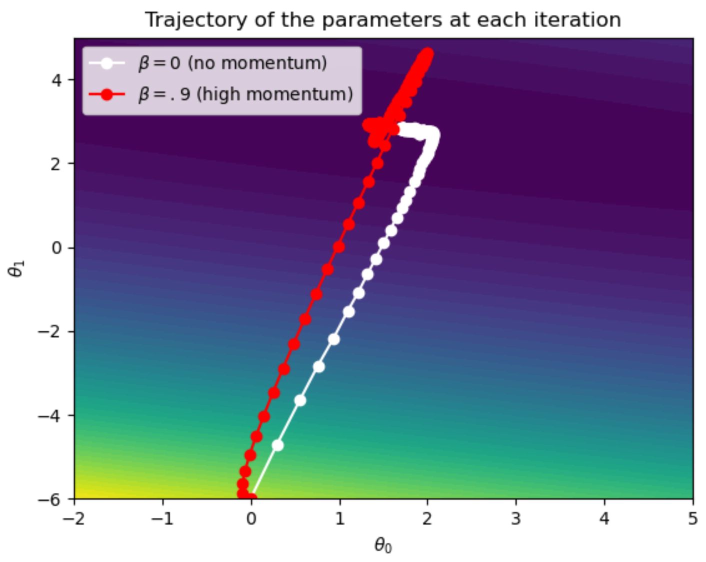


Figure 1.2: The plotted contour is a heatmap of the linear regression loss function. We plot stochastic gradient descent and SGD with momentum starting at the same position. As we can see, for the same amount of iteration, the momentum term actually gets to the minimum faster

```
    ## Begin the for loop
    for epoch in range(epochs):
        if epoch == 0:
            wsteps.append(w.flatten())

        batchnumber = 0 ## initialize to 0 as we run through all of the indices
        random_indices = np.random.choice(num_rows, size=num_rows, replace=False)
    ## random indices

        for batch in range(int(num_rows/batchsize)):
            batchnumber += 50
            minibatchindex = random_indices[int(batchnumber-50):int(batchnumber)]

            truncatedX = X[minibatchindex, :] ## truncating the feature matrix
    using the random indices
            truncatedy = y[minibatchindex, :] ## truncating the output vector
    using the random indices

            grad = 1/n * (2*truncatedX.T)@(truncatedX@w - truncatedy) ## compute
    the gradient

            ##This is where we will be adding the momentum
```

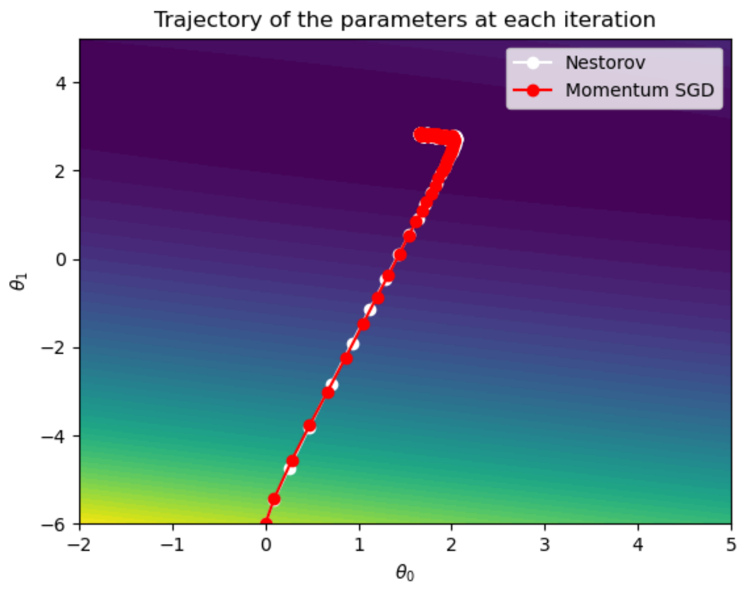


Figure 1.3: We now compare momentum SGD with Nesterov SGD with all the same initial parameters and we see that the path that it takes is actually slightly faster (white is leads red).

```
            m = beta * m + (1-beta)*grad
            w += -alpha * m ## update the gradient with the momentum now

            wsteps.append(w.flatten())
            lossSGD += [(truncatedy-truncatedX@w).T @ (truncatedy-truncatedX@w)]

        ## We can also implement early stopping as to not waste computational
    resources.
        ## Just comment it out if you don't want to use it

        #if np.abs(lossSGD[-1]-lossSGD[-2])/lossSGD[-2] <= .01:
            #print('Stable minimum has been found at epoch:',epoch,"!")
            #break


        print('epoch:',str(epoch+1)+"/"+str(epochs)," loss="+str(lossSGD[-1]))
    return w, wsteps, lossSGD
```

1.2.1 Nesterov's Accelerated Momentum

We make a quick note on Nesterov's accelerated momentum, which he calls "the fastest descent algorithm." His idea is literally that of stochastic gradient descent with momentum, but instead of computing the gradient at the current point and then having the momentum carry over, his algorithm calculates the gradient AHEAD of the path. This will give us a smoother path, and as it turns out, an even faster convergence. The modification is quite simple

$$\theta_{t+1} = \theta_t - \alpha\left(\beta \boldsymbol{m}_t + (1-\beta)\sum_{i\in\mathscr{B}_t}\frac{\partial L_i[\theta_t + \alpha\cdot \boldsymbol{m}_t]}{\partial\theta}\right) \tag{1.5}$$

```python
%matplotlib inline
## Define the momentum SGD function
def NestorovSGD(X, y,epochs,batchsize,alpha,beta):
    """This is the SGD algorithm with momentum. It takes in a series of parameters
    that define the algorithm (like number of epochs, batchsize, etc) and returns
    the final weights, the loss function at each step, and the values of weights
    at each step"""

    num_rows = len(y)
    lossSGD = [] ## a list to keep track of the loss function at each step
    wsteps = [] ## a list to keep track of the weights at each step
    w = np.array([0,-6.]).reshape(-1,1) ## Don't forget to initialize the weights!
    m = np.ones(len(w)).reshape(-1,1)

    ## Begin the for loop
    for epoch in range(epochs):
        if epoch == 0:
            wsteps.append(w.flatten())

        batchnumber = 0 ## initialize to 0 as we run through all of the indices
        random_indices = np.random.choice(num_rows, size=num_rows, replace=False)
    ## random indices

        for batch in range(int(num_rows/batchsize)):
            batchnumber += 50
            minibatchindex = random_indices[int(batchnumber-50):int(batchnumber)]

            truncatedX = X[minibatchindex, :] ## truncating the feature matrix
    using the random indices
            truncatedy = y[minibatchindex, :] ## truncating the output vector
    using the random indices

            grad = 1/n * (2*truncatedX.T)@(truncatedX@(w+alpha*m) - truncatedy) ##
     modified to gradient so that it computes ahead
```

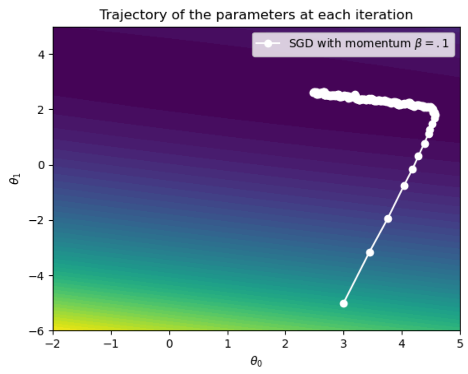
```python

            ##This is where we will be adding the momentum
            m = beta * m + (1-beta)*grad
            w += -alpha * m ## update the gradient with the momentum now

            wsteps.append(w.flatten())
            lossSGD += [(truncatedy-truncatedX@w).T @ (truncatedy-truncatedX@w)]

        ## We can also implement early stopping as to not waste computational resources.
        ## Just comment it out if you don't want to use it

        #if np.abs(lossSGD[-1]-lossSGD[-2])/lossSGD[-2] <= .01:
            #print('Stable minimum has been found at epoch:',epoch,"!")
            #break


        print('epoch:',str(epoch+1)+"/"+str(epochs)," loss="+str(lossSGD[-1]))
    return w, wsteps, lossSGD
```

1.3 ADAM

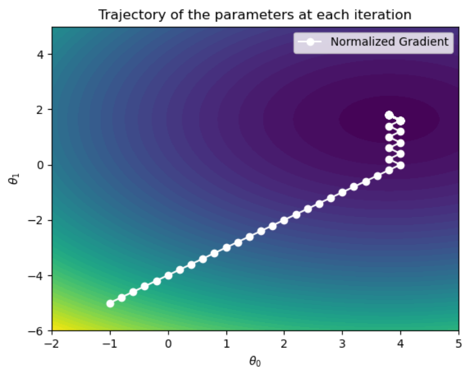
One of the properties of gradient descent in general is that the direction of the trajectory is dominated by the direction with the largest gradient. What this means is that suppose we have a surface such that in 1 direction, the slope is steeper than the other. The "particle" will travel towards that steeper direction first and then to the other direction afterwards. We can see this briefly in figure 1.3, where it goes northeast for a while and once it has reached near the bottom of the surface, it makes a sudden change going leftwards. The potential problem that this could have is that if the direction of the gradient is dominated in one direction, it might spend a lot of time going in the other direction once it gets near the bottom. An example of this can be seen in figure 1.4, where the particle takes large steps going down but then takes many small steps going in an almost perpendicular direction. The solution to this would be: don't just go straight up and then rightward, why not move up and right at the same time in equal proportions? Well, in order for them to move in equal proportions, we would need to normalize the gradients. Let us define the momentum term with $\beta = 0$ and a new parameter vector $\boldsymbol{v}$ to encode the square of the gradient

$$\boldsymbol{m}_{t+1} = \frac{\partial L[\theta]}{\partial \theta}$$

$$\boldsymbol{v}_{t+1} = \left(\frac{\partial L[\theta]}{\partial \theta}\right)^2$$



(a) Non-normalized gradient



(b) normalized gradient

Figure 1.4: plot (a) depicts the trajectory of a particle in parameter space via stochastic gradient descent with momentum. Notice that the particle takes large and few steps going northeast (down the bowl) and then many small steps in a direction almost perpendicular. In figure (b), when we normalize the gradient, the particle is no longer dominated by 1 direction, and takes step sizes that are all evenly spaced in parameter space. (for this figure, we centered the data to have mean 0 and scaled it to have the covariance matrix be the identity. This is to ensure that the loss surface is not hyper anisotropic.

and then the parameter update would be

$$\theta_{t+1} = \theta_t - \alpha \frac{\boldsymbol{m}_{t+1}}{\sqrt{\boldsymbol{v_{t+1}}} + \epsilon}$$

where the square and the square rooting are all elementwise. The ϵ is there to ensure that we don't have a division by 0 when we are close to the minimum. In the case of $\beta = 0$, what this essentially does is make it so that the gradient vector is a vector of all values ± 1, reducing the domination of one direction over the other. Since $\beta = 0$ this is example and we are doing full batch gradient descent, the learning rate is a constant. However, because our learning rate is a constant, when we approach the minimum, we will just oscillate back and forth (peep figure 1.4). We can introduce an effective learning rate schedule by introducing momentum into this normalized gradient, and when we do so, this is called **adaptive moment estimation**, or just **ADAM.** Introducing momentum to our $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$ gives

$$\boldsymbol{m}_{t+1} = \beta \boldsymbol{m}_t + (1-\beta)\nabla_t(\theta) \tag{1.6}$$

$$\boldsymbol{v}_{t+1} = \gamma \boldsymbol{v}_t + (1-\gamma)\left(\nabla_t(\theta)\right)^2 \tag{1.7}$$

And this makes our parameter updates being

$$\theta_{t+1} = \theta_t - \alpha \frac{\boldsymbol{m}_{t+1}}{\sqrt{\boldsymbol{v}_{t+1}} + \epsilon} \tag{1.8}$$

There is a subtle issue. In problem 2, we showed that adding momentum is effectively the same thing as a weighted infinite sum of the gradients with the weight being $\beta^{t-n}\beta$. In the limit that $t \to \infty$ and $n \to 0$, we see that the weights are effectively 0.

