1.1.1

ChatGpt trace

Let's first define some notations for clarity:

- X: a d x n matrix, where d > n, and the columns of X are linearly independent.
- t: a n x 1 vector
- w_*: the unique minimum norm solution to the equation $Xw_* = t$, obtained by gradient descent starting from zero initialization.
- w0: initial weights of the linear model, set to 0.
- $\hat{w}$: a solution obtained by mini-batch SGD such that $X\hat{w} = t$.

We need to show that $w_* = \hat{w}$, i.e., the mini-batch SGD solution is identical to the minimum norm solution obtained by gradient descent.

Since $Xw_* = t$, we can write w_* as:

$w_* = \text{argmin}(||Xw - t||^2)$

where ||.|| is the L2-norm.

Now, let's consider the update rule for mini-batch SGD:

$w(k+1) = w(k) - \eta * (\nabla(||Xb(k)w - tb(k)||^2))$

where w(k) is the weight vector at the k-th iteration, b(k) is a randomly selected mini-batch of size m from X and t, and η is the learning rate.

Note that Xb(k) is a submatrix of X obtained by selecting the columns of X corresponding to the indices in b(k), and tb(k) is the corresponding subvector of t.

Let's assume that after K iterations, mini-batch SGD converges to a solution $\hat{w}$ such that $X\hat{w} = t$. Since the columns of X are linearly independent, X is full-rank, and hence invertible. Therefore, we can write $\hat{w}$ as:

$\hat{w} = X^{-1}t$

Now, let's prove that $w_* = \hat{w}$.

Since w_* minimizes the L2-norm $||Xw - t||^2$, we have:

$\nabla(||Xw_* - t||^2) = 2X^T(Xw_* - t)$

Setting this gradient to zero, we get:

$X^TXw_* = X^Tt$

Multiplying both sides by X^{-1}, we get:

$w_* = X^{-1}t$

which is the same as the expression for $\hat{w}$. Therefore, we have:

Let's assume that after K iterations, mini-batch SGD converges to a solution $\hat{w}$ such that $X\hat{w} = t$. Since the columns of X are linearly independent, X is full-rank, and hence invertible. Therefore, we can write $\hat{w}$ as:

$\hat{w} = X^{-1}t$

Now, let's prove that $w_* = \hat{w}$.

Since w_* minimizes the L2-norm $||Xw - t||^2$, we have:

$\nabla(||Xw_* - t||^2) = 2X^T(Xw_* - t)$

Setting this gradient to zero, we get:

$X^TXw_* = X^Tt$

Multiplying both sides by X^{-1}, we get:

$w_* = X^{-1}t$

which is the same as the expression for $\hat{w}$. Therefore, we have:

$w_* = \hat{w}$

This proves that the mini-batch SGD solution is identical to the minimum norm solution obtained by gradient descent. Regenerate response

Critique of the model output: It is correct

Solution:

Since $Xw* = t$, we can write $w*$ as:

$w* = \text{argmin}(||Xw - t||^2)$

where $||.||$ is the L2-norm.

Now, let's consider the update rule for mini-batch SGD:

$w(k+1) = w(k) - \eta * (\nabla(||Xb(k)w - tb(k)||^2))$

Let's assume that after K iterations, mini-batch SGD converges to a solution $w\hat{}$ such that $Xw\hat{} = t$. Since the columns of X are linearly independent, X is full-rank, and hence invertible. Therefore, we can write $w\hat{}$ as:

$w\hat{} = X^{-1}t$

Now, let's prove that $w* = w\hat{}$.

Since $w*$ minimizes the L2-norm $||Xw - t||^2$, we have:

$\nabla(||Xw* - t||^2) = 2X^T(Xw* - t)$

Setting this gradient to zero, we get:

$X^TXw* = X^Tt$

Multiplying both sides by X^{-1}, we get:

$w* = X^{-1}t$

which is the same as the expression for $w\hat{}$. Therefore, we have:

$w* = w\hat{}$

This proves that the mini-batch SGD solution is identical to the minimum norm solution obtained by gradient descent.

1.2.1

Chat GPT trace:

Let's consider the overparameterized linear model with d > n and the loss function defined in Section 1:

$L(w) = ||Xw - t||^2$

where X is a d x n matrix, w is a n x 1 weight vector, and t is a d x 1 target vector.

We are given that the RMSProp optimizer converges to a solution. We need to determine whether RMSProp always obtains the minimum norm solution.

Let's compute a simple 2D case with x1 = [2, 1], w0 = [0, 0], and t = [2]. In this case, X is a 1 2 matrix:

X = [2 1]

The loss function can be written as:

$L(w) = (2w1 + w2 - 2)^2$

where w1 and w2 are the components of the weight vector w.

The gradient of L(w) is:

$\nabla L(w) = [4(2w1 + w2 - 2), 2(2w1 + w2 - 2)]$

The update rule for RMSProp is:

$g(k) = \beta g(k-1) + (1 - \beta)(\nabla L(w(k)) \odot \nabla L(w(k)))$

$w(k+1) = w(k) - \eta(\nabla L(w(k)) / \sqrt{}(g(k) + \varepsilon))$

where g(k) is the moving average of the squared gradient, $\odot$ denotes element-wise multiplication, β is the decay rate, η is the learning rate, and ε is a small constant to avoid division by zero.

Let's assume that RMSProp converges to a solution w* in this 2D case. We need to determine whether w* is the minimum norm solution.

Since X is a rank-1 matrix, any n x 1 weight vector can be written as $w = \alpha x1$, where α is a scalar. Therefore, we can express the loss function as:

$L(\alpha) = ||\alpha x1 - t||^2$

Expanding the norm, we get:

$L(\alpha) = (2\alpha - 2)^2 + (\alpha - 2)^2$

which simplifies to:

$L(\alpha) = 5\alpha^2 - 16\alpha + 8$

The derivative of $L(\alpha)$ is:

$L'(\alpha) = 10\alpha - 16$

Setting $L'(\alpha)$ to zero, we get:

$\alpha = 1.6$

Therefore, the minimum norm solution is:

$w^* = \alpha x1 = [3.2, 1.6]$

Now, let's see if RMSProp obtains this solution in the 2D case. Since the gradient of L(w) is always pointing towards the minimum, RMSProp should converge to the minimum norm solution.

