Unravelling Superposition

R.A. Stringer

Table of contents

Preface

Welcome to the course!

In the nascent field of AI Safety and Alignment, superposition is a key topic illustrative of the elusive nature of neural networks and how and what they learn, and how we can develop techniques to achieve greater insight.

The following notebooks will explain the concept and introduce you to some of the technical approches in PyTorch we can use to conduct practical research in the field.

1 Introduction

In the field of AI Alignment, 'Interpretability' is the study of understanding neural networks; what they learn from training data and how they are forming their predictions.

The first step in interpretability is typically to understand the features a neuron is learning from. There would be no mystery if neurons corresponded to verifiable input features. For example, if a neuron fires on dog tails, or on Korean poetry. Since neural networks incorporate non-linearities, this is not always the case, and we will see how small a fraction of features we are able to extract in relation to the number of neurons in a network. This phenonemon is known as 'superposition'.

Let's consider the canonical MNIST machine learning example. MNIST is a dataset of 60,000 training images of handwritten digits, 0-9 (10 classes). Each image is 28x28 pixels, so 784 pixels in total.

In the most interpretable scenario, each neuron in a neural network would correspond to a specific feature of the input, for example:

- loops (for 0, 6, 8, 9 etc)
- straight lines (1, 4, 7)
- curves (2, 3, 5)

In the image below, we see the original handwritten digits, and a heatmap showing the regions of the digits that had high predictive value. Notice for example how the criss-cross of the eight, which is unique to the digit, is highlighted with blue (strong correlation), and similarly, the curve of the five.

In practice, however, neural nets don't learn clean, one-to-one mappings between neurond and features.

1.0.1 Why does superposition occur?

- Efficiency: networks often have fewer parameters than the number of features they need to represent. In the case of large language models, for example, this means networks do not have to extend to every last possible feature of language found in vast text corpora.
- Generalization: overlapping representations help generalize to new data.

- Non-linearity: this allows for complex, overlapping representations. For example, the combination of features in different neurons, sushi in one and recipie quantities in another, can help formulate accurate predictions.

We will see in this short course that neural networks often represent more features than they have dimensions, and mix different, unrelated concepts in single neurons. For example, a neuron in a language model could fire in response to inputs as varied as code in Haskell, Spanish poetry and vehicle descriptions.

Let's move on to the next chapter to examine this fascinating field and look closely at what we can see neural networks are doing, and what we yet cannot.

2 Explorations in Superposition

```
!pip install torchviz torch transformers torchvision
```

Let's make a sinple neural network of two layers and a fully-connected layer.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)  # 784 input features
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten the input
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Create the model and move it to GPU
model = SimpleNet().to(device)
```

There are various ways of building a more intuitive understanding of the model we just made. The simplest is to print its architecture:

