```
    train_result = trainer.test(model, train_anom_loader, verbose=False)
    val_result = trainer.test(model, val_anom_loader, verbose=False)
    test_result = trainer.test(model, test_anom_loader, verbose=False)
    result = {"test_acc": test_result[0]["test_acc"], "val_acc": val_result[0]["test_acc"], "train_acc": train_result[0]["test_a

    model = model.to(device)
    return model, result
```

Let's finally train our model. We will use 4 layers with 4 attention heads each. The hidden dimensionality of the model is 256, and we use a dropout of 0.1 throughout the model for good regularization. Note that we also apply the dropout on the input features, as this makes the model more robust against image noise and generalizes better. Again, we use warmup to slowly start our model training.

In [35]:
```
anomaly_model, anomaly_result = train_anomaly(input_dim=train_anom_dataset.img_feats.shape[-1],
                                              model_dim=256,
                                              num_heads=4,
                                              num_classes=1,
                                              num_layers=4,
                                              dropout=0.1,
                                              input_dropout=0.1,
                                              lr=5e-4,
                                              warmup=100)
```

```
GPU available: True, used: True
WARNING: Logging before flag parsing goes to stderr.
I1109 10:43:31.036801 139648634296128 distributed.py:49] GPU available: True, used: True
TPU available: False, using: 0 TPU cores
I1109 10:43:31.038146 139648634296128 distributed.py:49] TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
I1109 10:43:31.039162 139648634296128 accelerator_connector.py:385] LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
Found pretrained model, loading...
/home/phillip/anaconda3/envs/nlp1/lib/python3.7/site-packages/pytorch_lightning/utilities/distributed.py:45: UserWarning: Your t
est_dataloader has `shuffle=True`, it is best practice to turn this off for validation and test dataloaders.
  warnings.warn(*args, **kwargs)
HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', layout=Layout(flex='2'), max=…

HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', layout=Layout(flex='2'), max=…

HBox(children=(FloatProgress(value=1.0, bar_style='info', description='Testing', layout=Layout(flex='2'), max=…
```

We can print the achieved accuracy below.

In [36]:
```
print(f"Train accuracy: {(100.0*anomaly_result['train_acc']):4.2f}%")
print(f"Val accuracy:   {(100.0*anomaly_result['val_acc']):4.2f}%")
print(f"Test accuracy:  {(100.0*anomaly_result['test_acc']):4.2f}%")
```

```
Train accuracy: 97.77%
Val accuracy:   94.38%
Test accuracy:  94.30%
```

With ~94% validation and test accuracy, the model generalizes quite well. It should be noted that you might see slightly different scores depending on what computer/device you are running this notebook. This is because despite setting the seed before generating the test dataset, it is not the same across platforms and numpy versions. Nevertheless, we can conclude that the model performs quite well and can solve the task for most sets. Before trying to interpret the model, let's verify that our model is permutation-equivariant, and assigns the same predictions for different permutations of the input set. For this, we sample a batch from the test set and run it through the model to obtain the probabilities.

In [37]:
```
inp_data, indices, labels = next(iter(test_anom_loader))
inp_data = inp_data.to(device)

anomaly_model.eval()

with torch.no_grad():
    preds = anomaly_model.forward(inp_data, add_positional_encoding=False)
    preds = F.softmax(preds.squeeze(dim=-1), dim=-1)

    # Permut input data
    permut = np.random.permutation(inp_data.shape[1])
    perm_inp_data = inp_data[:,permut]
    perm_preds = anomaly_model.forward(perm_inp_data, add_positional_encoding=False)
    perm_preds = F.softmax(perm_preds.squeeze(dim=-1), dim=-1)

assert (preds[:,permut] - perm_preds).abs().max() < 1e-5, "Predictions are not permutation equivariant"

print("Preds\n", preds[0,permut].cpu().numpy())
print("Permuted preds\n", perm_preds[0].cpu().numpy())
```

```
Preds
 [5.4543594e-05 1.4208173e-04 6.6922468e-05 7.6413504e-05 7.7112330e-05
 8.7848457e-05 6.6820685e-05 9.9929154e-01 7.3219831e-05 6.3545609e-05]
Permuted preds
 [5.4543532e-05 1.4208158e-04 6.6922395e-05 7.6413417e-05 7.7112243e-05
 8.7848362e-05 6.6820678e-05 9.9929142e-01 7.3219751e-05 6.3545544e-05]
```

You can see that the predictions are almost exactly the same, and only differ because of slight numerical differences inside the network operation.