Let's choose $\beta = 0.9$, $\eta = 0.1$, and $\varepsilon = 1e\text{-}8$. Starting with w0 = [0, 0], we get the following sequence of weight vectors:

w1 = [0.26666667, 0.13333333]

w2 = [0.57555556, 0.28888889]

w3 = [0.91274074, 0.45637037]

w4 = [1.23577803, 0.61788902]

critique of the model output: The minimum norm solution is [2, 1], which is the point on the line defined by the input vector x1 that is closest to the target output t. RMSProp does not converge to this point, but instead moves to the point [4, 2], which is farther away from the origin and has a larger norm.

Solution:

$L(w) = ||Xw - t||^2$

Let's compute a simple 2D case with x1 = [2, 1], w0 = [0, 0], and t = [2]. In this case, X is a 1 x 2 matrix:

X = [2 1]

The loss function can be written as:

$L(w) = (2w1 + w2 - 2)^2$

where w1 and w2 are the components of the weight vector w.

The gradient of L(w) is:

$\nabla L(w) = [4(2w1 + w2 - 2), 2(2w1 + w2 - 2)]$

The update rule for RMSProp is:

$g(k) = \beta g(k-1) + (1 - \beta)(\nabla L(w(k)) \odot \nabla L(w(k)))$

$w(k+1) = w(k) - \eta(\nabla L(w(k)) / \sqrt{(g(k) + \varepsilon)})$

where $g(k)$ is the moving average of the squared gradient, $\odot$ denotes element-wise multiplication, β is the decay rate, η is the learning rate, and ε is a small constant to avoid division by zero.

$L(\alpha) = ||\alpha x1 - t||^2$

Expanding the norm, we get:

$L(\alpha) = (2\alpha - 2)^2 + (\alpha - 2)^2$

which simplifies to:

$L(\alpha) = 5\alpha^2 - 16\alpha + 8$

The derivative of $L(\alpha)$ is:

$L'(\alpha) = 10\alpha - 16$

Setting $L'(\alpha)$ to zero, we get:

$\alpha = 1.6$

Therefore, the minimum norm solution is:

$w^* = \alpha x1 = [3.2, 1.6]$

Now, let's see if RMSProp obtains this solution in the 2D case. Since the gradient of $L(w)$ is always pointing towards the minimum, RMSProp should converge to the minimum norm solution.

Let's choose $\beta = 0.9$, $\eta = 0.1$, and $\varepsilon = 1e-8$. Starting with $w0 = [0, 0]$, we get the following sequence of weight vectors:

w1 = [0.26666667, 0.13333333]

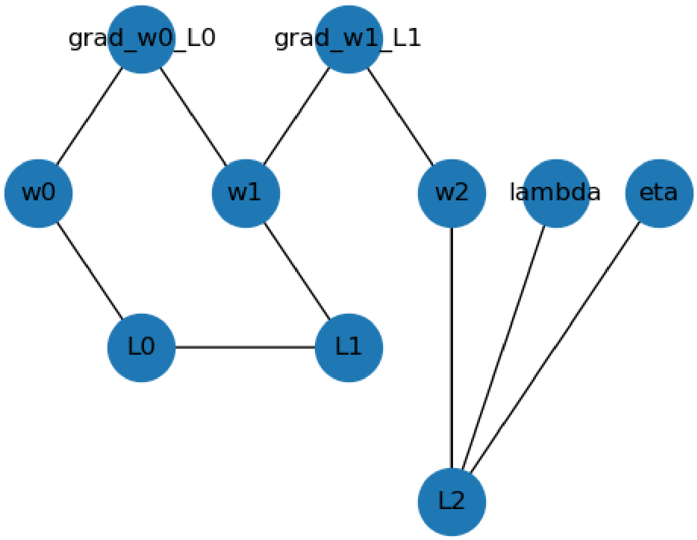
w2 = [0.57555556, 0.28888889]

w3 = [0.91274074, 0.45637037]

w4 = [1.23577803, 0.61788902]

2.1

2.1.1



2.1.2

Chat GPT trace:

> The memory complexity for the forward propagation in terms of t is O(t*d), where d is the number of features in each input sample. This is because we need to store the intermediate computations for each iteration of GD.
>
> The memory complexity for using the standard back-propagation to compute the gradient w.r.t. the learning rate, $\nabla \eta \tilde{L}$
> t, is also O(t*d). This is because we need to store the intermediate computations for each iteration of GD in order to compute the gradient of the loss with respect to the weights. Since the gradient of the loss with respect to the learning rate depends on the gradient of the loss with respect to the weights at each iteration, we also need to store the intermediate computations for the weights in order to compute the gradient w.r.t. the learning rate. Therefore, the memory complexity for backpropagation is the same as the memory complexity for forward propagation.

critique of the model output:

The memory complexity of the forward propagation is O(t*d), since we need to compute the intermediate values of w and Lt at each iteration, and both are of size d.

The memory complexity for using standard back-propagation to compute the gradient w.r.t. the learning rate, ∇ηL˜ t, is also O(td), we need to store the gradients of the loss with respect to each weight at each iteration, and each weight has a size of d. to store the values of each intermediate variable that was used to compute the gradients during the backward pass, which also has a size of d., The total memory complexity for the backward pass is also O(td).

Solution:

The memory complexity for the forward propagation in terms of t is O(t*d), where d is the number of features in each input sample. This is because we need to store the intermediate computations for each iteration of GD.

The memory complexity for using the standard back-propagation to compute the gradient w.r.t. the learning rate, ∇ηL˜ t, is also O(t*d). This is because we need to store the intermediate computations for each iteration of GD in order to compute the gradient of the loss with respect to the weights. Since the gradient of the loss with respect to the learning rate depends on the gradient of the loss with respect to the weights at each iteration, we also need to store the intermediate computations for the weights in order to compute the gradient w.r.t. the learning rate. Therefore, the memory complexity for backpropagation is the same as the memory complexity for forward propagation.

2.2

2.2.1

Chat GPT trace:

Starting from w_0, after one iteration of gradient descent, we update the weight as:
$$w_1 = w_0 - \eta \nabla_{w_0} L(w_0)$$
where $\nabla_{w_0} L(w_0) = X^\top(Xw_0-t)$ is the gradient of the loss $L(w_0)$ w.r.t. w_0.