```
print(model)
```

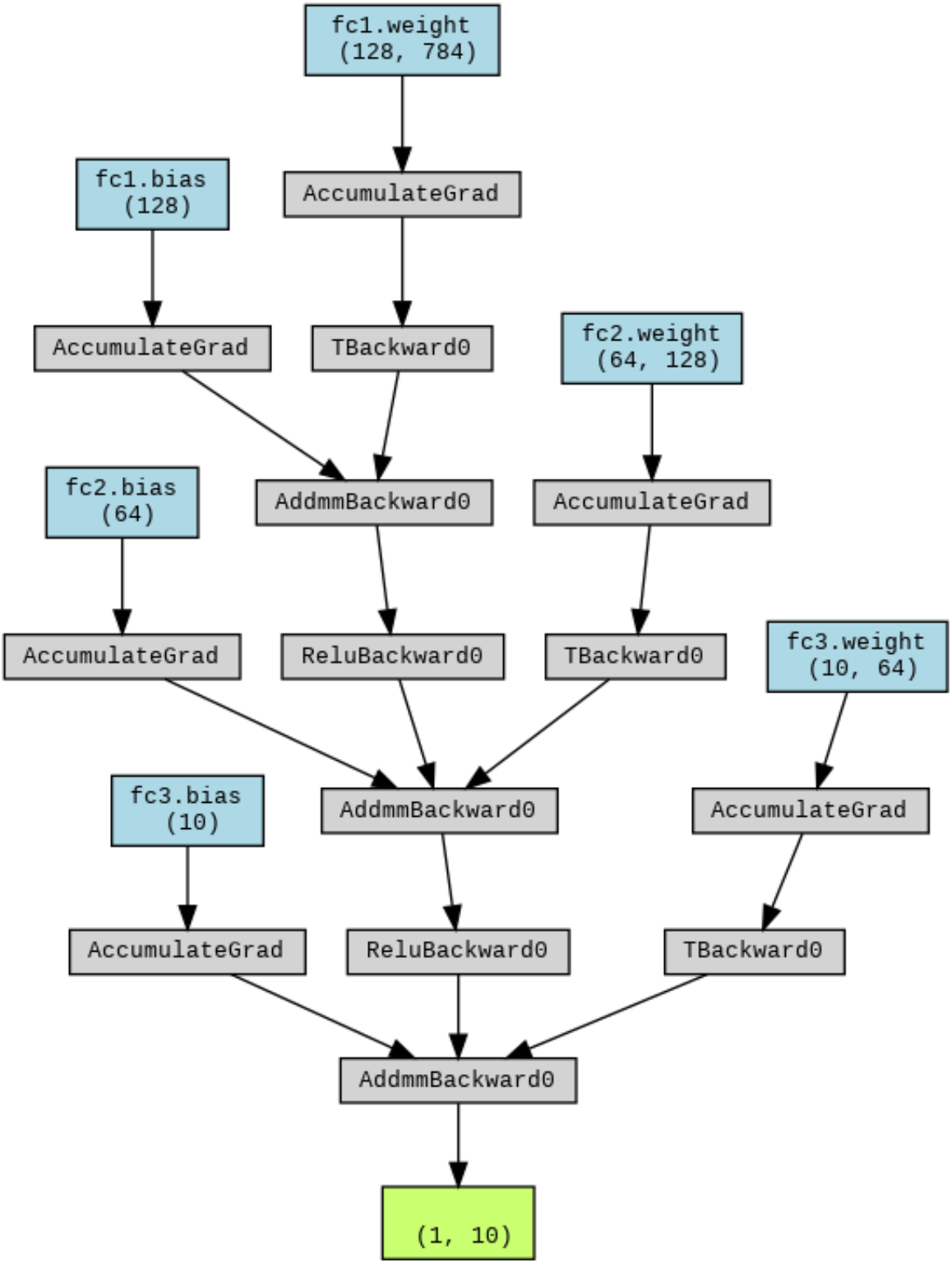
```
SimpleNet(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=10, bias=True)
)
```

We can also use the `torchviz` library to create a helpful visualization:

```
from torchviz import make_dot

# Same size as input data
dummy_input = torch.randn(1, 28, 28).cuda()

graph = make_dot(model(dummy_input), params=dict(model.named_parameters()))
graph.render("Model", format="png", cleanup=True)
```

```
'Model.png'
```

```
from IPython.display import Image, display

# Display the image in the notebook
image_path = "Model.png"
display(Image(filename=image_path))
```


fc1.weight
(128, 784)
fc1.bias
(128)
AccumulateGrad
AccumulateGrad
TBackward0
fc2.weight
(64, 128)
fc2.bias
(64)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
fc3.weight
(10, 64)
fc3.bias
(10)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
AddmmBackward0
(1, 10)

To keep things simple and accessible on a Colab with free resources (the T4 GPU), we will use the canonical MNIST dataset of handwritten digits, 0-9.

Our training loop will run for five epochs and should complete within a few minutes on the T4.

```
# Load and preprocess the MNIST dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.308
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

# Training loop
num_epochs = 5
for epoch in range(num_epochs):  # 5 epochs for demonstration
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data.view(data.size(0), -1))
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
    if (epoch + 1) % 1 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

```
Epoch [1/5], Loss: 0.0020
Epoch [2/5], Loss: 0.0002
Epoch [3/5], Loss: 0.0047
Epoch [4/5], Loss: 0.0026
Epoch [5/5], Loss: 0.0004
```