Problem 3.

As we saw in problem 2, adding momentum is the same thing as saying

$$\boldsymbol{m}_{t+1} = \sum_{n=1}^{t} \beta^{t-n}(1-\beta) \sum_{i \in \mathscr{B}_t} \frac{\partial L_i[\theta_n]}{\partial \theta}$$

With the weights being $\beta^{t-n}(1-\beta)$. Show that at early times, the **total weight** of the gradients is $1-\beta^t$, and therefore are artificially small.

Solution:
Ultimately, we want to find the total weights, which is effectively doing the sum

$$\sum_{n=0}^{t} \beta^{t-n}(1-\beta)$$

We can factor out the $1-\beta$ and compute the sum of just β^{t-n}. The sum of just beta can be rewritten as

$$(1-\beta)\sum_{n=0}^{t-1} \beta^n$$

due to symmetry (all we did was re-index the β's as well as state that 1+2+3+4 is the same as 4+3+2+1 because previously the power of β was decreasing; we just rewrote it so that now it is increasing. This changes nothing). The sum above is a geometric series, and we know what the partial sum of a geometric series is

$$S_n = \frac{1-\beta^n}{1-\beta}$$

Subbing the partial sum back into the previous equation gets us

$$1 - \beta\left(\frac{1-\beta^t}{1-\beta}\right) = 1 - \beta^t \tag{1.9}$$

In the limit of early times, when t is small, the magnitude of the total weight is effectively 0

Problem 3 highlights the issue with raw momentum, which at early times weights are vanishing. We can normalize the total size of the weights to unity by dividing equations (1.6) and (1.7) by (1.9) and we get the new momentum updates

$$\tilde{\boldsymbol{m}}_{t+1} = \frac{\boldsymbol{m}_{t+1}}{1-\beta^t} \tag{1.10}$$

$$\tilde{\boldsymbol{v}}_{t+1} = \frac{\boldsymbol{v}_{t+1}}{1-\gamma^t} \tag{1.11}$$

And finally, our parameter updates will be

$$\theta_{t+1} = \theta_t - \alpha\frac{\tilde{\boldsymbol{m}}_{t+1}}{\sqrt{\tilde{\boldsymbol{v}}_{t+1}} + \epsilon} \tag{1.12}$$

```python
## Define the momentum SGD function
def ADAM(X, y,epochs,batchsize,alpha,beta,gamma):
    """This is the SGD algorithm with momentum. It takes in a series of parameters
    that define the algorithm (like number of epochs, batchsize, etc) and returns
    the final weights, the loss function at each step, and the values of weights
    at each step"""
    num_rows = len(y)
    if num_rows%batchsize != 0:
        return "Error: Batchsize incompatible with matrix (hush I'll fix it later)
    "

    lossSGD = [] ## a list to keep track of the loss function at each step
    wsteps = [] ## a list to keep track of the weights at each step
    w = np.array([-1,-5.]).reshape(-1,1) ## Don't forget to initialize the weights
    !
    m = np.zeros(len(w)).reshape(-1,1)
    v = np.zeros(len(w)).reshape(-1,1)
    t = 1 ## This is to keep track of iteration so that we can use it in
    normalizing m and v
    e = 10e-20 ## Here just to make sure we don't get division by 0

    ## Begin the for loop
    for epoch in range(epochs):
        if epoch == 0:
            wsteps.append(w.flatten())

        batchnumber = 0 ## initialize to 0 as we run through all of the indices
        random_indices = np.random.choice(num_rows, size=num_rows, replace=False) ## random indices

        for batch in range(int(num_rows/batchsize)):
            batchnumber += batchsize
            minibatchindex = random_indices[int(batchnumber-batchsize):int(batchnumber)]

            truncatedX = X[minibatchindex, :] ## truncating the feature matrix using the random indices
            truncatedy = y[minibatchindex, :] ## truncating the output vector using the random indices

            grad = 1/batchsize * (2*truncatedX.T)@(truncatedX@w - truncatedy) ## compute the gradient

            ##This is where we will be adding the momentum
            m = (beta* m + (1-beta)*grad) #/(1-beta**t)
            v = (gamma*v + (1-gamma)*grad**2) #/(1-gamma**t)
            vhat = v / (1-gamma**t)
            mhat = m/(1-beta**t)

            w += -alpha * m/(np.sqrt(v)+e) ## update the gradient with the momentum now

            wsteps.append(w.flatten())
            lossSGD += [(truncatedy-truncatedX@w).T @ (truncatedy-truncatedX@w)]

            t += 1 ## Update the tick

        ## We can also implement early stopping as to not waste computational resources.
        ## Just comment it out if you don't want to use it

        #if np.abs(lossSGD[-1]-lossSGD[-2])/lossSGD[-2] <= .01:
            #print('Stable minimum has been found at epoch:',epoch,"!")
            #break

        ## You can uncomment this if you want to see the loss at each epoch.
        #print('epoch:',str(epoch+1)+"/"+str(epochs)," loss="+str(lossSGD[-1]))
    return w, wsteps, lossSGD
```

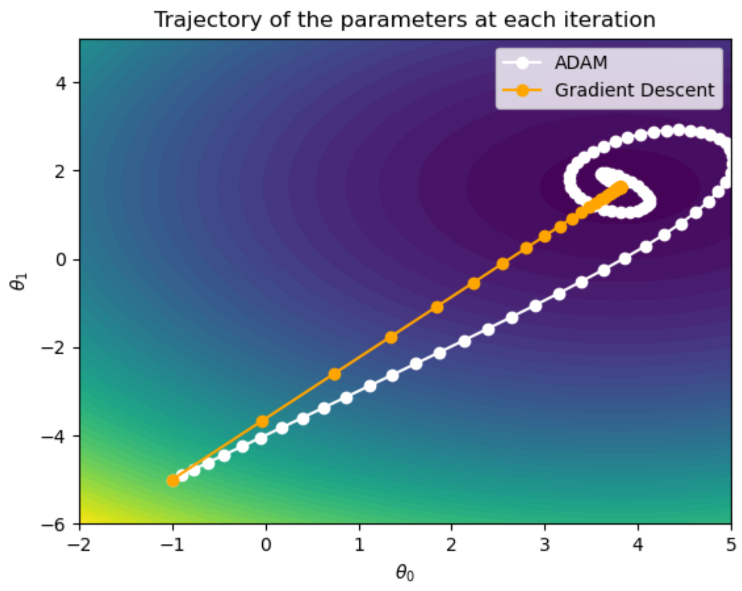


Figure 1.5: Plot of ADAM's trajectory through the parameter space. Adding momentum to the normalized gradient makes it spiral into the minimum.

Problem 4.

A surface is convex if the eigenvalues of the Hessian $\mathbf{H}[\boldsymbol{\theta}]$ are positive everywhere. In this case, the surface has a unique minimum, and optimization is easy. Find an algebraic expression for the Hessian matrix,

$$\mathbf{H}[\boldsymbol{\theta}] = \begin{bmatrix} \frac{\partial^2 L}{\partial \theta_0^2} & \frac{\partial^2 L}{\partial \theta_0 \partial \theta_1} \\ \frac{\partial^2 L}{\partial \theta_1 \partial \theta_0} & \frac{\partial^2 L}{\partial \theta_1^2} \end{bmatrix}$$

for the linear regression model. Prove that this function is convex by showing that the eigenvalues are always positive. This can be done by showing that both the trace and the determinant of the matrix are positive.

Solution:
Remember that the loss function for simple linear regression is

$$L(\theta_0, \theta_1) = \sum_{i=1}^{n} (y_i - \theta_0 - \theta_1 X_i)^2$$

If we take second derivatives of this, we see very simply that

$$\frac{\partial^2 L}{\partial \theta_0^2} = \sum_{i=1}^{n} 2 = 2n \qquad \frac{\partial^2 L}{\partial \theta_0 \partial \theta_1} = \frac{\partial^2 L}{\partial \theta_1 \partial \theta_0} = 2\sum_{i=1}^{n} X_i$$

$$\frac{\partial^2 L}{\partial \theta_1^2} = 2\sum_{i=1}^{n} X_i^2$$

making the Hessian

$$\mathbf{H}[\boldsymbol{\theta}] = \begin{bmatrix} 2n & 2\sum_i X_i \\ 2\sum_i X_i & 2\sum_i X_i^2 \end{bmatrix}$$

One key thing to note is that the entries of the Hessian matrix are all positive. We know that the eigenvalues of a 2×2 matrix can be given as

$$\lambda_1 \lambda_2 = \det(A) \quad \lambda_1 + \lambda_2 = \mathrm{Tr}(A)$$

The determinant of the Hessian matrix is

$$\det(\mathbf{H}) = 4n\sum_{i=1}^{n} X_i^2 - 4\left(\sum_{i=1}^{n} X_i\right)^2$$

If we factor out a $4/n^2$, we see that this is nothing more than just the sample variance. The sample variance is always positive thus makes the determinant positive. A positive determinant either means that both λ_1 and λ_2 are positive or negative. However, we see that the race of the Hessian is

$$\mathrm{Tr}(\mathbf{H}) = 2n + 2\sum_{i=1}^{n} X_i^2$$

which is always a positive number. Thus, the trace is positive. With the aforementioned relationship between trace and eigenvalues, this ensures that both λ_1 and λ_2 are positive. This shows that the loss surface for simple linear regression is convex.