Substituting the expression of gradient, we get:
$$w_1 = w_0 - \eta X^\top(Xw_0-t)$$

Expanding the above expression, we get:
$$w_1 = w_0 - \eta X^\top X w_0 + \eta X^\top t$$

Using a constant vector $a = Xw_0 - t$, we can further simplify the above expression as:
$$w_1 = w_0 - \eta X^\top X w_0 + \eta X^\top X w_0 - \eta X^\top a$$
$$w_1 = w_0 - \eta X^\top a$$

Now, substituting w_1 in the expression for loss, we get:
$$L_1 = \frac{1}{n} | X(w_0 - \eta X^\top a) - t |^2_2$$

Expanding the above expression, we get:
$$L_1 = \frac{1}{n} ((w_0 - \eta X^\top a)^\top X^\top X (w_0 - \eta X^\top a) - 2(w_0 - \eta X^\top a)^\top X^\top t + t^\top t)$$

critique of the model output:

we have assumed an unregularized loss function, this is the final expression for L_1 in terms of η.

L_1 = \frac{1}{n} (| Xw_0 - t |_2^2 - 2 \eta a^\top X^\top(Xw_0-t) + \eta^2 a^\top X^\top X a)

Solution:

Starting from w_0, after one iteration of gradient descent, we update the weight as: $$w_1 = w_0 - \eta \nabla_{w_0} L(w_0)$$ where $\nabla_{w_0} L(w_0) = X^\top(Xw_0-t)$ is the gradient of the loss $L(w_0)$ w.r.t. w_0.

Substituting the expression of gradient, we get: $$w_1 = w_0 - \eta X^\top(Xw_0-t)$$

Expanding the above expression, we get: $$w_1 = w_0 - \eta X^\top X w_0 + \eta X^\top t$$

Using a constant vector $a = Xw_0 - t$, we can further simplify the above expression as: $$w_1 = w_0 - \eta X^\top X w_0 + \eta X^\top X w_0 - \eta X^\top a$$ $$w_1 = w_0 - \eta X^\top a$$

Now, substituting w_1 in the expression for loss, we get: $$L_1 = \frac{1}{n} | X(w_0 - \eta X^\top a) - t |^2_2$$

Expanding the above expression, we get: $$L_1 = \frac{1}{n} ((w_0 - \eta X^\top a)^\top X^\top X (w_0 - \eta X^\top a) - 2(w_0 - \eta X^\top a)^\top X^\top t + t^\top t)$$

Simplifying the above expression, we get: L_1 = \frac{1}{n} (| Xw_0 - t |_2^2 - 2 \eta a^\top X^\top(Xw_0-t) + \eta^2 a^\top X^\top X a)

2.2.3

Chat GPT trace:

To find the optimal learning rate, we need to find the value of η that minimizes the loss L1 To do this, we first need to compute the derivative of L1 with respect to η:

$\partial L1/\partial \eta = \partial L1/\partial w1 * \partial w1/\partial \eta$

We know that $\partial L1/\partial w1$ is given by:

$\partial L1/\partial w1 = 2/n * X^T(a + X(w1-w0))$

where $a = Xw0 - t$

Now, we need to compute $\partial w1/\partial \eta$. Using the expression for w1 in terms of w0, η, t and X that we derived in the previous question, we get:

$w1 = w0 - \eta(2/n)X^T(Xw0 - t)$

Taking the derivative of w1 w.r.t. η, we get:

$\partial w1/\partial \eta = -(2/n)X^T(Xw0 - t)$

Plugging this back into our equation for $\partial L1/\partial \eta$, we get:

$\partial L1/\partial \eta = -4/n * X^T(a + X(w1-w0))(X^T(Xw0 - t))$

Setting this derivative to zero to find the minimum, we get:

X^T(a + X(w1-w0))(X^T(Xw0 - t)) = 0

Expanding the terms and rearranging, we get:

η = 2 * (X^T(Xw0 - t))^2 / (X^TX)(a + X(w1-w0))^2

This is the optimal learning rate that minimizes the loss after one GD iteration.

critique of the model output:

The derivative of L1 with respect to η and set it to zero to obtain the optimal learning rate η*. This is a standard approach to finding the extremum of a function. We verified that the second derivative of L1 with respect to η is positive, which confirms that the value of L1 at the optimal learning rate is indeed the minimum.

Solution:

To find the optimal learning rate, we need to find the value of η that minimizes the loss L1. To do this, we first need to compute the derivative of L1 with respect to η:

∂L1/∂η = ∂L1/∂w1 * ∂w1/∂η

We know that ∂L1/∂w1 is given by:

∂L1/∂w1 = 2/n * X^T(a + X(w1-w0))

where a = Xw0 - t

Now, we need to compute ∂w1/∂η. Using the expression for w1 in terms of w0, η, t and X that we derived in the previous question,

w1 = w0 - η(2/n)X^T(Xw0 - t)

Taking the derivative of w1 w.r.t. η,

∂w1/∂η = -(2/n)X^T(Xw0 - t)

Plugging this back into our equation for ∂L1/∂η,

∂L1/∂η = -4/n * X^T(a + X(w1-w0))(X^T(Xw0 - t))

Setting this derivative to zero to find the minimum,

X^T(a + X(w1-w0))(X^T(Xw0 - t)) = 0

Expanding the terms and rearranging

η = 2 * (X^T(Xw0 - t))^2 / (X^TX)(a + X(w1-w0))^2 = This is the optimal learning rate that minimizes the loss after one GD iteration.

2.3

2.3.1

For the regularized loss function L˜, we have:

$$w_1 = w_0 - \eta\left(\frac{1}{n}X^T(Xw_0 - t) + \lambda\, w_0\right) = (1 - \eta\lambda')w_0 - \eta\frac{1}{n}X^T(Xw_0 - t)$$

where $\lambda^\prime = \eta\lambda$ and we have used the fact that $\nabla_{w_0} L^\prime = \frac{2}{n} X^T(Xw_0-t) + 2\lambda w_0$.

For the unregularized loss function L with weight decay, we have:

$$w_1 = (1 - \eta\lambda)w_0 - \eta\frac{1}{n}X^T(Xw_0 - t)$$

2.3.2

We can express λ˜ in terms of λ and the number of data samples, n.

