L335 Practical

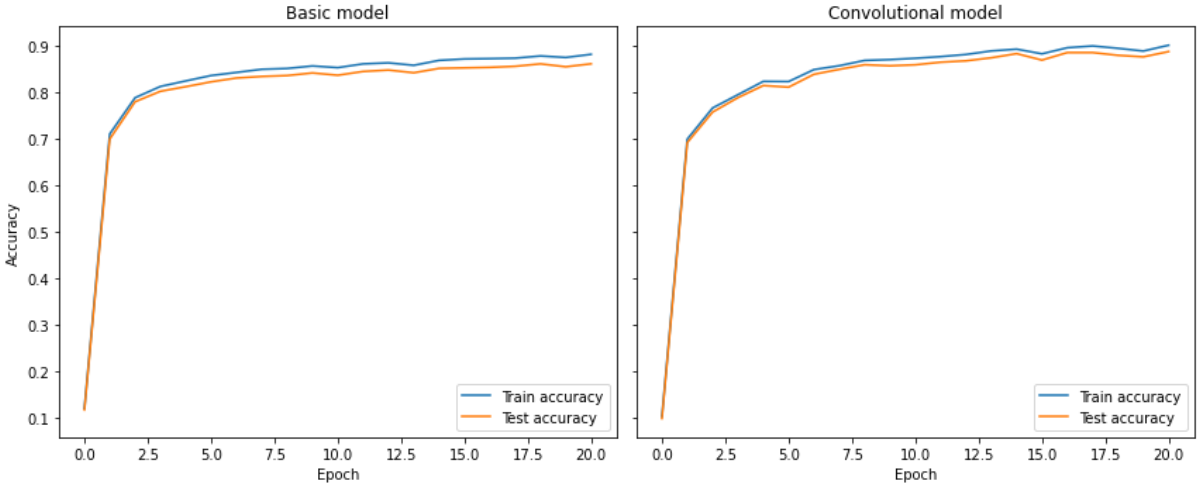
Classification

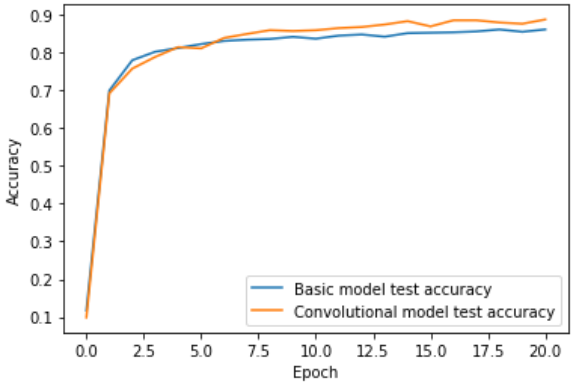
Task 1

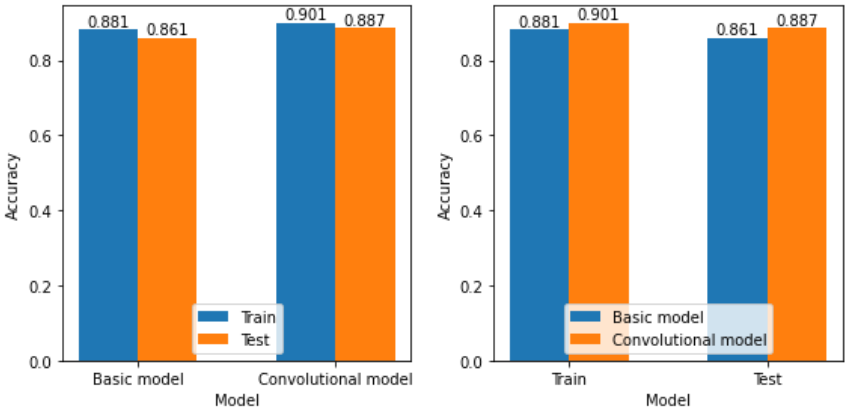
> The current forward model is a basic three-layer fully connected network (FCN). As introduced in the lecture, there are many possibilities, such as a CNN, to tailor the network architecture to better distil the local features of images. The first task is to update the network architecture to a CNN with three convolutional layers and two FC layers. Choose the hyperparameters of the CNN (e.g. the number of output channels, kernel size, hidden layer size, etc. of the convolutional layers, pooling layer, and the FC layers) such that the total number of training parameters is similar to that of the basic net. Retrain the network with the same epoch and batch size and compare the training and test accuracy of the CNN with the basic net.
>
> **Deliverable:** Report the choice of the hyperparameters of your CNN and show the calculation of the number of training parameters. Show a quantitative comparison of the model accuracy of the basic net and the CNN on the training and test data with a plot of accuracy vs. epochs. Report whether the model is under-fitted, well-fitted, or over-fitted, explain the possible causes, and suggest potential solutions. Visualise the feature maps of each convolutional layer.