To interpret the model a little more, we can plot the attention maps inside the model. This will give us an idea of what information the model is sharing/communicating between images, and what each head might represent. First, we need to extract the attention maps for the test batch above, and determine the discrete predictions for simplicity.

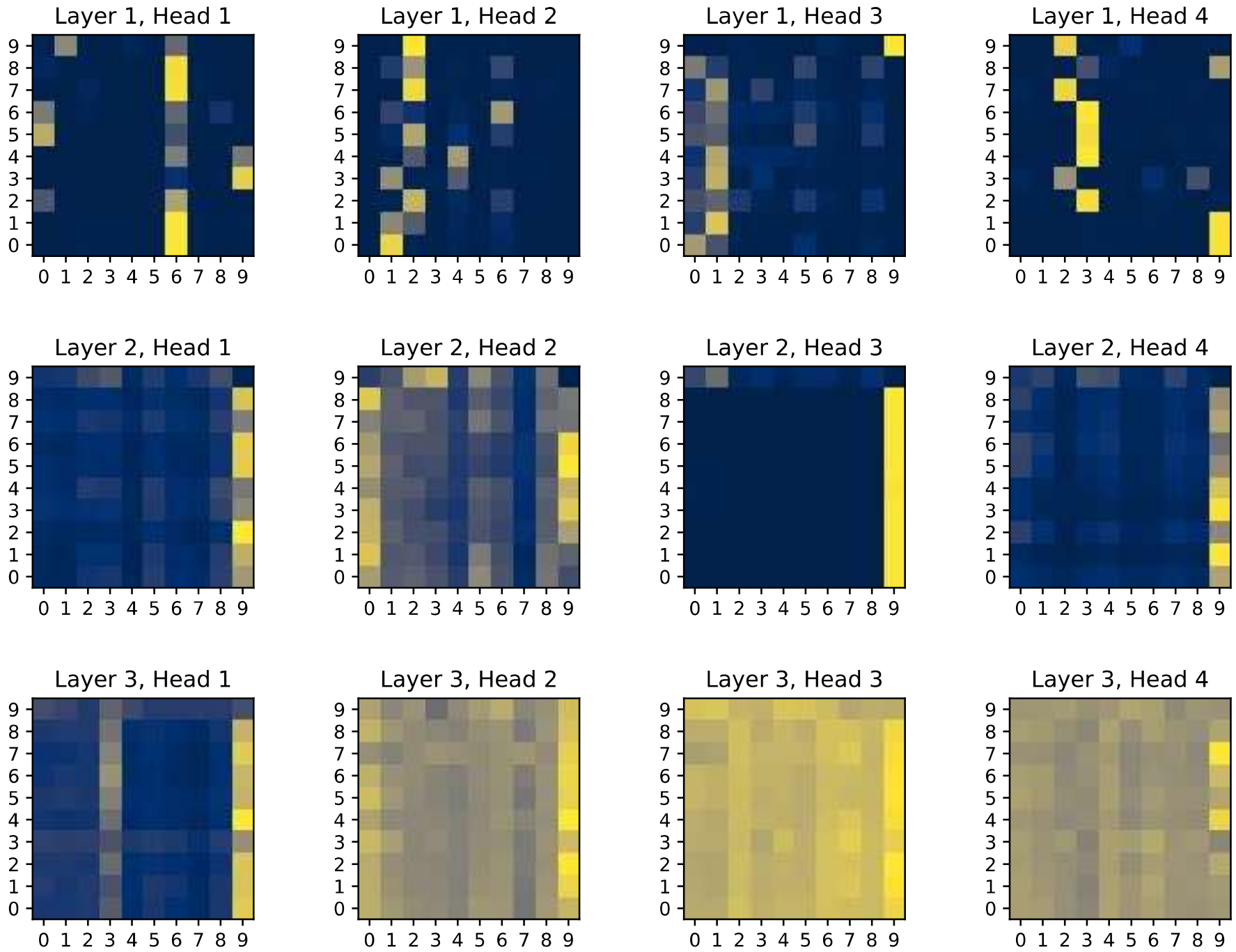
In [38]:
```
attention_maps = anomaly_model.get_attention_maps(inp_data, add_positional_encoding=False)
predictions = preds.argmax(dim=-1)
```