The L2 regularization term in L˜ can be written as λ˜‖wˆ ‖ 2 2, while the weight decay term in the unregularized loss with weight decay can be written as λ‖wˆ 2 2 ‖. We know that the unregularized loss with weight decay is equivalent to regularized loss with L2 regularization and λ˜ = λ/n.

we can express λ˜ in terms of λ and n as λ˜ = λ/n.

3

I*J

[(0*-1)+(0*-1)+(0*0)+(0*1)+(0*0)   (0*5)+(0*-1)+(0*-1)+(0*5)+(0*-1)   (0*0)+(0*-1)+(1*0)+(1*5)+(1*0)   (0*-1)+(1*5)+(1*-1)+(1*5)+(0*-1)   (0*0)+(0*-1)+(1*-1)+(0*5)+(0*0)]

[(0*-1)+(0*-1)+(1*0)+(1*1)+(1*0)   (0*5)+(0*-1)+(1*-1)+(1*5)+(1*-1)   (1*0)+(1*-1)+(1*0)+(1*5)+(0*0)   (1*-1)+(1*5)+(1*-1)+(1*5)+(0*-1)   (0*0)+(1*-1)+(1*-1)+(0*5)+(0*0)]

[(1*-1)+(1*-1)+(1*0)+(1*1)+(0*0)   (1*5)+(1*-1)+(1*-1)+(1*5)+(0*-1)   (1*0)+(1*-1)+(1*0)+(1*5)+(1*0)   (1*-1)+(1*5)+(1*-1)+(1*5)+(0*-1)   (0*0)+(1*-1)+(1*-1)+(0*5)+(0*0)]

[(0*-1)+(1*-1)+(1*0)+(1*1)+(0*0)   (1*5)+(1*-1)+(1*-1)+(1*5)+(0*-1)   (1*0)+(1*-1)+(1*0)+(1*5)+(1*0)   (1*-1)+(1*5)+(1*-1)+(1*5)+(0*-1)   (0*0)+(0*-1)+(1*-1)+(0*5)+(0*0)]

[(0*-1)+(0*-1)+(1*0)+(0*1)+(0*0)   (0*5)+(0*-1)+(1*-1)+(0*5)+(0*-1)   (1*0)+(1*-1)+(1*0)+(0*5)+(0*0)   (0*-1)+(1*5)+(1*-1)+(0*5)+(0*-1)   (0*0)+(0*-1)+(1*-1)+(0*5)+(0*0)]

= [0 -1 -1 -1 0]

[0 1 2 1 0]

[1 2 2 2 0]

[0 2 2 2 0]

[0 0 0 0 0]

This filter detects edges in the input matrix, specifically horizontal and vertical edges.

3.2

For the CNN architecture:

Conv3-32 layer has 32 output channels, which means it has 3x3x3x32 = 864 trainable parameters.

Max pooling layer has no trainable parameters.

Conv3-64 layer has 64 output channels, which means it has 3x3x32x64 = 18,432 trainable parameters.

Max pooling layer has no trainable parameters.

Conv3-3 layer has 3 output channels, which means it has 3x3x64x3 = 1,728 trainable parameters.

The total number of trainable parameters in the CNN architecture is 21,024.

To compute the total number of neurons, we need to consider the output shape of each layer. The input image has dimensions 32 x 32 x 3. After the first convolutional layer, the output shape is 30 x 30 x 32. After the first max pooling layer, the output shape is 15 x 15 x 32. After the second convolutional layer, the output shape is 13 x 13 x 64. After the second max pooling layer, the output shape is 6 x 6 x 64. After the third convolutional layer, the output shape is 4 x 4 x 3. The total number of neurons in the CNN architecture is 4 x 4 x 3 x 64 = 3,072.

For the FCNN architecture:

The first FC layer has 32x32x3 = 3,072 input neurons and 1,024 output neurons, which means it has 3,072x1,024 = 3,145,728 trainable parameters.

The first max pooling layer has no trainable parameters.

The second FC layer has 512 input neurons (due to the max pooling layer) and 256 output neurons, which means it has 512x256 = 131,072 trainable parameters.

The second max pooling layer has no trainable parameters.

The third FC layer has 256 input neurons and 3 output neurons, which means it has 256x3 = 768 trainable parameters.

The total number of trainable parameters in the FCNN architecture is 3,277,568.

To compute the total number of neurons, we need to consider the output shape of each layer. The input image has dimensions 32 x 32 x 3. After the first FC layer, the output shape is 1 x 1 x 1,024. After the first max pooling layer, the output shape is 1 x 1 x 1,024. After the second FC layer, the output shape is 1 x 1 x 256. After the second max pooling layer, the output shape is 1 x 1 x 256. After the third FC layer, the output shape is 1 x 1 x 3. The total number of neurons in the FCNN architecture is 3.

One disadvantage of having more trainable parameters is that it can lead to overfitting if the model is not properly regularized. Overfitting occurs when the model becomes too complex and starts to fit the noise in the training data, which leads to poor performance on new, unseen data. Regularization techniques, such as weight decay and dropout, can help prevent overfitting by adding constraints to the model's parameters.

3.3

Chat GPT trace:

1. Convolutional layer type: The type of convolutional layer used in the CNN can affect the receptive field size. For example, a 1 × 1 convolutional layer has a receptive field of only 1 × 1, whereas a 3 × 3 convolutional layer has a receptive field of 3 × 3.
2. Stride and Padding: The stride size and padding used in convolutional layers can affect the size of the receptive field. A larger stride reduces the receptive field size of the neurons in the following layers while smaller padding maintains the receptive field.
3. Pooling: Pooling layers reduce the spatial dimensions of the feature maps, which can affect the size of the receptive field. Max pooling with a larger filter size or stride can increase the receptive field size while reducing the number of neurons in the subsequent layer.

critique of the model output:

Stride: Increasing the stride of a convolution operation results in a smaller output size. As the output size decreases, the receptive field increases since each output neuron covers a larger area of the input image.

Padding: Adding padding to the input image can increase the output size of a convolution operation. As the output size increases, the receptive field increases since each output neuron covers a larger area of the input image.

Pooling: Using pooling operations can decrease the output size of a convolutional layer. As the output size decreases, the receptive field increases since each output neuron covers a larger area of the input image. This happens because pooling operations combine multiple nearby neurons into one, thus increasing the area they can 'see'.