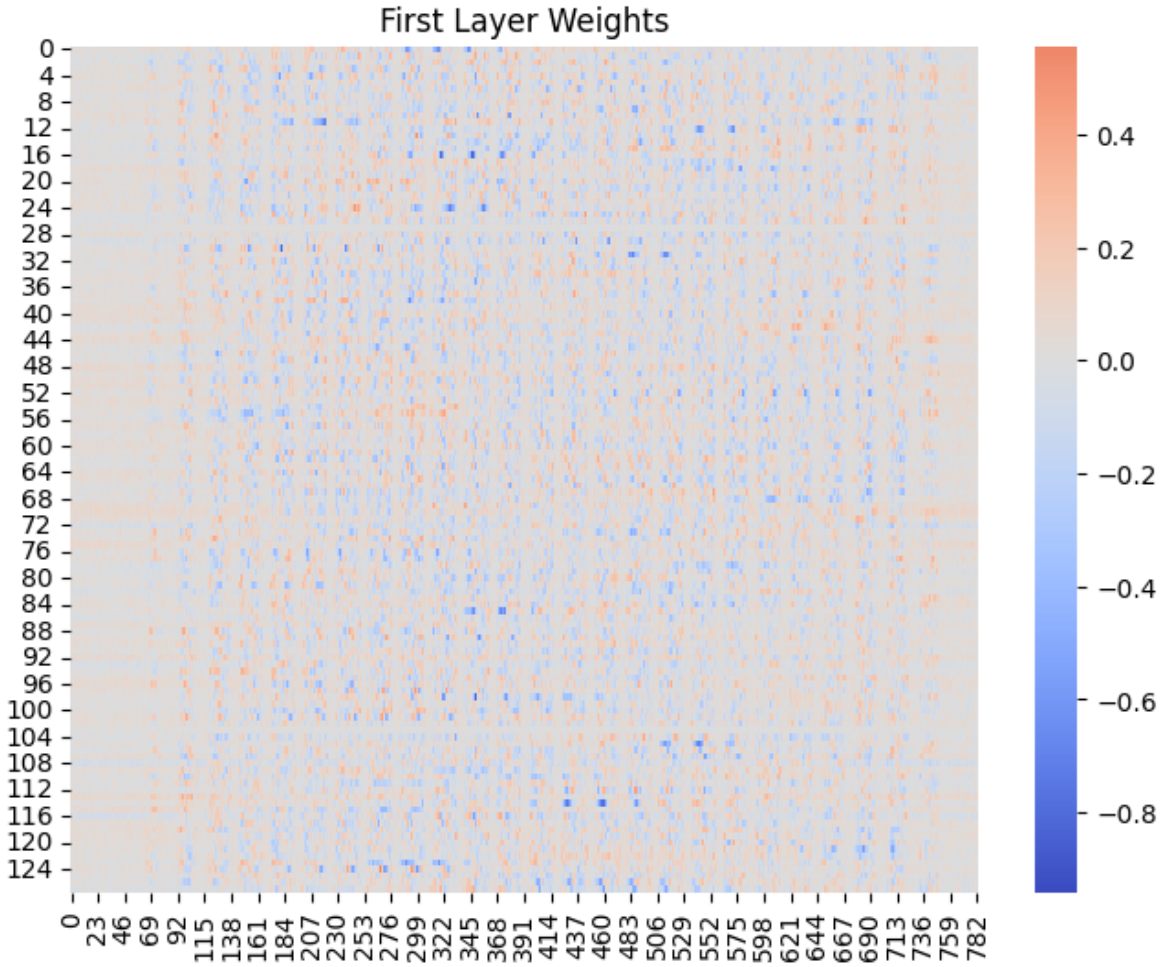
2.0.1 Visualizing Weight Matrices

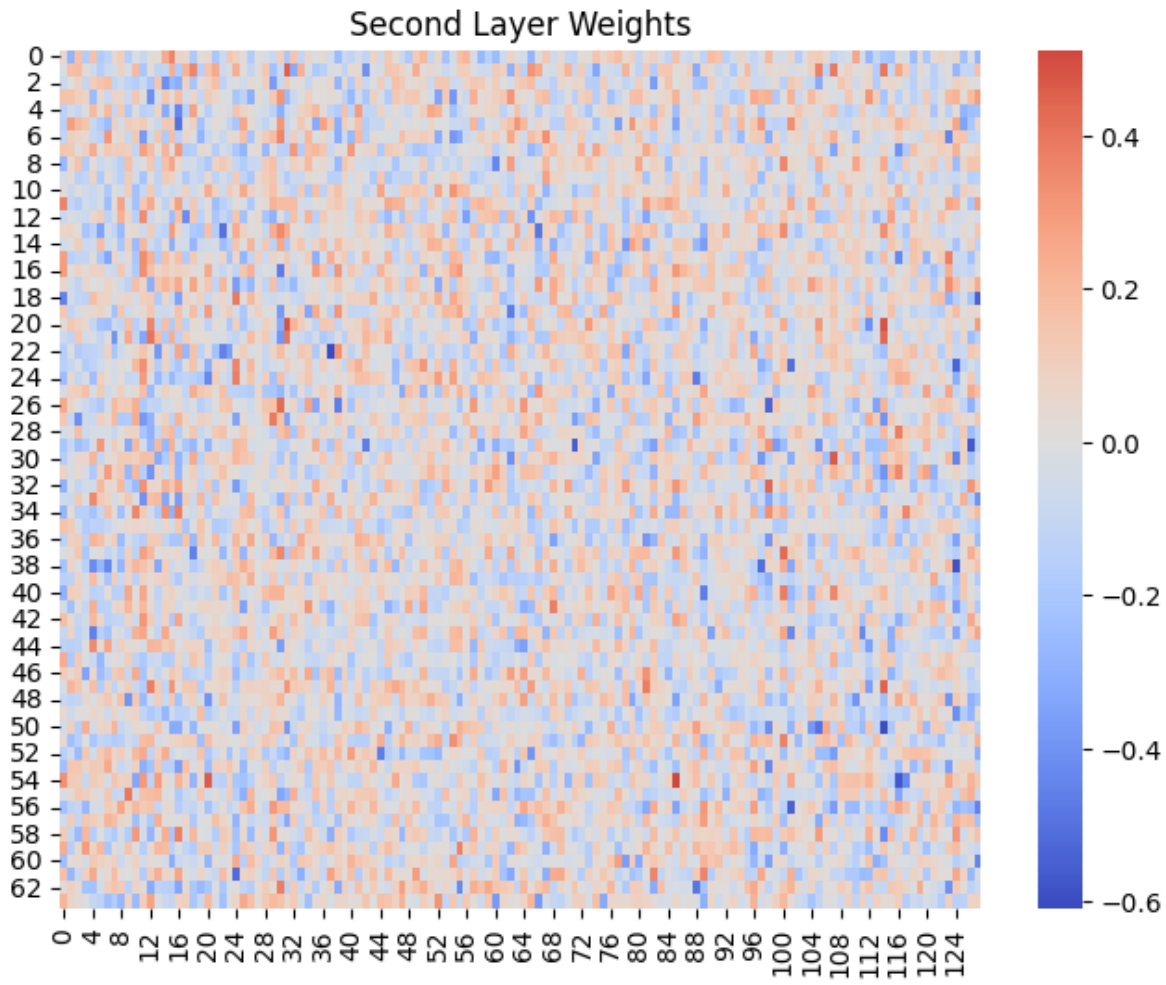
One way to observe superposition is by visualizing the weight matrices of our layers. We can plot these as heatmaps:

In these heatmaps, look for:

Patterns or structure in the weights Areas of high positive or negative values Regions where weights seem to cancel each other out

```python
import matplotlib.pyplot as plt
import seaborn as sns

def plot_weight_matrix(weight_matrix, title):
    plt.figure(figsize=(8, 6))
    sns.heatmap(weight_matrix.cpu().detach().numpy(), cmap='coolwarm', center=0)
    plt.title(title)
    plt.show()

plot_weight_matrix(model.fc1.weight, "First Layer Weights")
plot_weight_matrix(model.fc2.weight, "Second Layer Weights")
```