For my CNN, I kept the parameters the same for all of the convolutional layers. For my CNN, I kept the parameters the same for all of the convolutional layers. For all convolutional layers I introduced padding and for the pooling layers I used a stride of 1, as I found this produced the best results consistently and prevented the dimensionality from rapidly decreasing (since 28x28 isn't very large). For the convolutional portion of the network, I used a kernel size of 5 and produced 9 output-channels, while for the linear portion of the model I used a hidden size of 128. The basic model had 669,706 training parameters while the convolutional model has 662,744. Parameter counts were computed using the following code:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```



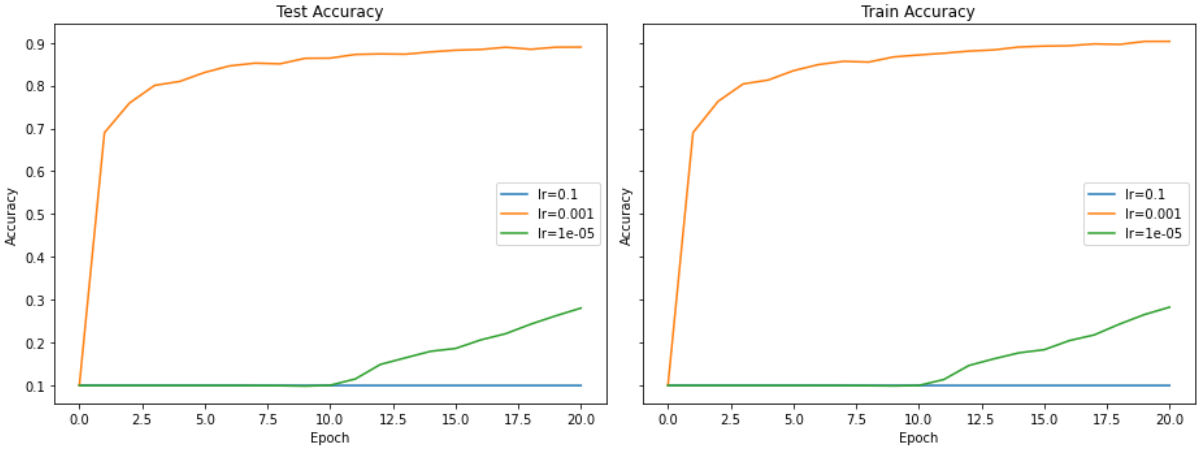


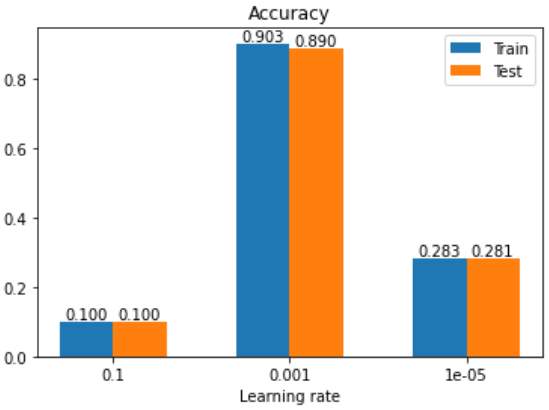


Overall the difference in performance between the basic model and the convolutional model is relatively small, but it is still notable (about 2.6% improvement on the test set and 2% on the training set). Additionally, the convolutional model began outperforming the basic model in only about 5 epochs. Both models are slightly overfitted, having an approximate 2% reduction in performance for the test set compared to the training set for the basic model, and 1.4% for the convolutional model. However, this different is rather small, suggesting a reasonably good fit. This overfitting may potentially be resolved using smarter techniques, such as edge detection.