1.4 Weight Initialization

In deep learning, we will use stochastic gradient descent with ADAM on models much more complex than the simple linear regression model; we will be using it in neural networks that use the backpropagation algorithm to learn the model (revisit my machine learning notes if you are unfamiliar with backpropagation). For very complicated models,

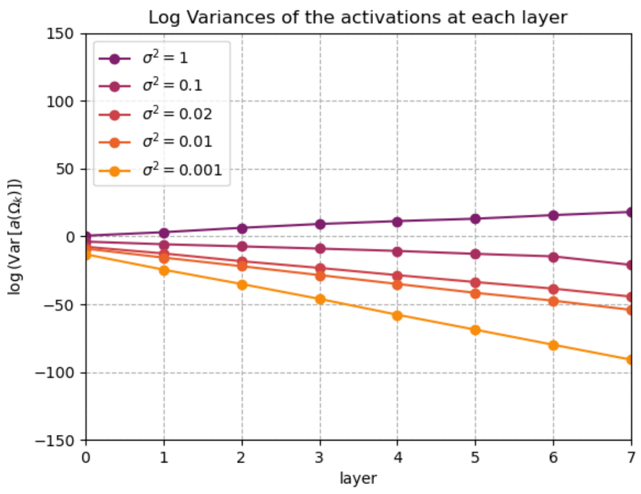


Figure 1.6: The variances of the activation functions going through the forward pass as a function of the layer. This plot is plotted in log-scale. Notice that growth/decay of the variance happens exponentially.

we will inevitably have more than just 2 parameters to initialize. As it turns out, our choice of initialization *matters*. Let's try to see what we mean by this before we delve into the math.

Suppose that for each weight in our neural network $\boldsymbol{\Omega_k}$ is initialized as a centered normal random variable with variance σ^2 and the biases $\boldsymbol{b_k}$ are initialized to zeros. What we want to know is what effect does this have on the activations and the gradients in our model? Looking at figure 1.6, we see that if we initialize different variances, going through a network with 7 layers and 4 hidden units each has the variances of the activations blow up for certain values of the variance ($\sigma^2 = 1$) while other values have the variance decaying exponentially ($\sigma^2 = .001$). And this is just the forward pass, so imagine what would happen if we throw this in the backpropagation algorithm; we would have that the variances of the gradients blow up or vanish even more as it backpropagates! This is known as the **exploding/vanishing** gradient problem, and it's a problem because this can affect the stability and speed of our model.

1.4.1 The Forward Pass

Having seen *what* the problem is, let us now try to understand *why* it happens. Let us look at the variances of consecutive pre-activations. We will define

$$\boldsymbol{h}^k = a\left[\boldsymbol{f}^{k-1}\right] \quad \text{output at k-th hidden layer} \tag{1.13}$$

$$\boldsymbol{f}^k = \boldsymbol{b}^k + \boldsymbol{\Omega}^k \boldsymbol{h}^k \quad \text{k-th pre-activation} \tag{1.14}$$

where $\boldsymbol{b}^k$ is the bias vector. If we wanted to find the variance of the subsequent pre-activation, we will need to compute the mean and second moment using the identity

$$\text{Var}[X] = \mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2 \tag{1.15}$$

Let's find the mean first, because it is easy. Element-wise expectation of the subsequent pre-activation gives

$$\begin{aligned}
\mathbb{E}\left[f_i^k\right] &= \mathbb{E}\left[b_i^k + \sum_{j=1}^{n} \Omega_{ij}^k h_j^k\right] \\
&= \mathbb{E}\left[b_i^k\right] + \mathbb{E}\left[\sum_{j=1}^{n} \Omega_{ij}^k h_j^k\right] \quad \text{(linearity of expectation)} \\
&= \mathbb{E}\left[b_i^k\right] + \sum_{j=1}^{n} \mathbb{E}\left[\Omega_{ij}^k h_j^k\right] \quad \text{(interchange of limits)} \\
&= \mathbb{E}\left[b_i^k\right] + \sum_{j=1}^{n} \mathbb{E}\left[\Omega_{ij}^k\right] \mathbb{E}\left[h_j^k\right] \quad \text{(independence)} \\
&= 0 \quad \text{(initialization is a centered normal random variable)}
\end{aligned}$$

In the fourth line we used the assumption that the different h^k are independent from each other. The above calculation shows (maybe as we would expect) that the expectation of every pre-activation is 0. Before we continue with finding the second moment, we will the following result.

Problem 5.

Suppose that we have X be a symmetric random variable. Find the second moment of $\text{ReLU}(X)$.

Solution:
Remember that the ReLU function is defined piecewise as

$$\text{ReLU}(x) = \begin{cases} x & x > 0 \\ 0 & \text{else} \end{cases}$$

Then the second moment of the ReLU would be

$$\begin{aligned}
\mathbb{E}\left[\mathrm{ReLU}(X)^2\right] &= \int_{-\infty}^{\infty} \mathrm{ReLU}(x)^2 f_X(x)\, dx \\
&= \int_{0}^{\infty} x^2 f_X(x)\, dx \quad \text{(only positive values survive)} \\
&\boxed{= \frac{1}{2}\mathbb{E}\left[X^2\right]}
\end{aligned}$$

Let us now take the second moment of the pre-activation.

$$\mathrm{Var}\left[f_i^k\right] = \mathbb{E}\left[\left(f_i^k\right)^2\right] = \mathbb{E}\left[\left(b_i^k + \sum_{j=1}^{n} \Omega_{ij}^k h_j^k\right)^2\right]$$

Expanding the quadratic and interchanging the limits gives

$$\mathrm{Var}\left[f_i^k\right] = \mathbb{E}[\left(b_i^2\right)^k] + \sum_{\mu=1}^{n} 2\mathbb{E}\left[b_i^k \Omega_{i\mu}^k h_\mu^k\right] + \sum_{l=1}^{n}\sum_{j=1}^{n} \mathbb{E}\left[\Omega_{il}^k \Omega_{ij}^k h_l^k h_j^k\right]$$

During initialization, all of the biases are set as 0. This makes the first term on the right vanish. Using independence and the expectation of Ω being 0, the second term vanishes. All that we are left with is

$$\mathrm{Var}\left[f_i^k\right] = \sum_{l=1}^{n}\sum_{j=1}^{n} \mathbb{E}\left[\Omega_{il}^k \Omega_{ij}^k h_l^k h_j^k\right]$$

This sum can be split into 2 pieces; one where the index $j = l$ and the other when the indices don't equate

$$\begin{aligned}
\mathrm{Var}\left[f_i^k\right] &= \sum_{j=l}^{n} \mathbb{E}\left[\left(\Omega_{ij}^2\right)^k \left(h_j^2\right)^k\right] + \sum_{j\neq l}^{n(n-1)} \mathbb{E}\left[\Omega_{il}^k \Omega_{ij}^k h_l^k h_j^k\right] \\
&= \sum_{j=l}^{n} \underbrace{\mathbb{E}\left[\left(\Omega_{ij}^2\right)^k\right]}_{=\sigma^2} \mathbb{E}\left[\left(h_j^2\right)^k\right] + \cancelto{0}{\sum_{j\neq l}^{n(n-1)} \mathbb{E}\left[\Omega_{il}^k\right] \mathbb{E}\left[\Omega_{ij}^k\right] \mathbb{E}[h_l^k] \mathbb{E}[h_j^k]} \quad \text{(independence)} \\
&= \left(\sigma^2\right)^k \sum_{j=1}^{n} \mathbb{E}\left[\left(h_j^2\right)^k\right]
\end{aligned}$$

Now it is our job to find the second moment of the hidden layer. We know that the hidden layer is a result of passing a linear transformation through a ReLU activation (equation 1.13).

But we saw from problem 5 that the second moment of a symmetric random variable being passed through the ReLU function is just half the second moment of the untransformed random variable. Thus

$$\begin{aligned}
\left(\sigma^2\right)^k \sum_{j=1}^{n} \mathbb{E}\left[\left(h_j^2\right)^k\right] &= \left(\sigma^2\right)^k \sum_{j=1}^{n} \mathbb{E}\left[\text{ReLU}\left(\cancelto{0}{b_i^{k-1}} + \Omega_{ij}^{k-1} h_j^{k-1}\right)^2\right] \quad \text{(initialize bias to 0)} \\
&= \left(\sigma^2\right)^k \sum_{j=1}^{n} \frac{1}{2}\left(\sigma^2\right)^{k-1} \quad \text{(result of problem 5 and } \mathbb{E}[\Omega] = 0\text{)} \\
&= \boxed{\frac{1}{2} n \left(\sigma^2\right)^k \left(\sigma^2\right)^{k-1}} \qquad (1.16)
\end{aligned}$$

So we see that variances of subsequent layers are compounded with the variances of the previous layers! Now it makes sense why we would have an exploding/vanishing gradient problem; if we initialize our variances too small or too large, they will compound exponentially. Now, this equation also tells us that to stabilize the gradients, it's not as simple as just choosing "regular sized variances"; we see in figure 1.6 that even when we initialize $\sigma^2 = 1$, we would still see exponential blow up. Equation (1.6) tells us that in order to stabilize the variances, we need to initialize the variance of the next layer as

$$\left(\sigma^2\right)^k = \frac{2}{n} \qquad (1.17)$$

where n is the dimension of the hidden layer. It is only in the special case of constant hidden layer size that n is a fixed constant; if we had varying hidden layer sizes, then we would need to account for that. We will do so by writing the dimension of the hidden layer as D_k instead, making the above

$$\left(\sigma^2\right)^k = \frac{2}{D_k} \qquad (1.18)$$

This will keep the variance stable throughout the forward pass.

1.4.2 The Backwards Pass

The argument for how we would initialize the variances for the backwards pass is the exact same, so we will just outline the argument and see how it equates. Remember that in during the backpropagation algorithm, the gradients are computed by taking the derivatives of the loss function (at the end of the model) with respect to the parameter of interest. Since the neural net is nested and connected, what we really have is a recursive chain rule.

$$\frac{\partial \ell}{\partial \boldsymbol{b}^k} = \frac{\partial \ell}{\partial \mathbf{f}^k} \qquad (1.19)$$

$$\frac{\partial \ell}{\partial \boldsymbol{\Omega}^k} = \frac{\partial \ell}{\partial \mathbf{f}^k} \left(\mathbf{h}^T\right)^k \tag{1.20}$$

$$\frac{\partial \ell}{\partial \mathbf{f}^{k-1}} = \frac{\partial \ell}{\partial a} \odot \left(\left(\boldsymbol{\Omega}^T\right)^k \frac{\partial \ell}{\partial \mathbf{f}^k} \right) \tag{1.21}$$

where $\odot$ denotes elementwise multiplication of the vectors, a is the activation function, and the index k runs over the layers of the neural network. If we initialize all the biases to 0, then equation (1.18) is no longer relevant. As for the gradients of $\boldsymbol{\Omega}$, we have

$$\frac{\partial \ell}{\partial \boldsymbol{\Omega}^{k-1}} = \left(\frac{\partial \ell}{\partial a} \odot \left(\left(\boldsymbol{\Omega}^T\right)^k \frac{\partial \ell}{\partial \mathbf{f}^k} \right) \right) \left(\mathbf{h}^T\right)^{k-1}$$

In the case where the activation is ReLU, we would have the derivative be the Heaviside theta function $\Theta(x)$

$$\frac{\partial \ell}{\partial \boldsymbol{\Omega}^{k-1}} = \left(\Theta(\mathbf{f}^{k-1}) \odot \left(\left(\boldsymbol{\Omega}^T\right)^k \frac{\partial \ell}{\partial \mathbf{f}^k} \right) \right) \left(\mathbf{h}^T\right)^{k-1}$$

Backpropagation contains (recursively) the vector $\partial \ell / \partial \mathbf{f}^k$, so to make our lives a bit easier, we will call this term $\boldsymbol{\delta}^k$ and write in index notation

$$\frac{\partial \ell}{\partial \Omega^{k-1}_{il}} = \sum_{j=1}^{n} \Theta\left(f_i^{k-1}\right) \Omega^k_{ji} \delta^k_j \; h_l^{k-1}$$

Now, if we want to find the variance of the gradient, we need to first find the expectation

$$\begin{aligned}
\mathbb{E}\left[\frac{\partial \ell}{\partial \Omega^k_{il}}\right] &= \mathbb{E}\left[\sum_{j=1}^{n} \Theta\left(f_i^{k-1}\right) \Omega^k_{ji} \delta^k_j \; h_l^{k-1}\right] \\
&= \sum_{j=1}^{n} \mathbb{E}\left[\Theta\left(f_i^{k-1}\right)\right] \cancelto{0}{\mathbb{E}\left[\Omega^k_{ji}\right]} \mathbb{E}[h_l^{k-1}] \quad \text{(independence)} \\
&= 0 \quad (\text{centering of } \Omega)
\end{aligned}$$

Great! Now this is looking just like the forward pass.