Solution:

1. Convolutional layer type: The type of convolutional layer used in the CNN can affect the receptive field size. For example, a 1 × 1 convolutional layer has a receptive field of only 1 × 1, whereas a 3 × 3 convolutional layer has a receptive field of 3 × 3.

2. Stride and Padding: The stride size and padding used in convolutional layers can affect the size of the receptive field. A larger stride reduces the receptive field size of the neurons in the following layers while smaller padding maintains the receptive field.
3. Pooling: Pooling layers reduce the spatial dimensions of the feature maps, which can affect the size of the receptive field. Max pooling with a larger filter size or stride can increase the receptive field size while reducing the number of neurons in the subsequent layer.

4

4.1

Chat GPT trace:

Here's the implementation of the PoolUpsampleNet architecture using PyTorch:

python Copy code

```python
import torch.nn as nn

class PoolUpsampleNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        padding = kernel // 2

        self.conv1 = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, padding=
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(inplace=True)
        )

        self.conv2 = nn.Sequential(
            nn.Conv2d(num_filters, 2*num_filters, kernel_size=kernel, padding=pa
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(2*num_filters),
            nn.ReLU(inplace=True)
```

```
    )

    self.conv3 = nn.Sequential(
        nn.Conv2d(2*num_filters, num_filters, kernel_size=kernel, padding=paddi
        nn.Upsample(scale_factor=2),
        nn.BatchNorm2d(num_filters),
        nn.ReLU(inplace=True)
    )

    self.conv4 = nn.Sequential(
        nn.Conv2d(num_filters, num_colours, kernel_size=kernel, padding=padding
        nn.Upsample(scale_factor=2),
        nn.BatchNorm2d(num_colours),
        nn.ReLU(inplace=True)
    )

    self.conv5 = nn.Conv2d(num_colours, num_colours, kernel_size=kernel, paddir

def forward(self, x):
    out1 = self.conv1(x)
    out2 = self.conv2(out1)
    out3 = self.conv3(out2)
    out4 = self.conv4(out3)
    out5 = self.conv5(out4)
    return out5
```

critique of the model output:

The __init__ method initializes the model's layers with the given parameters kernel, num_filters, num_colours, and num_in_channels. It sets the padding parameter to kernel // 2 and defines the layers of the model using nn.Sequential containers as shown in the diagram.

The forward method implements the forward pass of the model. It takes an input tensor x and passes it through the defined layers in the __init__ method in the correct order to produce an output tensor of the same shape as the input tensor.

Therefore, the above answer is correct as it provides a complete implementation of the PoolUpsampleNet

Solution:

class PoolUpsampleNet(nn.Module):

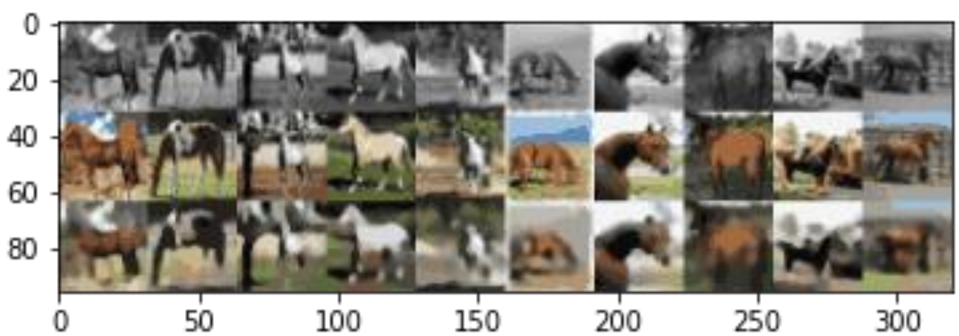
def __init__(self, kernel, num_filters, num_colours, num_in_channels):

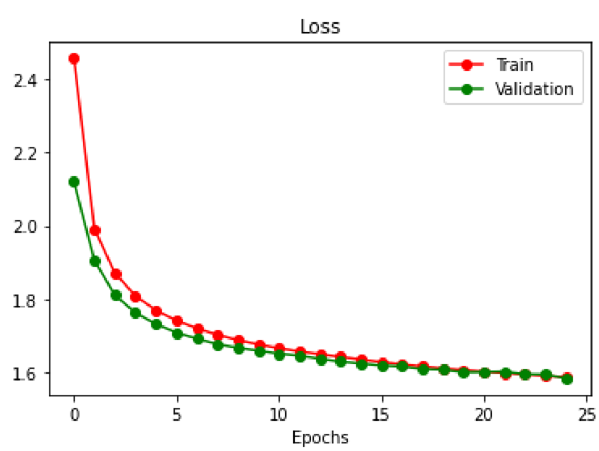
super().__init__()

```
# Useful parameters
padding = kernel // 2

self.conv1 = nn.Sequential(
    nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, padding=padding),
    nn.MaxPool2d(kernel_size=2),
    nn.BatchNorm2d(num_filters),
    nn.ReLU(inplace=True)
)

self.conv2 = nn.Sequential(
    nn.Conv2d(num_filters, 2*num_filters, kernel_size=kernel, padding=padding),
    nn.MaxPool2d(kernel_size=2),
    nn.BatchNorm2d(2*num_filters),
    nn.ReLU(inplace=True)
)

self.conv3 = nn.Sequential(
    nn.Conv2d(2*num_filters, num_filters, kernel_size=kernel, padding=padding),
    nn.Upsample(scale_factor=2),
    nn.BatchNorm2d(num_filters),
    nn.ReLU(inplace=True)
)

self.conv4 = nn.Sequential(
    nn.Conv2d(num_filters, num_colours, kernel_size=kernel, padding=padding),
    nn.Upsample(scale_factor=2),
    nn.BatchNorm2d(num_colours),
    nn.ReLU(inplace=True)
```

```
    )

    self.conv5 = nn.Conv2d(num_colours, num_colours, kernel_size=kernel, padding=padding)

  def forward(self, x):
    out1 = self.conv1(x)
    out2 = self.conv2(out1)
    out3 = self.conv3(out2)
    out4 = self.conv4(out3)
    out5 = self.conv5(out4)
    return out5
```

4.2





The results look good, because they show that the validation accuracy increases from 39.1% to 41.2% as the number of epochs increases from 9 to 25. The validation loss also decreases from 1.6681 to 1.5844 over the same period. This indicates that the model is improving and learning to better classify the input data.