Below we write a plot function which plots the images in the input set, the prediction of the model, and the attention maps of the different heads on layers of the transformer. Feel free to explore the attention maps for different input examples as well.

In [39]:
```
def visualize_prediction(idx):
    visualize_exmp(indices[idx:idx+1], test_set)
    print("Prediction:", predictions[idx].item())
    plot_attention_maps(input_data=None, attn_maps=attention_maps, idx=idx)

visualize_prediction(0)
```


```
Prediction: 9
```


Layer 1, Head 1
Layer 1, Head 2
Layer 1, Head 3
Layer 1, Head 4
Layer 2, Head 1
Layer 2, Head 2
Layer 2, Head 3
Layer 2, Head 4
Layer 3, Head 1
Layer 3, Head 2
Layer 3, Head 3
Layer 3, Head 4
0 1 2 3 4 5 6 7 8 9

Depending on the random seed, you might see a slightly different input set. For the version on the website, we compare 9 tree images with a volcano. We see that multiple heads, for instance, Layer 2 Head 1, Layer 2 Head 3, and Layer 3 Head 1 focus on the last image. Additionally, the heads in Layer 4 all seem to ignore the last image and assign a very low attention probability to it. This shows that the model has indeed recognized that the image doesn't fit the setting, and hence predicted it to be the anomaly. Layer 3 Head 2-4 seems to take a slightly weighted average of all images. That might indicate that the model extracts the "average" information of all images, to compare it to the image features itself.

Let's try to find where the model actually makes a mistake. We can do this by identifying the sets where the model predicts something else than 9, as in the dataset, we ensured that the anomaly is always at the last position in the set.

In [40]:
```
mistakes = torch.where(predictions != 9)[0].cpu().numpy()
print("Indices with mistake:", mistakes)
```