Problem 6.

Let X be a centered, symmetric random variable. Find the second moment of $\Theta(X)$.

Solution:

Remember that the Heaviside theta function is defined as

$$\Theta(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{else} \end{cases}$$

Thus, doing the expectation integral is nothing more than just truncating the domain of the probability integral

$$\boxed{\mathbb{E}\left[\Theta(X)^2\right] = \int_{-\infty}^{\infty} \Theta(x)^2 f_X(x)\, dx = \int_{-\infty}^{\infty} \Theta(x) f_X(x)\, dx = \int_{0}^{\infty} f_X(x) = \frac{1}{2}}$$

where we have used the fact that $\Theta(x)^2 = \Theta(x)$.

With the expectation of the gradient being 0, we have

$$\begin{aligned}
\mathrm{Var}\left[\frac{\partial \ell}{\partial \Omega_{il}^k}\right] &= \mathbb{E}\left[\frac{\partial \ell}{\partial \Omega_{il}^k}\right] = \mathbb{E}\left[\left(\sum_{j=1}^{n} \Theta\left(f_i^{k-1}\right) \Omega_{ji}^k \delta_j^k\ h_l^{k-1}\right)^2\right] \\
&= \cancelto{1/2}{\mathbb{E}\left[\Theta\left(f_i^{k-1}\right)^2\right]}\ \mathbb{E}\left[\left(\sum_{j=1}^{n} \Omega_{ji}^k \delta_j^k\ h_l^{k-1}\right)^2\right] \qquad \text{(expand and use independence)} \\
&= \frac{1}{2}\mathbb{E}\left[\left(\sum_{j=1}^{n} \Omega_{ji}^k \delta_j^k\ h_l^{k-1}\right)^2\right] \qquad \text{(Result from problem 6)} \qquad (1.22)
\end{aligned}$$

And this goes almost exactly like the forward pass. The only difference is the included δ_j^k term in the expectation. If we define the variance of $\delta_j^k = v_\delta^k$, then equation equation (1.21) becomes

$$\mathrm{Var}\left[\frac{\partial \ell}{\partial \Omega_{il}^k}\right] = \frac{1}{2} n \left(\sigma^2\right)^k v_\sigma^k \left(\sigma^2\right)^{k-1} \qquad (1.23)$$

To keep the variance effectively a constant, we would need to initialize

$$\left(\sigma^2\right)^k = \frac{2}{D_k} \qquad (1.24)$$

where D_k is the number of hidden layers in the k-th layer. This initialization is called the **He initialization**. We should make a comment about the D_k in the forward pass versus the backwards pass. Because fundamentally both algorithms go in opposite directions of the architecture, D_k has subtly different interpretations. For the forward pass, it uses the width of the *previous* layer (called fan-in layer) while in the backwards pass uses the *next* layer.

2

Generalization, Regularization, and Transfer Learning

"To all subjects of Ymir, my name is Error Jaeger"

– Error Jaeger

We've already spent a decent amount of time in my machine learning notes talking about sources of error in our models, overfitting, and regularization. I don't want to go into too much detail about these ideas because I feel like it would be redundant. What we will do instead is look at the bigger picture: how and why does a DNN generalize? We will look more in depth on these topics while also exploring modern-ish developments. The first thing that we are going to do is lay the statistical framework for error and then revisit the bias-variance tradeoff to see theoretically what we would expect from our model and what we actually see in practice. We will examine the peculiar case of the double descent phenomenon and discuss the model's ability to generalize. One fascinating and exciting model for generalized learning is transfer learning, and we will explore the ramifications of that.

2.1 Statistical Decision Theory

The backbone for the analysis of error and model complexity is statistical decision theory. Traditionally, statistical decision theory is about choosing the "best" (optimal) estimator by comparing statistical procedures. While the methodologies used in classical statistics vs.

machine learning are different (the problems they are trying to solve are slightly different), the fundamental question of "how good is our model?" is still the same.

2.1.1 Loss Functions and Risk

The measure of error between our model prediction and the true population estimate is encoded by the **loss function.** We've known about this since forever; the entirety of machine learning can practically be summed up as doing gradient descent on some loss function. The common and uncommon loss functions used are

$$\begin{aligned}
L(\theta,\hat{\theta}) &= |\theta-\hat{\theta}| && L_1 \text{ norm}\\
L(\theta,\hat{\theta}) &= |\theta-\hat{\theta}|^2 && \text{Squared error (regression)}\\
L(\theta,\hat{\theta}) &= |\theta-\hat{\theta}|^p && L_p \text{ norm}\\
L(\theta,\hat{\theta}) &= -\sum \theta \log\left(\hat{\theta}\right) && \text{cross-entropy (classification)}\\
L(\theta,\hat{\theta}) &= \int \log\left(\frac{f(x;\theta)}{f(x;\widehat{\theta})}\right) f(x;\theta)dx && \text{Kullback-Leibler distance}
\end{aligned}$$

Equipped with a loss function, we can define the risk of an estimator $\hat{\theta}$ as the average of the loss function

$$\boxed{R(\theta,\hat{\theta}) = \mathbb{E}\left[L\left(\theta,\hat{\theta}\right)\right]} \tag{2.1}$$

Problem 1.

Consider the linear regression problem. We know that the maximum likelihood estimator for the coefficients in the regression problem is given as

$$\hat{\boldsymbol{\theta}} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

Calculate the risk of using squared error loss with this estimator.

Solution:
Originally, when we found the estimator $\hat{\boldsymbol{\theta}}$ we minimized the squared error loss function. This question is effectively asking us to plug our estimator back into the loss function to find the average value of the loss. Using (2.1)

$$\mathbb{E}\left[\left(\boldsymbol{\theta}-\hat{\boldsymbol{\theta}}\right)^2\right]$$

$$= \mathbb{E}\left[\left(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}\right)^T \left(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}\right)\right]$$
$$= \mathbb{E}\left[\boldsymbol{\theta}^T\boldsymbol{\theta} - 2\boldsymbol{\theta}^T\hat{\boldsymbol{\theta}} + \hat{\boldsymbol{\theta}}^T\hat{\boldsymbol{\theta}}\right]$$

We know that $\hat{\boldsymbol{\theta}}$ is an unbiased estimator (if you didn't know this, it's from the fact that the randomness in the output y comes from white noise, which has mean 0. Plugging this back into the estimator just gives us $\boldsymbol{\theta}$), thus when we distribute the expectation

$$\mathbb{E}\left[\left(\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}\right)^2\right] = \mathbb{E}\left[\hat{\boldsymbol{\theta}}^T\hat{\boldsymbol{\theta}}\right] - \boldsymbol{\theta}^T\boldsymbol{\theta}$$

and this is nothing more than just the variance of $\hat{\boldsymbol{\theta}}$. But the variance of $\hat{\boldsymbol{\theta}}$ comes from the variance of $\boldsymbol{y}$. We know from linear models that the variance of the model output comes from white noise with constant variance. What this tells us is that the risk of this estimator with the squared error loss can be boiled down to just literal noise in the data.

2.1.2 Comparing Loss Functions

We saw in machine learning and in problem 1 that oftentimes our choice of an estimator comes from our choice of the loss function. If we wanted to compare the effectiveness of 2 estimators, we are really just comparing their risk functions. In general, however, we do not get uniform dominance of one risk function over the other. To see this, consider the setup in problem 1. We saw that the risk function for the estimator $\hat{\boldsymbol{\theta}}$ is just the variance of a model; a constant. Now suppose that instead of using the MLE, we use a far-fetched estimator; a constant c. Computing the risk for this estimator gives

$$R(\boldsymbol{\theta}, 3) = \mathbb{E}\left[(\boldsymbol{\theta} - 3)^2\right] = (\boldsymbol{\theta} - 3)^2$$

The risk function is, unsurprisingly, a quadratic function. If we compared the 2 risk functions, we can clearly see that neither estimator uniformly dominate the other (figure 2.1). For some region between $\pm\sigma$, the risk function for the constant estimator is smaller than the MLE, while outside this region, it is larger. Perhaps this means that comparing the risk functions is not the optimal method; but this was almost obvious from the get-go that if we compared functions, we might run into the possibility of non-uniform dominance. To avoid this, we look at summaries of the risk function instead. We define the **maximum risk** to be

$$R_{\max}\left(\hat{\theta}\right) = \sup_{\theta} R\left(\theta, \hat{\theta}\right) \tag{2.2}$$

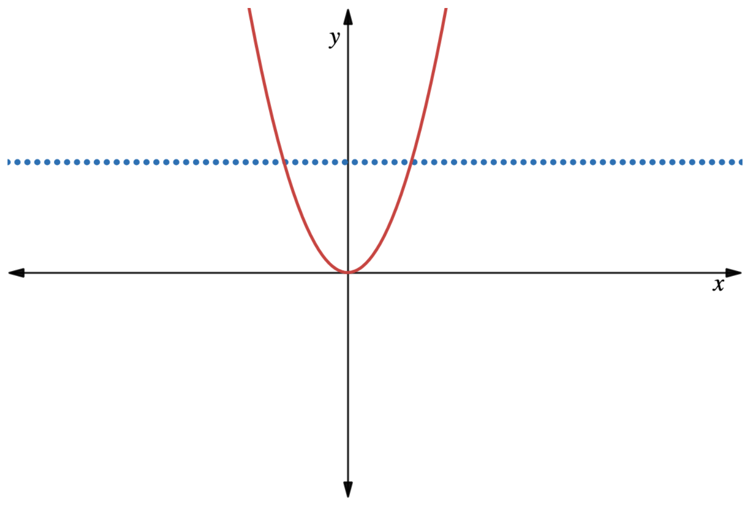


Figure 2.1: The risk function of the constant estimator (linear regression) plotted in red with the risk function for the MLE plotted in blue. For some region $\pm\sigma$, the red risk function will be smaller than the MLE's.