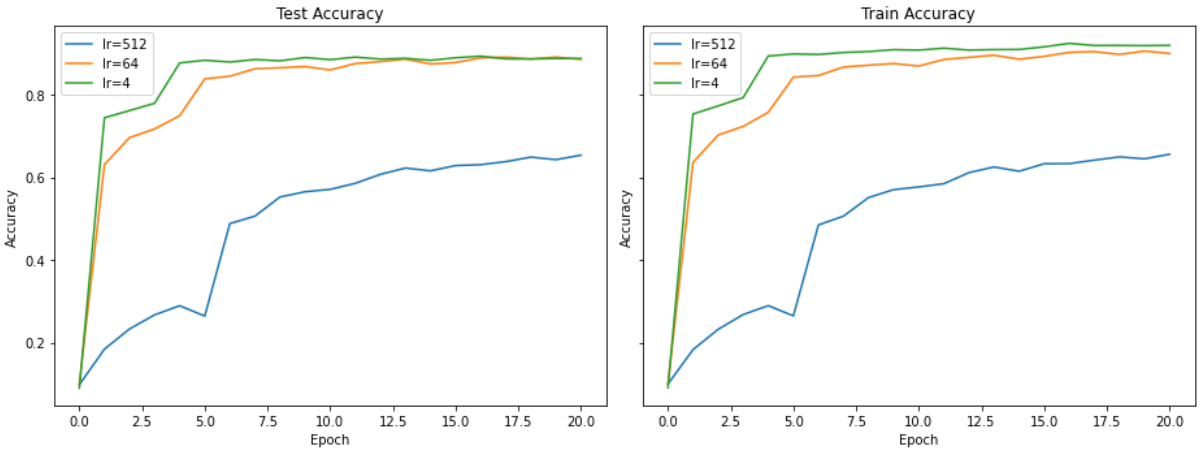
Task 2

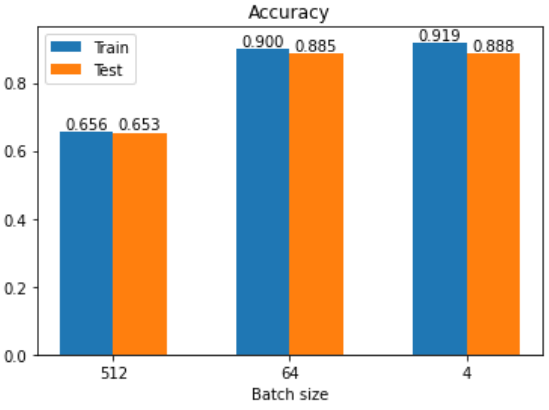
> In addition to the network design, training parameters and algorithms also influence the performance of the network. The second task is to retrain your CNN constructed in task 1 with a different learning rate and batch size.
>
> **Deliverable:** Show a quantitative comparison of the model accuracy on the training and test data with a plot of accuracy vs. epochs with a learning rate of 0.1, 0.001, and 0.00001. Summarise the difference in performance and explain the cause. Fixing the learning rate to be 0.001, show a quantitative comparison of the model accuracy on the training and test data with a plot of accuracy vs. epochs with a batch size of 4, 64, and 512. Summarise the difference in performance and explain the cause.





Based on this experiment, we can see that the learning rate strongly impacts how fast the model learns and to what degree. In the case of `lr=0.1`, the learning rate is so large that the model fails to learn at all, resulting in completely random predictions. This is because the learning rate is so large that the optimizer always overshoots any minima it is optimizing for. With `lr=0.00001` the model fails to fully generalize, but does begin learning very slowly toward the end of the training. This is because the movements of the optimizer are so short that it takes a very long time to actually begin approaching a minima (Note: On a few experimental trials with `lr=0.00001` the model failed to learn anything for the entire duration of training). Notably, with this learning rate the train and test performance is very similar (though admittedly very poor). These contrast significantly from `lr=0.001`, which does successfully learn, reaching a test accuracy of 89%. This is because this learning rate is large enough to approach the minima relatively quickly, but small enough that it doesn't overshoot it (or at least, not as much).