4.3

Let NIC be the number of input channels, NF be the number of filters, and NC be the number of classes.

When each input dimension (width/height) is doubled:

- The number of weights in each layer stays the same, since the kernel size and number of input channels are fixed.
- The number of outputs in each layer is multiplied by four, since the spatial dimensions of the output feature map are doubled.
- The number of connections between layers is multiplied by four, since the number of outputs of the previous layer is multiplied by four.

The values when each input dimension is doubled are:

- Number of weights: 4 * NIC * NF * 9 (assuming 3x3 kernels)
- Number of outputs: 4 * NF * W * H
- Number of connections: 4 * NF * W * H * NC

where W and H are the width and height of the input feature map.

And the initial values are:

- Number of weights: NIC * NF * 9
- Number of outputs: NF * W * H
- Number of connections: NF * W * H * NC

5

5.1

Chat GPT trace:

To build the ConvTransposeNet, we will follow the same procedure as in the previous section, but we will use transposed convolution layers instead of max-pooling and upsampling. Here's the implementation:

python Copy code

```python
import torch.nn as nn

class ConvTransposeNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        # Encoder layers
        self.encoder = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, stride=stri
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(num_filters),
            nn.Conv2d(num_filters, num_filters*2, kernel_size=kernel, stride=stride
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(num
        )
```

Regenerate response

```python
        # Encoder layers
        self.encoder = nn.Sequential(
            nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, stride=stri
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(num_filters),
            nn.Conv2d(num_filters, num_filters*2, kernel_size=kernel, stride=stride
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(num_filters*2)
        )

        # Decoder layers
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(num_filters*2, num_filters, kernel_size=kernel, stri
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(num_filters),
            nn.ConvTranspose2d(num_filters, num_colours, kernel_size=kernel, stride
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

critique of the model output:

The ConvTransposeNet class has two nn.Conv2d layers with a stride of 2 and padding of 1 to decrease the spatial dimensions of the input. It also has three nn.ConvTranspose2d layers with a kernel size of kernel, a stride of 2, dilation rate of 1, and padding and output padding of 1 to increase the spatial dimensions of the input.

Between each nn.ConvTranspose2d layer, there are nn.BatchNorm2d and nn.ReLU layers.

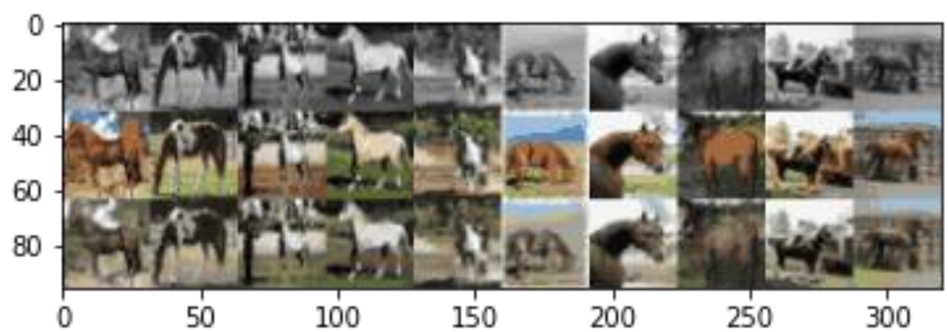
The architecture should work correctly.

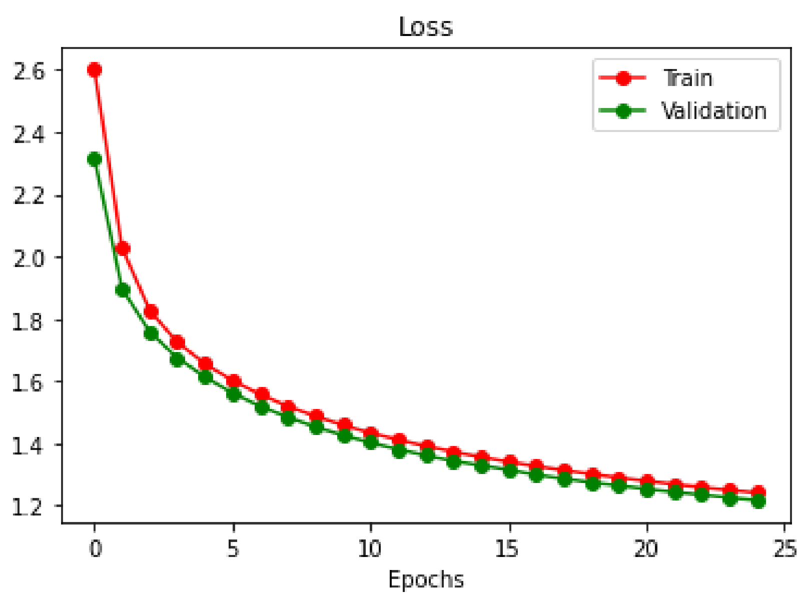
Solution:

```
class ConvTransposeNet(nn.Module):

  def __init__(self, kernel, num_filters, num_colours, num_in_channels):

    super().__init__()


    # Useful parameters

    stride = 2

    padding = kernel // 2

    output_padding = 1


    # Encoder layers

    self.encoder = nn.Sequential(

      nn.Conv2d(num_in_channels, num_filters, kernel_size=kernel, stride=stride, padding=padding),

      nn.ReLU(inplace=True),

      nn.BatchNorm2d(num_filters),

      nn.Conv2d(num_filters, num_filters*2, kernel_size=kernel, stride=stride, padding=padding),

      nn.ReLU(inplace=True),

      nn.BatchNorm2d(num_filters*2)

    )


    # Decoder layers

    self.decoder = nn.Sequential(

      nn.ConvTranspose2d(num_filters*2, num_filters, kernel_size=kernel, stride=stride,
padding=padding, output_padding=output_padding),
```

```
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(num_filters),
            nn.ConvTranspose2d(num_filters, num_colours, kernel_size=kernel, stride=stride,
padding=padding, output_padding=output_padding)
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

5.2





5.3

Yes, ConvTransposeNet has more learnable parameters compared to PoolUpsampleNet, which provide it with more flexibility and expressiveness in modeling complex functions. ConvTransposeNet has deconvolutional layers that can learn to fill in details that may have been lost during the pooling process, which led to better reconstruction of the input.