and the **Bayes risk** to be

$$R_{\text{Bayes}}\left(f, \hat{\theta}\right) = \int R\left(\theta, \hat{\theta}\right) f(\theta)\, d\theta \tag{2.3}$$

where $f(\theta)$ is a prior distribution on θ. Whichever one you choose is a choice up to you. Re-examining the 2 risk functions from before, we see that if we calculate the maximum risk between the 2 estimators, the maximum risk for the MLE is a constant while the maximum risk for the constant is infinite. Between the 2 estimators, we should choose the MLE because it has the lower maximum risk. What we have stumbled upon is called the **minimax rule**

$$\sup_{\theta} R\left(\theta, \hat{\theta}\right) = \inf_{\theta_\star} \sup_{\theta} R\left(\theta, \theta_\star\right) \tag{2.4}$$

which is the decision rule that minimizes the maximum risk. Similarly if we were to minimize the Bayes risk, that would be called **Bayes' Rule**.

$$R_{\text{Bayes}}\left(f, \hat{\theta}\right) = \inf_{\theta_\star} R_{\text{Bayes}}\left(f, \theta_\star\right) \tag{2.5}$$

In both cases, the infimum is happening over *all estimators.* This means that in practice, finding the minimax estimator analytically is *very difficult* if not impossible to do. One can develop algorithms to estimate what the minimax estimator should be, and that is an entire field of study on its own.

Problem 2.

Even 1-number summaries like the maximum risk are imperfect. Let us explore an example. Suppose we have $X_1, .., X_n$ be Bernoulli data with unknown parameter p. Compare 2 estimators

$$\hat{p}_1 = \frac{1}{n}\sum_{i=1} X_i \quad \hat{p}_2 = \frac{\sum_i X_i + \sqrt{n/4}}{2\sqrt{n/4} + n}$$

With the former being the unbiased MLE and the latter being pulled from thin air. Find the minimum of the maximum risk between the 2, using the squared error loss. Also find the Bayes risk with uniform prior on p. Is one always better than the other?

Solution:
Since we know that $\hat{p}_1$ is the unbiased MLE, that means that the MSE decomposition (as alluded to in problem 1) is

$$R(p, \hat{p}_1) = \text{Var}\left[\hat{p}_1\right] = \frac{1}{n^2}\sum_{i=1}^{n} \text{Var}\left[X_i\right] = \frac{p(1-p)}{n}$$

With the last equality coming from the i.i.d. property. To maximize this, we just take a derivative and set it equal to 0. Since this is just a quadratic in p, we find that the maximum happens at $p = 1/2$ and thus the maximum risk is

$$R_{\max}(p, \hat{p}_1) = \frac{1}{4n}$$

What about the second estimator? Using the MSE decomposition, we find

$$\text{MSE} = \text{Var}\left[\hat{p}_2\right] + \text{bias}^2(\hat{p}_1)$$

The variance of the estimator is easiest to compute, so let's start with that.

$$\begin{aligned}\text{Var}\left[\frac{\sum_i X_i + \sqrt{n/4}}{2\sqrt{n/4} + n}\right] &= \frac{1}{\left(2\sqrt{n/4} + n\right)^2}\sum_{i=1}^{n} \text{Var}\left[X_i\right] \\ &= \frac{np(1-p)}{\left(\sqrt{n} + n\right)^2}\end{aligned}$$

As for the bias term, which just really mean we need the expected value

$$\mathbb{E}\left[\frac{\sum_i X_i + \sqrt{n/4}}{2\sqrt{n/4}+n}\right] = \frac{np+\sqrt{n/4}}{2\sqrt{n/4}+n}$$

Taking the difference with p and squaring gives

$$= \left(\frac{p\sqrt{n} + \cancel{np - np} - \sqrt{n/4}}{\sqrt{n}+n}\right)^2 = \left(\frac{p\sqrt{n} - \sqrt{n/4}}{\sqrt{n}+n}\right)^2$$

If we add this with the variance

$$\text{MSE} = \frac{np(1-p) + \left(p\sqrt{n} - \sqrt{n/4}\right)^2}{\left(\sqrt{n}+n\right)^2}$$

Taking the derivative of this and setting it to 0

$$n \cancel{- 2np + 2np} - n/2 = 0$$

reveals that the MSE is monotonically increasing! Since p is defined only on [0,1], the maximum occurs at $p = 1$. Thus the maximum risk is

$$R_{\max}(\hat{p_1}) = \frac{n}{4\left(\sqrt{n}+n\right)^2}$$

This function is always smaller than the maximum risk of $\hat{p}_1$ for all values of n. Thus, under maximum risk, $\hat{p}_2$ is a better estimator for p than $\hat{p}_1$. Now looking at the Bayes risk with uniform prior on p, we find that

$$R_{\text{Bayes}}(1, \hat{p}_1) = \int_0^1 \frac{p(1-p)}{n} dp = \frac{1}{6n}$$

$$R_{\text{Bayes}}(1, \hat{p}_2) = \int_0^1 \frac{np(1-p) + n\left(p + \sqrt{1/4}\right)^2}{\left(\sqrt{n}+n\right)^2} dp = \frac{n}{4\left(\sqrt{n}+n\right)^2}$$

Now we no longer see uniform dominance! For small values of n, the Bayes risk says that we should choose $\hat{p}_2$ but when n becomes sufficiently large, $\hat{p}_1$ is the better estimator. Thus, the choice of prior matters.

2.2 Sources Of Error

Every model that we are going to develop must have error encoded in it; otherwise, we could just learn the model exactly (and we wouldn't be doing statistics). Oftentimes, this is a consequence of not knowing what the data actually looks like, especially in higher dimensions. Our ignorance will often lead us to creating models that are high **bias** or high **variance**. And even if we could theoretically know what the data looks like, not all errors are systematic. Some are forced upon us by nature, and such error is called noise.

2.2.1 Irreducible Error

The first type of error that we are going to examine is noise. Noise is a random variable added into a system that increases the uncertainty of our model. For example, for the same input X_i, we might get out multiple different Y_i outputs; and these outputs might change for different instances of observing Y_i. In this sense, we call noise **irreducible** because no matter how we tweak our model, we cannot get rid of it. One simple example of a model with noise is the linear regression model

$$Y_i = X_i^T \theta + \epsilon \tag{2.6}$$

where ϵ is typically assumed to be a centered normal random variable. As we can see, if we make independent observations of X_i, we might get different Y_i. However, because the noise is centered, the expectation of our model output should always just be $X_i^T \theta$.

2.2.2 The Bias Variance Trade Off

Not all noise is systematic, but some are. When errors occur in our modeling that are not due to noise, it usually either introduces variability or bias. In the machine learning notes, we examined the bias-variance tradeoff in the case of the mean squared error loss. In general, we say that the risk function defined as

$$R\left(f, \hat{f}\right) = \mathbb{E}\left[L\left(f, \hat{f}\right)\right] \tag{2.7}$$

where L is some loss function, f is the true density function we are trying to estimate, and $\hat{f}$ is the estimator for the function f. Note that $\hat{f}$ is a random function evaluated at some point x_0. The risk function can always be decomposed as

$$R\left(f, \hat{f}\right) = \int b^2(x)\, dx + \int v(x)\, dx \tag{2.8}$$

with

$$\underbrace{b(x) = \mathbb{E}\left[\hat{f}(x)\right] - f(x)}_{\text{bias}} \qquad \underbrace{v(x) = \mathbb{E}\left[\left(\hat{f}(x) - \mathbb{E}\left[\hat{f}(x)\right]\right)^2\right]}_{\text{variance}} \tag{2.9}$$

When the loss function is the mean squared error, we proved this to be true back in machine learning. The reason that we always use the mean squared error example is because it is the simplest loss function that demonstrates this. Most other loss functions are very difficult to decompose. On top of that, the mean squared error loss is a household standard when doing machine learning, so in some sense, it is the only one that's really worth mentioning.

However, we have to be very clear about what we mean by the bias and the variance in machine learning. The reason for this is because these terms are no longer point statistics; they are functions! To see what we mean by this, suppose we use the standard squared error loss function

$$L(x) = \left(\hat{f}(x) - y(x)\right)^2 \tag{2.10}$$

where $\hat{f}(x)$ is our model and y is our observation data. This is the part where we have to be a bit careful; in statistics, we assume that there exists some true parameter θ that we can estimate from the data. When doing machine learning, we don't explicitly assume that the true model is some function $f(x)$, because more often than not, we just don't know. All that we have is just the observation $y(x)$. But this is more just philosophy because mathematically we can always write the loss function as described above to being a loss function of some true function we'll call it $f(x)$

$$\begin{aligned} L(x) &= \left(\hat{f}(x) - y(x)\right)^2 \\ &= \left(\left(\hat{f}(x) - f(x)\right) + (f(x) - y(x))\right) \quad \text{introduce 0 term } \pm f(x) \\ &= \left(\hat{f}(x) - f(x)\right)^2 + 2(\hat{f}(x) - f(x))\,(f(x) - y(x)) + (f(x) - y(x))^2 \quad \text{expand} \end{aligned}$$

There are 2 stochastic elements in the above. The first is obviously our model $\hat{f}(x)$ and the second is the observation $y(x)$. Our model will obviously depend on the observation $y(x)$, so naturally if we are to compute the risk function, we would do so with respect to the random variable $y(x)$

$$\begin{aligned} \mathbb{E}_y\left[L(x)\right] &= \mathbb{E}_y\left[\left(\hat{f}(x) - f(x)\right)^2 + 2(\hat{f}(x) - f(x))\,(f(x) - y(x)) + (f(x) - y(x))^2\right] \\ &= \left(\hat{f}(x) - f(x)\right)^2 + 2(\hat{f}(x) - f(x))\mathbb{E}_y\left[\cancelto{0}{(f(x) - y(x))}\right] + \underbrace{\mathbb{E}_y\left[(f(x) - y(x))^2\right]}_{\sigma^2} \end{aligned}$$

$$= \left(\hat{f}(x,\theta) - f(x)\right)^2 + \sigma^2 \tag{2.11}$$

where σ^2 by definition is the irreducible noise term (or just the variance of the true function to the observation). So when machine learning, the mean squared error risk can be decomposed as the squared error of our model to the true function $f(x)$ plus an irreducible error σ^2. Because we still have $\hat{f}(x)$ as a random variable, we can continue to do bias variance decomposition. But with respect to what variable? We know that our model $\hat{f}(x)$ has stochasticity from the θ parameters, and the θ parameters are determined from some dataset $\mathscr{D}$. Let us define

$$\mu(x) = \mathbb{E}_{\mathscr{D}}\left[\hat{f}(x,\theta[\mathscr{D}])\right] \tag{2.12}$$

as the expectation of our model with respect to the space of datasets $\mathscr{D}$. If we apply the exact same procedure as we did to get (2.11), we would have