Based on this experiment, the batch size has a relatively minor impact on model accuracy once fully trained, but significantly impacts the dynamics of how fast the model trains. This is because with smaller batch sizes, a much greater number of expensive operations need to occur, causing training to take much longer per epoch. However, based on `batch_size=4`, having a smaller batch size leads to very slightly better final test performance, but also notably more overfitting (test accuracy from batch size of 64 to 4 increases by only 0.3%, while train accuracy increases by 1.9%). Looking at `batch_size=512`, we see lower model accuracy, but the model also had not fully converged by the end of training, and would likely approach similar performance to the other models if trained longer. Additionally, the model trained with larger batch size appeared to have less overfitting, with train and test accuracy only differing by 0.3%. The differences in overfitting occur because using a small batch size causes the optimizer to consider only a small number of samples when calculating its gradients, resulting in model adjustments due to microscopic effects that may not generalize as well to the broader behavior. By contrast, having a larger batch size allows for better consideration of the average (general) behavior. That said, while we did not explore it here, having too large of a batch size can also result in overfitting or underfitting due to being unable to consider microscopic, per-sample behaviors as well. This is why the model trained on a large batch size did not converge before training finished.

Regression

Task 1

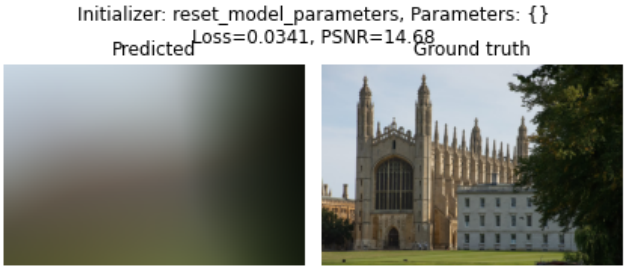
> Change the global seed (line 1 in the training code block) to 1 and retrain the network. This results in a significantly poorer reconstruction. Changing the seed results in a different initial configuration of the weights. In fact, most seeds result in poor

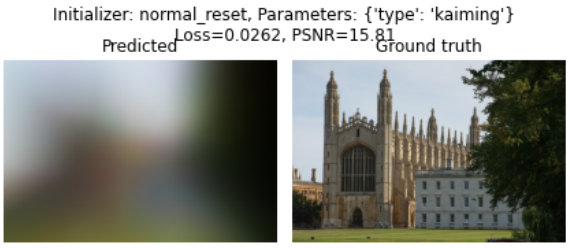
convergence. Can you identify a network initialization function that results in PSNR > 15 regardless of the seed chosen?

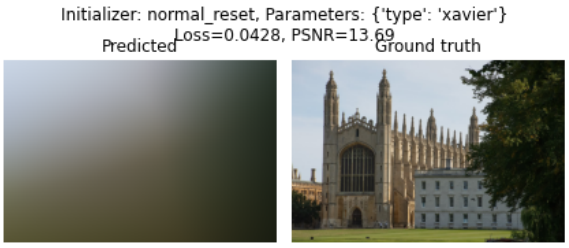
Constraints: For this task, you are **not allowed** to change

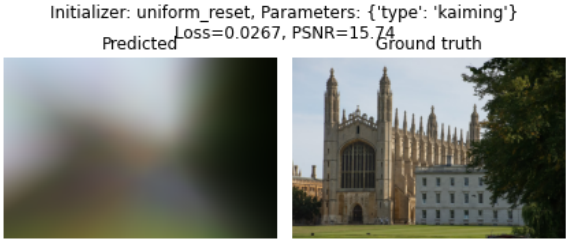
1. Number of training epochs
2. Network size
3. Optimizer (or its parameters such as learning rate, betas)

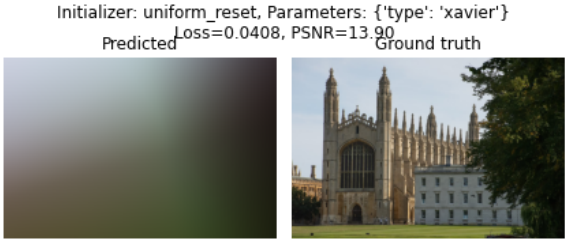
Deliverable: Quantitative comparison showing the effect of different functions for initialising weights. Use the PSNR metric for all tables and plots.