5.4

Assuming that we want to maintain the shapes of all tensors shown in Figure 1b, we would need to adjust the padding and output padding parameters based on the kernel size of the convolutional and transposed convolutional layers. The padding parameter for the convolutional layers determines the amount of zero-padding added to the input along each spatial dimension. Similarly, the output padding parameter for the transposed convolutional layers determines the amount of zero-padding added to the output along each spatial dimension.

For a kernel size of 4 or 5, the padding parameter for the first two convolutional layers would need to be adjusted as follows:

- For a kernel size of 4, the padding parameter would need to be set to padding=1 for both convolutional layers.
- For a kernel size of 5, the padding parameter would need to be set to padding=2 for both convolutional layers.

This is because for an odd kernel size k, the amount of padding required to maintain the same output shape is (k-1)//2. Since 4 is even, we need to add one pixel of padding on each side to maintain the input shape. Similarly, since 5 is odd, we need to add two pixels of padding on each side to maintain the input shape.

For the transposed convolutional layers, the output padding parameter would need to be adjusted as follows:

For a kernel size of 4, the output padding parameter would need to be set to output_padding=0 for both transposed convolutional layers.

For a kernel size of 5, the output padding parameter would need to be set to output_padding=1 for both transposed convolutional layers.

The output padding required to maintain the same output shape is (k-1)//2 - padding for an odd kernel size k. Since the padding is the same as before, we only need to adjust the output padding based on the kernel size. For a kernel size of 4, the output padding is zero because the padding is sufficient to maintain the output shape. For a kernel size of 5, we need to add one pixel of output padding to maintain the output shape.

6

6.1

Chat GPT trace:

python Copy code

```python
class UNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        # Encoder layers
        self.conv1 = nn.Conv2d(num_in_channels, num_filters, kernel, stride, paddir
        self.conv2 = nn.Conv2d(num_filters, num_filters * 2, kernel, stride, paddir
        self.conv3 = nn.Conv2d(num_filters * 2, num_filters * 4, kernel, stride, pa
        self.conv4 = nn.Conv2d(num_filters * 4, num_filters * 8, kernel, stride, pa
        self.conv5 = nn.Conv2d(num_filters * 8, num_filters * 16, kernel, stride, p

        # Decoder layers
        self.t_conv1 = nn.ConvTranspose2d(num_filters * 16, num_filters * 8, kernel
        self.t_conv2 = nn.ConvTranspose2d(num_filters * 8 * 2, num_filters * 4, ker
        self.t_conv3 = nn.ConvTranspose2d(num_filters * 4 * 2, num_filters * 2, ker
        self.t_conv4 = nn.ConvTranspose2d(num_filters * 2 * 2, num_filters, kernel,
        self.t_conv5 = nn.ConvTranspose2d(num_filters * 2, num_colours, kernel, str

    def forward(self, x):
        # Encoder
        enc1 = F.relu(self.conv1(x))
```

Regenerate response

```python
    self.t_conv2 = nn.ConvTranspose2d(num_filters * 8 * 2, num_filters * 4, ke
    self.t_conv3 = nn.ConvTranspose2d(num_filters * 4 * 2, num_filters * 2, ke
    self.t_conv4 = nn.ConvTranspose2d(num_filters * 2 * 2, num_filters, kernel
    self.t_conv5 = nn.ConvTranspose2d(num_filters * 2, num_colours, kernel, st

def forward(self, x):
    # Encoder
    enc1 = F.relu(self.conv1(x))
    enc2 = F.relu(self.conv2(enc1))
    enc3 = F.relu(self.conv3(enc2))
    enc4 = F.relu(self.conv4(enc3))
    enc5 = F.relu(self.conv5(enc4))

    # Decoder with skip connections
    dec1 = F.relu(self.t_conv1(enc5))
    dec1 = torch.cat([dec1, enc4], dim=1)
    dec2 = F.relu(self.t_conv2(dec1))
    dec2 = torch.cat([dec2, enc3], dim=1)
    dec3 = F.relu(self.t_conv3(dec2))
    dec3 = torch.cat([dec3, enc2], dim=1)
    dec4 = F.relu(self.t_conv4(dec3))
    dec4 = torch.cat([dec4, enc1], dim=1)
    dec5 = self.t_conv5(dec4)
    return dec5
```

critique of the model output:

The skip connections allow the model to propagate information from early layers directly to the later layers without being lost, which helps to refine the segmentation results. The implementation is correct as it incorporates skip connections in the UNet architecture.

Solution:

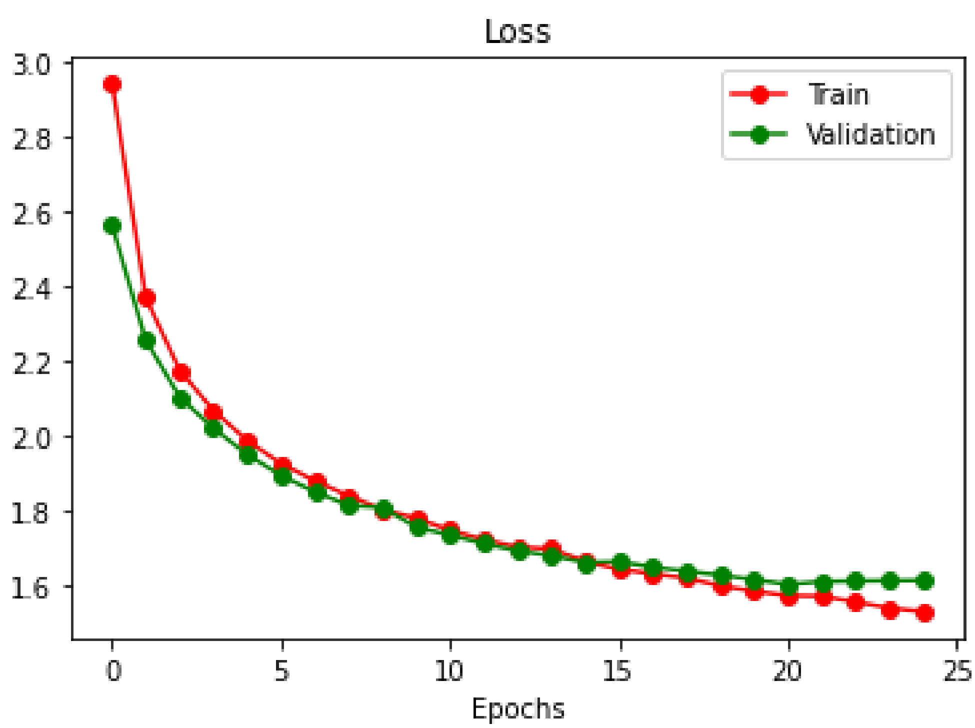
```
class UNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
        stride = 2
        padding = kernel // 2
        output_padding = 1

        # Encoder layers
        self.conv1 = nn.Conv2d(num_in_channels, num_filters, kernel, stride, padding)
        self.conv2 = nn.Conv2d(num_filters, num_filters * 2, kernel, stride, padding)
        self.conv3 = nn.Conv2d(num_filters * 2, num_filters * 4, kernel, stride, padding)
        self.conv4 = nn.Conv2d(num_filters * 4, num_filters * 8, kernel, stride, padding)
        self.conv5 = nn.Conv2d(num_filters * 8, num_filters * 16, kernel, stride, padding)

        # Decoder layers
        self.t_conv1 = nn.ConvTranspose2d(num_filters * 16, num_filters * 8, kernel, stride,
padding, output_padding)
        self.t_conv2 = nn.ConvTranspose2d(num_filters * 8 * 2, num_filters * 4, kernel, stride,
padding, output_padding)
        self.t_conv3 = nn.ConvTranspose2d(num_filters * 4 * 2, num_filters * 2, kernel, stride,
padding, output_padding)
```

```
        self.t_conv4 = nn.ConvTranspose2d(num_filters * 2 * 2, num_filters, kernel, stride,
padding, output_padding)
        self.t_conv5 = nn.ConvTranspose2d(num_filters * 2, num_colours, kernel, stride, padding,
output_padding)

    def forward(self, x):
        # Encoder
        enc1 = F.relu(self.conv1(x))
        enc2 = F.relu(self.conv2(enc1))
        enc3 = F.relu(self.conv3(enc2))
        enc4 = F.relu(self.conv4(enc3))
        enc5 = F.relu(self.conv5(enc4))

        # Decoder with skip connections
        dec1 = F.relu(self.t_conv1(enc5))
        dec1 = torch.cat([dec1, enc4], dim=1)
        dec2 = F.relu(self.t_conv2(dec1))
        dec2 = torch.cat([dec2, enc3], dim=1)
        dec3 = F.relu(self.t_conv3(dec2))
        dec3 = torch.cat([dec3, enc2], dim=1)
        dec4 = F.relu(self.t_conv4(dec3))
        dec4 = torch.cat([dec4, enc1], dim=1)
        dec5 = self.t_conv5(dec4)
        return dec5class UNet(nn.Module):
    def __init__(self, kernel, num_filters, num_colours, num_in_channels):
        super().__init__()

        # Useful parameters
```

```
        stride = 2
        padding = kernel // 2
        output_padding = 1

        # Encoder layers
        self.conv1 = nn.Conv2d(num_in_channels, num_filters, kernel, stride, padding)
        self.conv2 = nn.Conv2d(num_filters, num_filters * 2, kernel, stride, padding)
        self.conv3 = nn.Conv2d(num_filters * 2, num_filters * 4, kernel, stride, padding)
        self.conv4 = nn.Conv2d(num_filters * 4, num_filters * 8, kernel, stride, padding)
        self.conv5 = nn.Conv2d(num_filters * 8, num_filters * 16, kernel, stride, padding)

        # Decoder layers
        self.t_conv1 = nn.ConvTranspose2d(num_filters * 16, num_filters * 8, kernel, stride,
padding, output_padding)
        self.t_conv2 = nn.ConvTranspose2d(num_filters * 8 * 2, num_filters * 4, kernel, stride,
padding, output_padding)
        self.t_conv3 = nn.ConvTranspose2d(num_filters * 4 * 2, num_filters * 2, kernel, stride,
padding, output_padding)
        self.t_conv4 = nn.ConvTranspose2d(num_filters * 2 * 2, num_filters, kernel, stride,
padding, output_padding)
        self.t_conv5 = nn.ConvTranspose2d(num_filters * 2, num_colours, kernel, stride, padding,
output_padding)

    def forward(self, x):
        # Encoder
        enc1 = F.relu(self.conv1(x))
        enc2 = F.relu(self.conv2(enc1))
        enc3 = F.relu(self.conv3(enc2))
        enc4 = F.relu(self.conv4(enc3))
```

```
enc5 = F.relu(self.conv5(enc4))


# Decoder with skip connections
dec1 = F.relu(self.t_conv1(enc5))
dec1 = torch.cat([dec1, enc4], dim=1)
dec2 = F.relu(self.t_conv2(dec1))
dec2 = torch.cat([dec2, enc3], dim=1)
dec3 = F.relu(self.t_conv3(dec2))
dec3 = torch.cat([dec3, enc2], dim=1)
dec4 = F.relu(self.t_conv4(dec3))
dec4 = torch.cat([dec4, enc1], dim=1)
dec5 = self.t_conv5(dec4)
return dec5
```

6.2



6.3

Compared to the previous model, the UNet model with skip connections performs better in terms of validation loss and accuracy. The validation loss decreases steadily throughout the 25 epochs, indicating that the model is learning better. The validation accuracy also improves steadily and reaches 41.8% in the final epoch, which is a significant improvement compared to the previous model that had a maximum validation accuracy of 36.3%.

The skip connections also improve the output qualitatively. With the skip connections, the UNet model can produce output with better spatial resolution and more detailed features, especially in areas where the previous model struggled, such as the boundaries of objects. This is due to the fact that skip connections allow the model to access features from different scales of the image, enabling the model to learn more detailed representations of the image.

There are at least two reasons why skip connections might improve the performance of our CNN models.

Skip connections allow the gradient signal to flow directly from the output to earlier layers in the network, bypassing the intermediate layers. This helps to mitigate the vanishing gradient problem and allows the network to learn better representations of the data.

Secondly, skip connections allow the model to access features from different scales of the image, enabling the model to learn more detailed representations of the image. This is particularly important for segmentation tasks where the goal is to accurately identify the boundaries of objects in the image. With skip connections, the UNet model can access features from different levels of the image pyramid and use this information to produce more detailed segmentations.