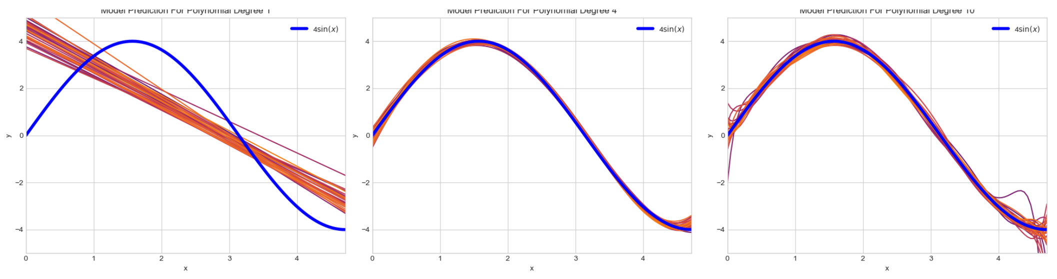
$$\mathbb{E}_{\mathscr{D}}\left[\left(\hat{f}(x,\theta[\mathscr{D}]) - f(x)\right)^2 + \sigma^2\right] = \underbrace{\mathbb{E}_{\mathscr{D}}\left[\left(\hat{f}(x,\phi[\mathscr{D}]) - \mu(x)\right)^2\right]}_{\text{variance}} + \underbrace{(\mu(x) - f(x))^2}_{\text{bias}} + \underbrace{\sigma^2}_{\text{noise}} \tag{2.13}$$

where, to reiterate, the expectation is done over the space of datasets. To shine a bit more light on what we mean, suppose we generate some dataset and we fit the data with some model. When we say we are taking an expectation with respect to a dataset (equation (2.12)), what we really are doing is we are finding the average outcome *for all possible datasets*. Obviously in practice, we cannot do this. Computationally, however, we can use the law of large numbers to retrieve an estimate for what the function (2.12) should be. Again, I stress that the estimate is NOT a single point estimate but rather an entire function. Thus, when we talk about the variance of our model, what we're really saying is if we obtained a fresh batch of data $\mathscr{D}$, how much does our model deviate from the previous? Our model output is a function, so it makes sense that the average model output should be a function.

Figures 2.2 and 2.3 visualize these results from generated data. The code for creating these plots is available in the chapter 2 jupyter notebook file.

2.2.3 The Double Descent Surprise

In the interesting is that in practice, this isn't always the case. For the following analysis, we are going to be working with the MNIST-1D dataset, which is a collection of handwritten digits 0-9 that have been compressed into 1 dimension. It is a tradition to use this dataset because neural networks (when doing classification) have performed very well with this dataset. We will also re-use the deep neural network created in chapter 1 to investigate this



(a) k=1 (b) k=4 (c) k=10

Figure 2.2: We perform polynomial regression on datasets generated by an underlying function $f(x) = 4\sin(x)$ plus white noise. In plots (a-c), we have the degree of the fitted polynomial as k=1,k=4,and k=10 respectively. We see that as we increase the model complexity (the degree of the polynomial), the bias starts off very high in figure 1 and then drops as we increase k. The variance however, fluctuates wildly at the tails when k becomes large.

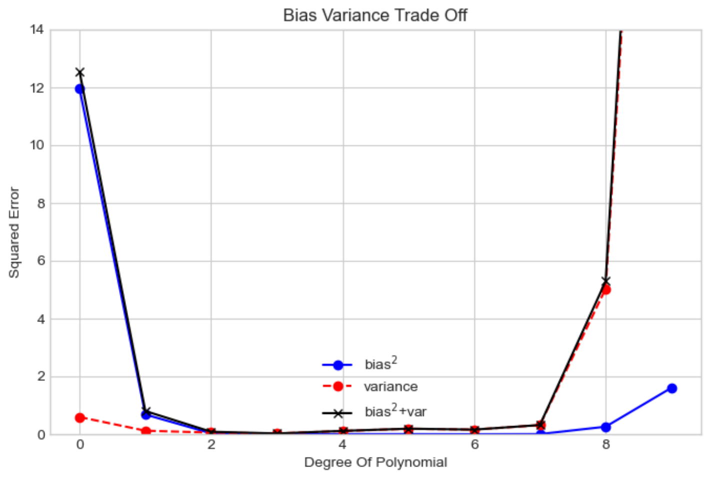


Figure 2.3: A computational plot of the bias variance tradeoff. On the x-axis, we have the degree of the fitted polynomial while on the y axis, we have the squared error. In blue is the squared bias, in red is the variance, and the black line is the sum of the 2. As we expect, when we have low complexity, the bias between model and true function is high but the variance is low. The opposite happens when we have high complexity. There exists some sweet spot (k=4) when the total error is a minimum, and this corresponds well with figure 2.2

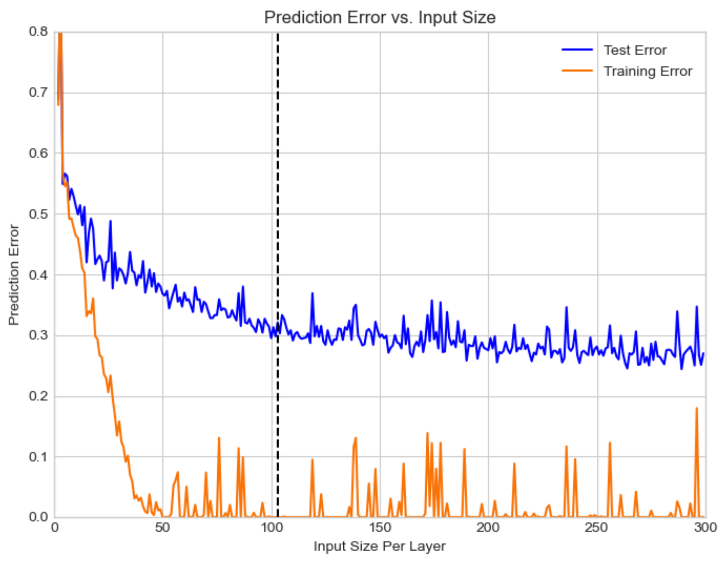


Figure 2.4: A plot of the test error and training error both as functions of the number of nodes k in each of the 2 layers of our neural network. The black dotted line is the critical point when the number of parameters equal the number of data points. Notice that the model has begun overfitting even before the line line but the testing error does not skyrocket. It gradually continues to decreases, contrary to what we expect.

problem. We will load the model with 2 hidden layers and k nodes per layer. We expect from figure 2.3 and equation (2.13) that as we increase the number of nodes in the 2 layers that eventually we will hit a critical point where the number of parameters in our model is equal to the number of training data. At this point, the model has begun overfitting and we should expect that the testing accuracy will plummet. This does not happen. In fact, the model's testing accuracy continues to increase! In fact, if we introduce label noise (that is, we incorrectly label a random subset of the prediction y), we see that the testing error does increase up until the critical point and then the error decreases again. This phenomenon is known as the **double descent phenomenon**. The name double descent refers to the fact that the error decays twice. But *why* does this happen?

As it turns out, we don't know. Well, that's not entirely true. We understand *what* is causing it to generalize, but we don't know exactly *why* it happens. To try to begin understanding what's going on under the hood, we have to first understand the scale of what is going on. The MNIST-1D data set is a collection of 40 dimensional vectors of numbers, and there are 10 possible classes that we can put them in. That is a whopping 10^{40} different combinations

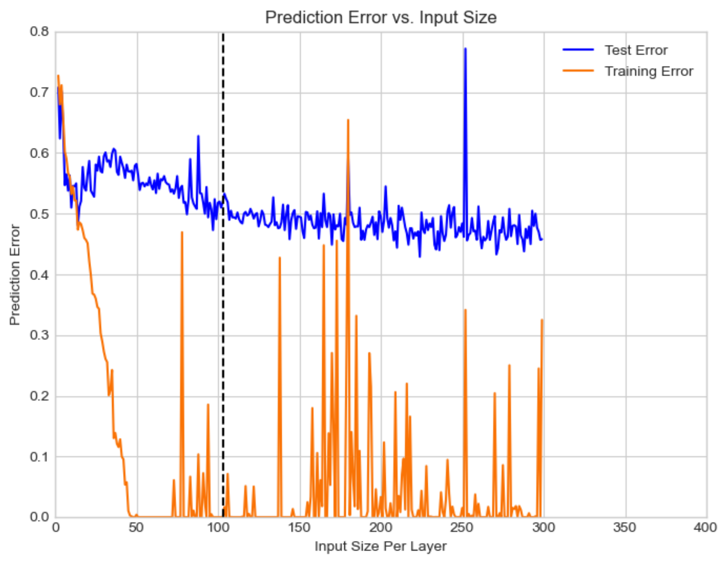


Figure 2.5: If we add training noise, we see the double descent curve more prominently. As the model becomes more saturated, the testing error increases up until it hits some point and then the testing accuracy decreases.

of elements! The original dataset has 16,000 examples, so we would have 10^{40} possible elements binned by $16,00 \times 10 = 1.6 \times 10^5$ data points. Effectively, this means that there would be on average approximately 1 data point for every 10^{35} bins! Most of the space would be empty! Machine learning people call this the **curse of dimensionality** but I feel like that is a bit of a misnomer in this context.

Problem 3.

The volume of a hypersphere with radius r in d dimensions is

$$\mathrm{V}(r) = \frac{r^d \pi^{d/2}}{\Gamma(d/2+1)}$$

where $\Gamma(x)$ is the Gamma function. Show using Stirling's approximation that the volume of a hypersphere of diameter one (radius $r = 0.5$) becomes zero as the dimension increases.

Solution:

This one is actually pretty easy. Stirling's approximation for the factorial function is

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

The reason we need this is because the gamma function $\Gamma(x)$ is a generalization of the factorial function. More specifically, $\Gamma(n+1) = n!$. Thus, the volume of a n dimensional hypersphere is

$$\mathrm{V}_n(r) \approx \frac{r^n \pi^{n/2}}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}$$

We don't really care about the $\sqrt{n}$ in the denominator because the square root function will be dominated by the exponential function. If we simplify this a bit, we get

$$\mathrm{V}_n(r) = \frac{1}{\sqrt{2\pi n}}\left(\frac{e\sqrt{\pi}}{n}r\right)^n$$

Taking the limit

$$\boxed{\lim_{n\to\infty} \frac{1}{\sqrt{2\pi n}}\left(\frac{e\sqrt{\pi}}{n}r\right)^n = 0}$$

The limit evaluates to 0. The justification is that for $r = 1/2$, there will exist a value for n such that n is larger than the numerator. When that happens, the whole term in parentheses becomes smaller than 1, and when exponentiated, will continue to be smaller than 1. Thus the limit evaluates to 0, showing that for a fixed radius, the dimension of the hypersphere vanishes as the dimensionality increases!