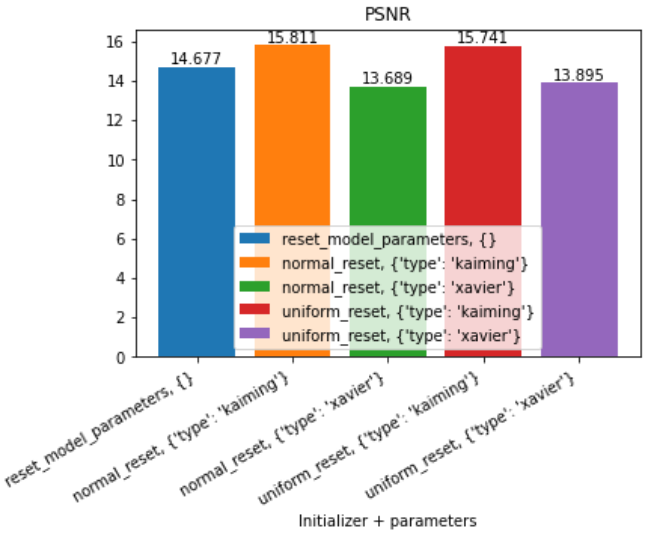












Based on my experimentation, I found that using Kaiming initialization with either the normal or uniform distributions results in improved performance over the default initialization (PSNR of 15.811 and 15.741 respectively compared to 14.677 on the default initialization technique). I also tried Xavier initialization, however I was unable to get this to produce consistent results, often underperforming the default. Notably, looking at the source code for the default `reset_parameters` function, the default initialization technique actually uses Kaiming uniform. However, it does not set the activation technique, causing it to default to Leaky ReLU instead of standard ReLU (what we are using).

I will also note that in order to get consistent results, I had to change the final activation function of the model from ReLU to sigmoid. This felt necessary, as the expected of the model is supposed to be bounded within the `[0,1]` range anyways. This results in slightly different behavior from the code as initially provided, but was necessary to prevent cases where the weights or biases get too large and cause the output to clip. Thus the use of sigmoid improved the stability significantly (though did not noticeably improve performance compared to ReLU on the default initialization with set seed 0).

Task 2

> Experiment with different architectures (deeper or wider networks), different reconstruction losses (L1, Huber) and optimizers. Report your results, again with quantitative metrics. Check if your design choices are consistent for different inputs with possibly different resolutions.
>
> You will notice that the network is unable to fit the image properly because it is limited by the number of input dimensions. You can fix this by *Fourier feature mapping* — by passing the input 2D coordinates through a Positional Encoding function [1] — to better learn high-frequencies in the image.
>
> [1] Tancik, Matthew, et al. "Fourier features let networks learn high frequency functions in low dimensional domains." Advances in Neural Information Processing Systems 33 (2020): 7537-7547. Link: https://proceedings.neurips.cc/paper/2020/hash/55053683268957697aa39fba6f231c68-Abstract.html

> Specifically, you will implement the Gaussian Fourier feature mapping: $\gamma(\mathbf{g}) = [\cos(2\pi\mathbf{B}\mathbf{g}), \sin(2\pi\mathbf{B}\mathbf{g})]^{\mathrm{T}}$,
> where each entry in $\mathbf{B} \in \mathbb{R}^{m\times 2}$ is sampled from $\mathcal{N}(0, \sigma^2)$. Here, m is the number of normally distributed sinosoidal frequencies and $\mathbf{g} = (x, y)$ is an input 2D coordinate.
>
> The Fourier feature mapping $\gamma(\mathbf{g})$ thus provides a $2 \times m$ input to the MLP. In place of the direct mapping, we learn
>
> $$(r, g, b) = f(\gamma(\mathbf{g})).$$
>
> **Deliverable:** Demonstrate the effectiveness of FFN on own your set of diverse images. Different images will benefit from a different choice of σ (standard deviation of frequencies). Show the result of fitting an INR with FFN for an image composed of lower frequencies (such as sky) as well as an image with high frequencies (such as grass).

Across all of my experiments, a σ value of 5 proved to produce the best results. Notably, the lower σ is, the more blurry the output, while the higher it is, the noisier the output. This makes sense with the FFN approach, as a lower standard deviation of frequencies would mean less variation throughout the image, and thus a blurrier image. By contrast, a higher standard deviation results in much more variance, resulting in a lot of color-noise in the image. The impact of this is that images that have a lot of variability across it (i.e., images with lots of detail) would benefit more from a high standard deviation, while relatively uniform images (like the sky image) benefit more from lower standard deviations. Kaiming normal weight initialization was used in all experiments.

Task 3

> The neural network approximates a continuous signal and can be queried at arbitrary resolutions. Your final task is to produce a 4x upsampled image by providing a denser

> input grid to the network. Recall that the grid coordinates are rescaled to the range $[0, 1]$.

Below is the 4x upsampled output using the King's Chapel image. Kaiming normal weight initialization and $\sigma = 5$ was used in training.