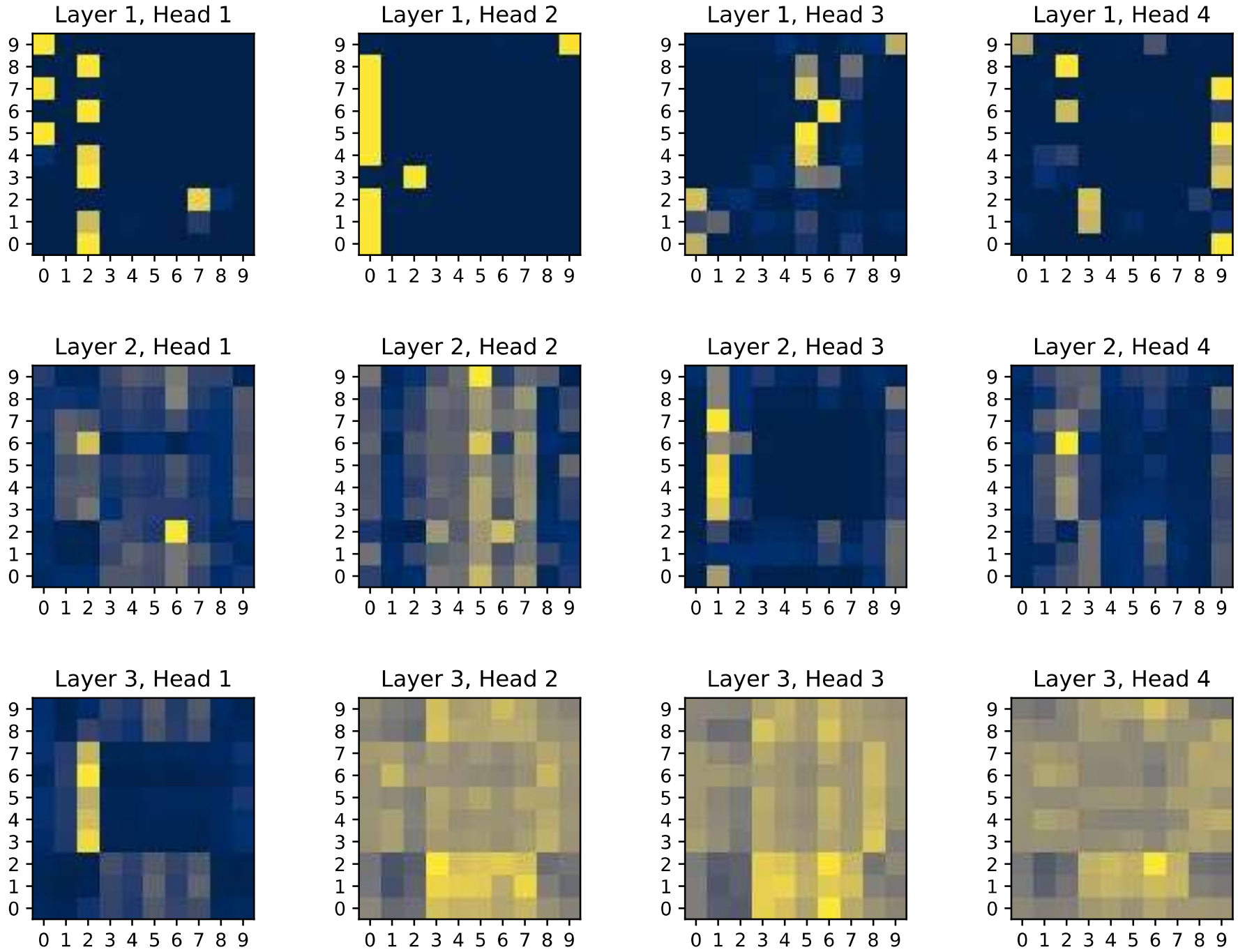
```
Indices with mistake: [36 49]
```

As our model achieves ~94% accuracy, we only have very little number of mistakes in a batch of 64 sets. Still, let's visualize one of them, for example the last one:

In [41]:
```
visualize_prediction(mistakes[-1])
print("Probabilities:")
for i, p in enumerate(preds[mistakes[-1]].cpu().numpy()):
    print(f"Image {i}: {100.0*p:4.2f}%")
```

Anomaly examples on CIFAR100

```
Prediction: 2
```


Layer 1, Head 1
Layer 1, Head 2
Layer 1, Head 3
Layer 1, Head 4
Layer 2, Head 1
Layer 2, Head 2
Layer 2, Head 3
Layer 2, Head 4
Layer 3, Head 1
Layer 3, Head 2
Layer 3, Head 3
Layer 3, Head 4
0 1 2 3 4 5 6 7 8 9

```
Probabilities:
Image 0: 0.06%
Image 1: 1.63%
Image 2: 89.63%
Image 3: 0.01%
Image 4: 0.01%
Image 5: 0.01%
Image 6: 0.01%
Image 7: 0.01%
Image 8: 0.01%
Image 9: 8.63%
```

In this example, the model confuses a palm tree with a building, giving a probability of ~90% to image 2, and 8% to the actual anomaly. However, the difficulty here is that the picture of the building has been taken at a similar angle as the palms. Meanwhile, image 2 shows a rather unusual palm with a different color palette, which is why the model fails here. Nevertheless, in general, the model performs quite well.

Conclusion

In this tutorial, we took a closer look at the Multi-Head Attention layer which uses a scaled dot product between queries and keys to find correlations and similarities between input elements. The Transformer architecture is based on the Multi-Head Attention layer and applies multiple of them in a ResNet-like block. The Transformer is a very important, recent architecture that can be applied to many tasks and datasets. Although it is best known for its success in NLP, there is so much more to it. We have seen its application on sequence-to-sequence tasks and set anomaly detection. Its property of being permutation-equivariant if we do not provide any positional encodings, allows it to generalize to many settings. Hence, it is important to know the architecture, but also its possible issues such as the gradient problem during the first iterations solved