The revelation from problem 3 is that the volume that a hyper-sphere of fixed radius occupies relative to the unit box is vanishing as the number of dimensions increases. What this means is that most of high dimension is just empty space! What that means for us is that most of the data looks like empty space, so as we continue to increase the number of parameters in our model, the interpolation between points will become increasingly smooth. And this makes sense! When the number of parameters equals the training data, the model has to contort itself to fit every single point (because there are as many parameters as there are points). Imagine you and your friend are across a canyon and you want to throw a (rigid) rope across to your friend. If the rope is rigid and there are few degrees of freedom in which it can bend and flex, in order for it to connect you and your friend, it will probably zigzag sporadically. However, if you add more and more degrees of freedom along the rope, it becomes smoother and smoother.

While this could be a plausible description, it still doesn't answer *why* this happens. It could be that the model prefers smoothness when interpolating. But what causes this smoothness? Why do high dimensional spaces encourage better generalization? The unfortunate truth is that we just don't know.

Problem 4.

Let's examine another really cool property of high dimensional geometry. Consider a hyper-sphere of radius $r = 1$. Find an expression for the proportion of the total volume that lies in the outermost 1% of the distance from the center (i.e., in the outermost shell of thickness 0.01). Show that this becomes one as the dimension increases.

Solution:
We know from problem 3 that the volume of the hyper-sphere is

$$\mathrm{V}(r) = \frac{r^d \pi^{d/2}}{\Gamma(d/2+1)}$$

and using Stirling's approximation, we'd have

$$\mathrm{V}_n(r) = \frac{1}{\sqrt{2\pi n}} \left(\frac{e\sqrt{\pi}}{n} r \right)^n$$

To find the relative volume (we say relative because the radius is set to 1, so whatever volume we find is actually a percentage), we'd have to take a ratio (noticing all the constants cancel out)

$$\frac{V_{.01,n}}{V_{1,n}} = \frac{r^n - (.99r)^n}{r^n}$$

Why is this the case? The ratio $.99r/r$ is the proportion of the radius from the center up to 99 percent of the radius. But that's not what we're looking for; what we're looking for is the remainder (if you're not convinced, you can just integrate a uniform density with respect to r from $.99r$ to r and you'll get the same thing). Now we have to take this limit

$$\boxed{\lim_{n\to\infty} \frac{V_{.01,n}}{V_{1,n}} = \lim_{n\to\infty} 1 - \left(\frac{.99\cancel{r}}{\cancel{r}} \right)^n = 1}$$

What this tells us is that most of the space in a hyper-sphere is on the edge! You can think of it as almost like a bobblehead, with the feet at the origin.

Figure 2.6: An example of an image that is distorted. In real life, cats do not have rainbow colored fur (certainly hope not), and yet, we are able to recognize that this is an image of a cat. Introducing data that is distorted or sometimes just wrong can help the model generalize better.

2.3 Error Reduction

In the last section, we talked about the different sources of error that we could encounter from our model, as well as the pleasant surprise from the double descent phenomenon. We've also introduced the idea of generalization: the ability of the model to extrapolate to unforeseen data. We are going to talk about generalization a bit in this section, more specifically about the different methods we can use to help the model generalize better.

2.3.1 Data Augmentation

Data augmentation more often than not just means adding noise into the training labels to the model to make it be less confident. Surprisingly, this actually reduces the variance of the model by sometimes quite a bit. It is also what causes the pronounced second descent in double descent. This has the most noticeable impact when doing image recognition problems (we will explore this in more detail in the next chapter). To understand why this happens, imagine you have a dataset of handwritten numbers like the MNIST data set. Different people will obviously write numbers differently, and having more examples would be good. However, sometimes we just can't obtain more examples. What we can do is we can distort the images. For example, we could stretch the images, flip some pixels, etc etc. This will help train our model to better generalize on what it means for a number to be the number

6. Humans are exceptionally good at recognizing patterns, and in some sense, you could teach a human to generalize better by giving them distorted training data. A non-trivial analogy of this principle would be through language. When we learn a language for the first time (as adults), we typically adhere to pretty strict grammatical structures and sentence construction. This makes it pretty difficult to generalize to new thoughts and sound natural because native speakers will actually sometimes use improper grammar. If you train yourself with examples of usage of improper grammar and wacky sentences, you can generalize a lot better and sound a lot more natural. That is essentially what we are doing.

2.3.2 Implicit and Explicit Regularization

In the machine learning notes, we talked about regularization as a Lagrange multiplier term to the loss function, which will introduce bias in the model

$$\tilde{L}(\theta) = L(\theta) + \lambda \underbrace{\text{norm}(\theta)}_{\text{regularization}}$$

and different norms gave different results. For example, in linear regression, if we introduced an L_2 regularization term on the θ parameters, we got Ridge regression. This was nice for linear regression because it stabilizes our model by lifting degeneracies of the covariance matrix. Remember that the Ridge coefficients were

$$\hat{\theta}_{\text{Ridge}} = \left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y} \tag{2.14}$$

where the addition of $\lambda \boldsymbol{I}$ to the covariance matrix forces the term in the parentheses to always be invertible. This will stabilize the model by biasing some solutions of θ over others; particularly ones that don't have θ blowing up. Understanding this, if we were to add regularization to our networks, it would push θ towards solutions that don't fit the training data exactly and therefore trading off overfitting (variance) for a little bit of bias so that the model generalizes better. But this is old news! We've known this for a long time now. What's more interesting is the implicit regularization that we do.

When doing gradient descent, we are actually biasing solutions. To understand why this is the case, consider the smoothed (when $\alpha \to 0$) trajectory of gradient descent; this trajectory would be governed by the differential equation

$$\frac{\partial \theta}{\partial t} = -\frac{\partial L(\theta)}{\partial \theta} \tag{2.15}$$

Gradient descent, then, would be equivalent to using Euler's method to find what the solution would be given some initial condition. The fact that the step sizes are finite means that the trajectory of the particle under gradient descent will vary (sometimes wildly) off course from what the actual path would be. This imposes an **implicit bias** towards some solutions over others, and hence the name **implicit regularization**. To see how the gradient descent deviates from the continuous trajectory, start with the gradient of θ

$$\frac{\partial\theta}{\partial t} = g(\theta)$$

the above is just a statement that the evolution of θ just depends on some function of θ. This makes sense because the trajectory depends on the gradient of the loss function which is a function of θ as seen in (2.15). Now we use a technique from differential equations called **backwards error analysis**, which starts by saying that when we use a method like Euler's method (discrete gradient descent in our case), then the error in the trajectory can be treated as perturbations with respect to the learning rate α

$$\frac{\partial\theta}{\partial t} \approx g(\theta) + \alpha g_1(\theta) + \mathcal{O}(\alpha^2) \tag{2.16}$$

where $g_1(\theta)$ is some deviation from the continuous gradient descent trajectory controlled by a nonzero step size α. We've ignored higher ordered terms in α. Obviously, in the continuous case, α is 0 so we would just get back (2.15). Solving these types of solutions is usually impossible, but we can use an asymptotic trick which is to expand the parameter function around its starting point

$$\theta(t+\alpha) \approx \theta(t) + \alpha\frac{\partial\theta}{\partial t} + \frac{\alpha^2}{2}\frac{\partial^2\theta}{\partial t^2} + \mathcal{O}(\alpha^3) \tag{2.17}$$

Subbing (2.16) into (2.17) and applying the chain rule gives

$$\begin{aligned}
\theta(t+\alpha) &\approx \theta(t) + \alpha\left(g(\theta) + \alpha g_1(\theta)\right) + \frac{\alpha^2}{2}\left(\frac{\partial g}{\partial\theta}\frac{\partial\theta}{\partial t} + \alpha\frac{\partial g_1}{\partial\theta}\frac{\partial\theta}{\partial t}\right) \\
&= \theta(t) + \alpha\left(g(\theta) + \alpha g_1(\theta)\right) + \frac{\alpha^2}{2}\left(\frac{\partial g}{\partial\theta}g(\theta) + \alpha\frac{\partial g_1}{\partial\theta}g(\theta)\right) \\
&\approx \theta(t) + \alpha g(\theta) + \alpha^2\left(g_1(\theta) + \frac{1}{2}\frac{\partial g}{\partial\theta}g(\theta)\right) + \mathcal{O}(\alpha^3) \qquad (2.18)
\end{aligned}$$

Where we have ignored all terms with α^3 or higher. Now, in order for this perturbation to make contact with the unperturbed trajectory (that is, in order for discrete gradient descent and continuous GD to arrive at the same solution), we must have the correction term be

$$g_1(\theta) = -\frac{1}{2}\frac{\partial g}{\partial \theta} g(\theta) \tag{2.19}$$

as to cancel the rightmost terms of (2.18). With this condition, we can now revisit (2.16) and plug in (2.19)

$$\frac{\partial \theta}{\partial t} \approx g(\theta) - \frac{\alpha}{2}\frac{\partial g}{\partial \theta} g(\theta) \tag{2.20}$$

Remembering that $g(\theta)$ is nothing more than just the gradient of the loss function

$$\frac{\partial \theta}{\partial t} \approx -\frac{\partial L(\theta)}{\partial \theta} - \frac{\alpha}{2}\frac{\partial^2 L(\theta)}{\partial \theta^2}\frac{\partial L(\theta)}{\partial \theta} \tag{2.21}$$

Note that this gradient trajectory is a correction of the continuous trajectory to end at the same location as the discrete trajectory. Undoing the derivative (while remembering that we are doing gradient *descent* so we must undo the negative sign as well) gets us the effective loss function as a function of the continuous trajectory's loss

$$L_{GD}(\theta) = L(\theta) + \frac{\alpha}{4}\left|\left|\frac{L(\theta)}{\partial \theta}\right|\right|^2 \tag{2.22}$$

So what does this tell us? The deviation of gradient descent from the "true" trajectory is governed by the learning rate α as well as the gradient of the loss at that point. We see natural repulsion from the usual trajectory just from the fact that α is nonvanishing, but we also see that gradient descent is repelled away from trajectories with very steep bends (because we are trying to minimize the loss, going in directions of very large gradients is not the direction that will minimal loss as seen in (2.22)). This repulsion could possibly take us on trajectories that converge to different minimums.

In fact, if we were to perform a similar analysis on stochastic gradient descent and its trajectory, we would see

$$\tilde{\mathrm{L}}_{SGD}(\theta) = \mathrm{L}(\theta) + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \theta}\right\|^2 + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \theta} - \frac{\partial L}{\partial \theta}\right\|^2 \tag{2.23}$$

with

$$L = \frac{1}{I}\sum_{i=1}^{I}\ell_i\left(\mathbf{x}_i, y_i\right) \quad \text{and} \quad L_b = \frac{1}{|\mathscr{B}|}\sum_{i\in\mathscr{B}_b}\ell_i\left(\mathbf{x}_i, y_i\right) \tag{2.24}$$

where L_b is the loss for the b-th element of the B batches and I is the number of individual losses in the full batch and $|\mathscr{B}|$ is the size of the mini-batch. Notice that the effective loss for

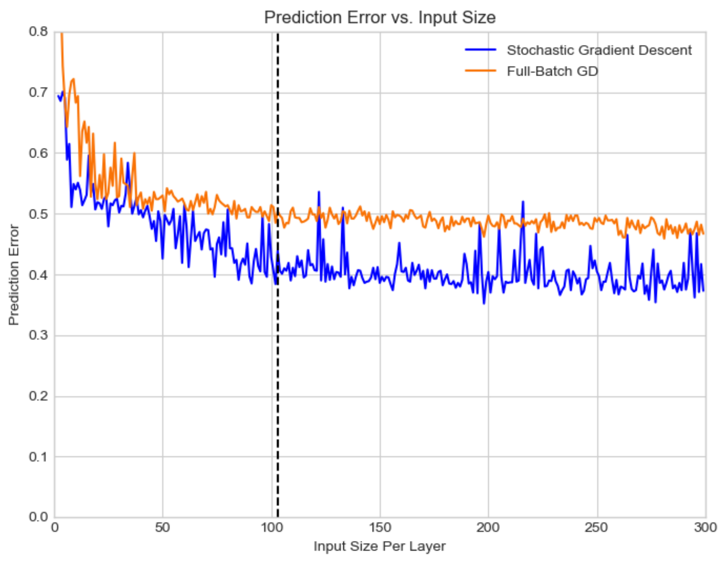


Figure 2.7: Stochastic gradient descent vs. Full batch gradient descent's testing error as a function of input layer size. The network is trained on 2 hidden layers with a learning rate of lr=.1. As we can see, for almost every layer size, SGD outperforms GD's ability to generalize.

stochastic gradient descent has the same regularization term as full batch gradient descent *as well as* a secondary regularization term. The existence of this second penalty implies that stochastic gradient descent does a better job generalizing on new data than full batch gradient descent.

Problem 5.

Show that the weight decay parameter update with decay rate λ :

$$\theta \longleftarrow (1-\lambda)\theta - \alpha\frac{\partial L}{\partial \theta}$$

on the original loss function $\mathrm{L}(\theta)$ is equivalent to a standard gradient update using L2 regularization so that the modified loss function $\tilde{\mathrm{L}}(\theta)$ is:

$$\tilde{\mathrm{L}}(\theta) = \mathrm{L}(\theta) + \frac{\lambda}{2\alpha}\sum_k \theta_k^2$$

where θ are the parameters, and α is the learning rate.

Solution:
For this problem, we don't have to do any repetition of the previous analysis. To see this, first start with the discretized gradient descent step

$$\theta_{t+\alpha} = (1-\lambda)\theta_t - \alpha\frac{\partial L}{\partial \theta}$$

If we distribute the θ_t term and divide out by α, we can rewrite the gradient step as

$$\frac{\theta_{t+\alpha} - \theta_t}{\alpha} = -\frac{\lambda}{\alpha}\theta_t - \frac{\partial L}{\partial \theta}$$

Sending the limit that $\alpha \to 0$ gives the continuous diffeq

$$\frac{\partial \theta}{\partial t} = -\frac{\lambda}{\alpha}\theta - \frac{\partial L}{\partial \theta} \equiv g(\theta)$$

where we have defined $g(\theta)$ to just be the middle term of above. Now, the point of backward error analysis is to ask: at each step, how much does the modified gradient deviate from the true gradient? Remember that the true gradient is

$$\frac{\partial \theta}{\partial t} = -\frac{\partial L}{\partial \theta}$$

However, our modified gradient has an extra term $-\lambda\theta/\alpha$, so in fact *that* is the deviation! If we integrate both sides with respect to θ (don't forget the minus sign because we are doing gradient *descent*!), we see that the effective loss is

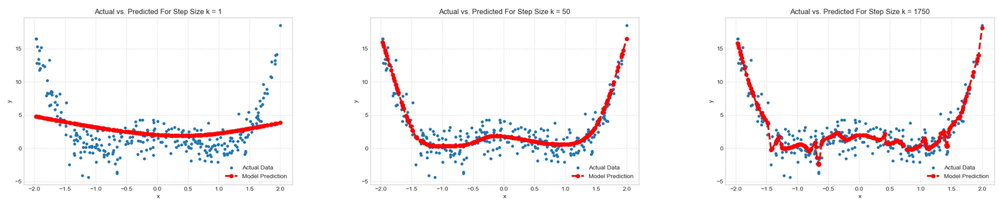
$$\boxed{\tilde{L}(\theta) = L(\theta) + \frac{\lambda}{2\alpha}\|\theta\|^2}$$

which is what we had aimed to solve.

2.3.3 Early Stopping, Ensembling, Dropout

The previous sections contextualized what we mean by a model's ability to generalize, specifically through the process of regularization. In this section, we are going to provide some heuristics to improve model performance and how it connects with regularization.

The first of these methods is called **early stopping** and it is exactly what it sounds like: we stop the model before it can fully converge. What this does is that once we've



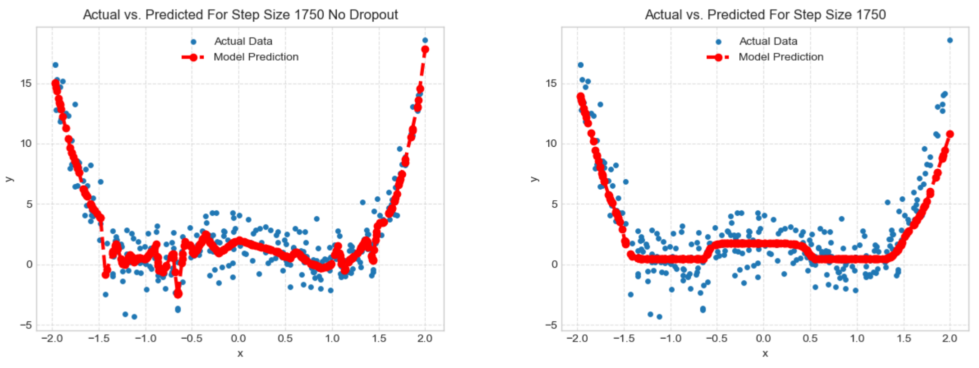
(a) k=1 (b) k=50 (c) k=1750

Figure 2.8: We perform polynomial regression on datasets generated by an underlying function $f(x) = 2\left(x^2 - 1\right)^2$ plus white noise. In plots (a-c), we have the degree of the number of steps the model took before finishing as k=1,k=50,and k=1750 respectively. We initialize the neural network to have 2 hidden layers, 200 nodes each. We train on the Adam optimizer. We see that after 1 step, the model exhibits high bias. After 50 steps, it does a pretty good job. Then when we allow it to run for 1750 steps, we see clear signs of overfitting.

found a good enough approximation of the true function, we stop the model before it has time to overfit. To understand why this happens, recall back to the previous chapter when we talked about weight initialization. We saw that for some initialization of the weights, the variance of the gradients can blow up. Early stopping prevents the weights of the models from reaching the point where the weights and its gradients explode. This is effectively the same thing as L_2 regularization (shrinkage of the coefficients). If we understand this, we next need to understand how one would implement this. When do we choose a good stop? For what value of the loss is it "small enough"? Unfortunately (or fortunately, depending on how you view it), the number of steps until we stop is a hyperparameter that we have to choose empirically. Examine figure 2.8 for the results.

The second method that we are going to examine is called **ensembling**. An ensemble is a collection of (potentially) different systems. What we will do is create many different models to train on the same dataset, and then average their predictions. Averaging predictions could be implemented in many different ways depending on the type of problem that we are trying to solve (such as regression or classification). For example, in classification problems, we can take the mean of the pre-softmax outputs of all the models. We can also change the model in a number of ways, including different parameter initialization or re-sampling the same dataset with different models (called **bootstrap aggregation** or **bagging**). The obvious downside to this method is that it is clearly very expensive.

The last method that we are going to examine is called **drop-out**, which involves randomly setting some of the weights in our model to be 0 at each step of gradient descent. This



(a) No dropout (b) With dropout, p = .5

Figure 2.9: Implementation of 2 layer network, 200 nodes with drop out using p value equaling .5 for number of steps totaling 1750. Notice that without drop out, we see overfitting as predicted in figure 2.8. Now, if we add drop out, the lines flatten out and our model can reasonably predict the function even when the number of parameters overwhelm the number of data points

will make very sensitive weights (those that will blow up) contribute less on average, thus smoothing out the function that we are trying to estimate and also reducing the variance of our model since exploding weights will contribute less to the loss. This is also effectively an L_2 regularization scheme.

3

Convolutional Neural Networks

"My life is a convoluted web of lies"

– Meg Cabot

The result that arguably put deep learning on the map of "serious science" was probably its ability to do image recognition. Even today, deep neural networks (specifically convolutional networks) are developed and used in fields like computer vision and natural language processing. Although they have wide ranging applications (we will use them later on in the book again and again), for this chapter, we will just focus on applying the theory to image data, as that it will make the most intuitive sense. But why image data? What's so special about image data that is different from other data (like the MNIST1D data set)? Fundamentally, an image is nothing more than a 2 dimensional grid of 224 possible RBG values. This means that the input dimension alone is $224^2 = 150,528$. IF we had just a single layer, the number of weights becomes $150,528^2$ which is a number that I am too lazy to type out because of its length. So as we can see, even for a simple image, we run into practical issues of scaling.

However, there is a nice property of images that reduces the complexity of the problem: pixel correlation. Imagine an image of a snowy mountain; if you choose any random pixel, you might imagine that its color would be close to white. Now look at the neighboring pixels. They are also likely to be white. Not only that, the image is preserved under certain transformations of the system like rotations. A picture of a cat rotated 180 degrees still describes a cat. The fact that the information stored within the pixels are deeply correlated to each other allows us to make the solution a bit more tractable by introducing a

convolution. In this chapter, we will begin talking about the convolution operator, image symmetry and how the convolution preserves such symmetries, and then finish up with the architecture of convolutional neural networks. This chapter will mark the beginning of a shift from the previous chapters, as the conversation becomes more scientific and less computational.

3.1 Image Transformations

3.2 The Convolution Operator

3.2.1 The Continuous Convolution

3.2.2 The Discrete Convolution In 1D

3.2.3 The Discrete Convolution In 2D

3.3 Convolutional Network Architecture

3.3.1 Convolutional Layers

3.3.2 Stride, Size, Dilation, and Padding

3.3.3 Downsampling and Upsampling

3.4 Image Classification and Computer Vision

4

Transformers

5

Graph Neural Networks

6

Generative Adversarial Networks

7 Normalizing Flow

8

Variational Autoencoders