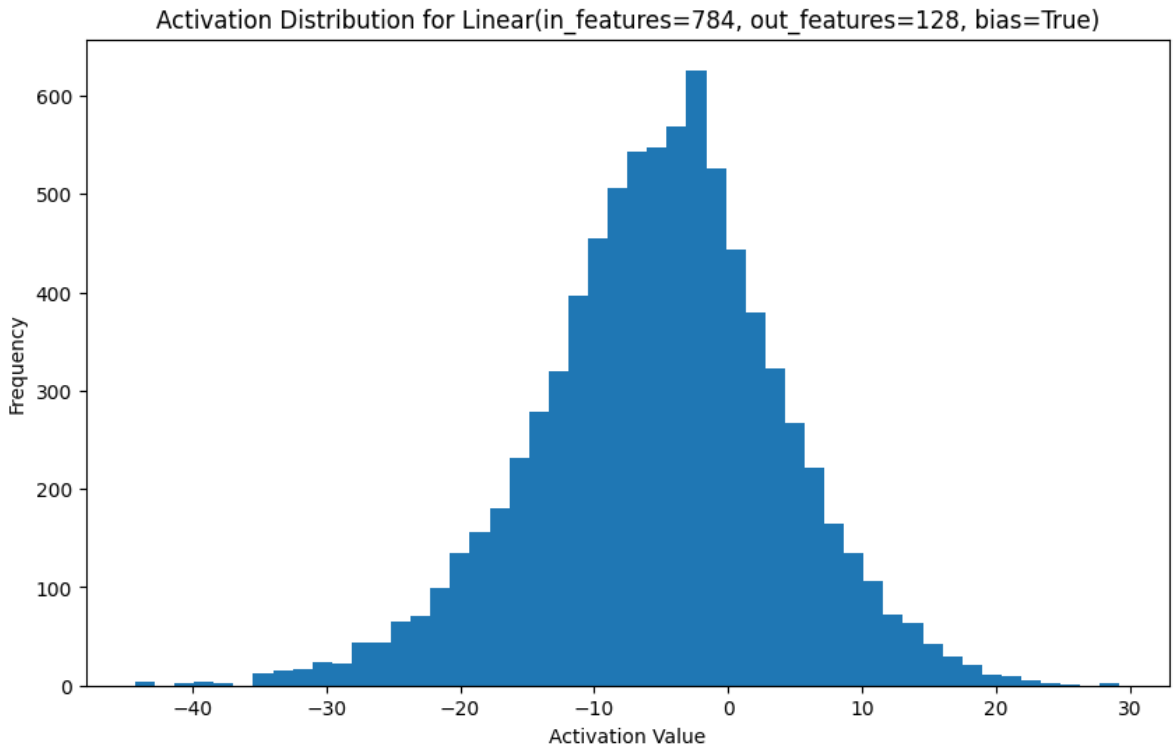


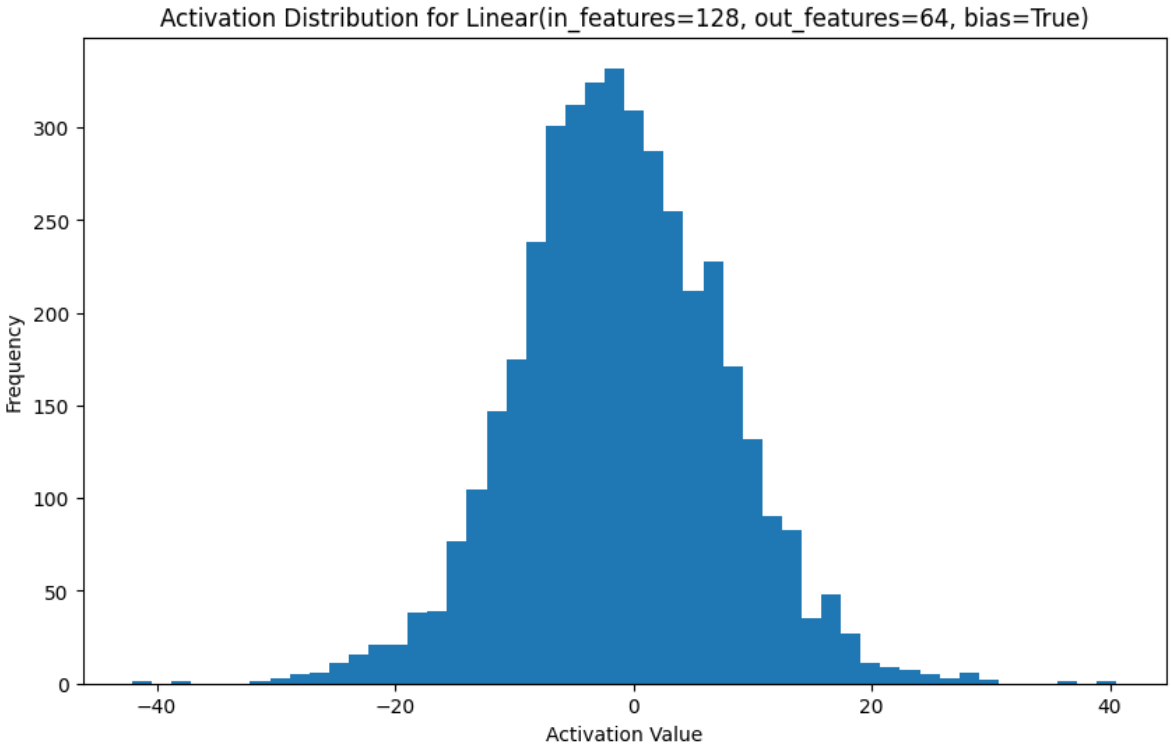
2.0.2 Analyzing Activations

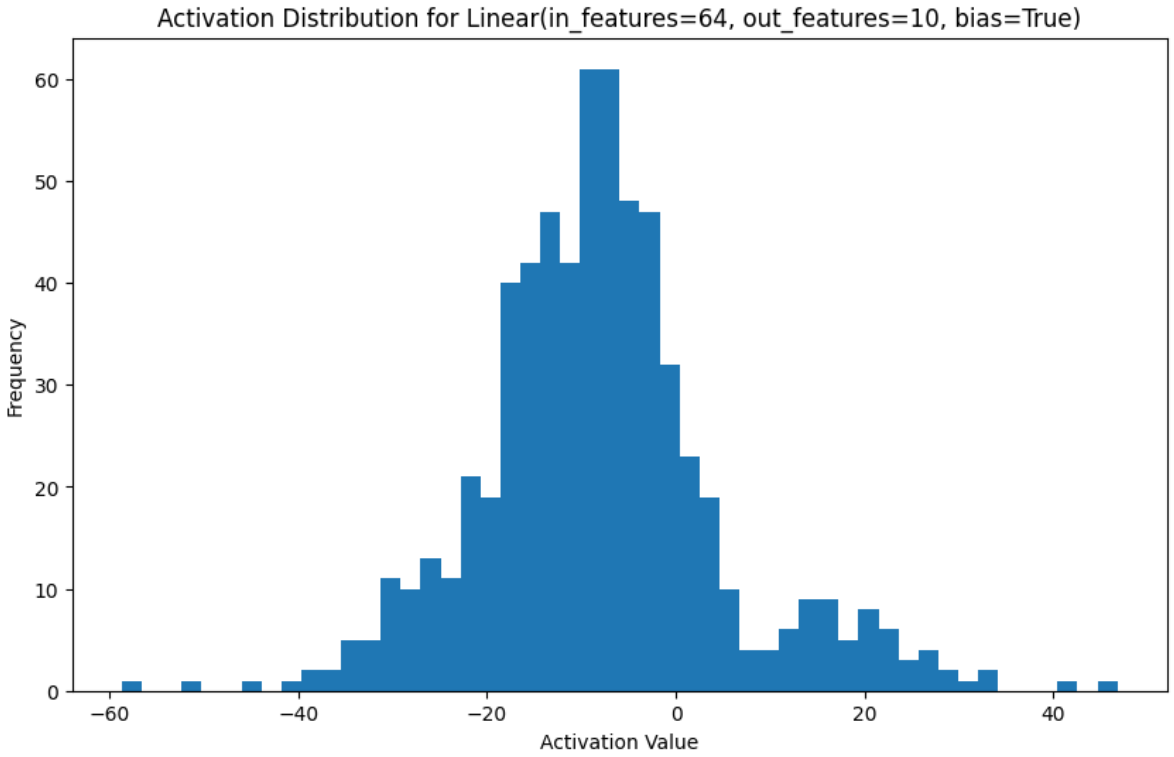
Another approach is to analyze the activations of neurons in response to different inputs:

```
def get_activations(model, input_data):
    activations = {}

    def hook_fn(module, input, output):
        activations[module] = output.detach()

    for name, module in model.named_modules():
        if isinstance(module, nn.Linear):
            module.register_forward_hook(hook_fn)
```

```
    input_data = input_data.to(device)
    model(input_data)
    return activations

# Get a batch of test data
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(root='./data', train=False, transform=transform),
    batch_size=64, shuffle=True)
test_data, _ = next(iter(test_loader))

activations = get_activations(model, test_data.view(test_data.size(0), -1))

# Plot activation distributions
for name, activation in activations.items():
    plt.figure(figsize=(10, 6))
    plt.hist(activation.cpu().numpy().flatten(), bins=50)
    plt.title(f"Activation Distribution for {name}")
    plt.xlabel("Activation Value")
    plt.ylabel("Frequency")
    plt.show()
```




Activation Distribution for Linear(in_features=128, out_features=64, bias=True)
Frequency
Activation Value
0
50
100
150
200
250
300
−40
−20
0
20
40



2.0.3 Measuring Superposition

To quantify superposition, we can use techniques like Singular Value Decomposition (SVD) on the weight matrices.

SVD is a technique in linear algebra to distill matrices into simpler component matrices.

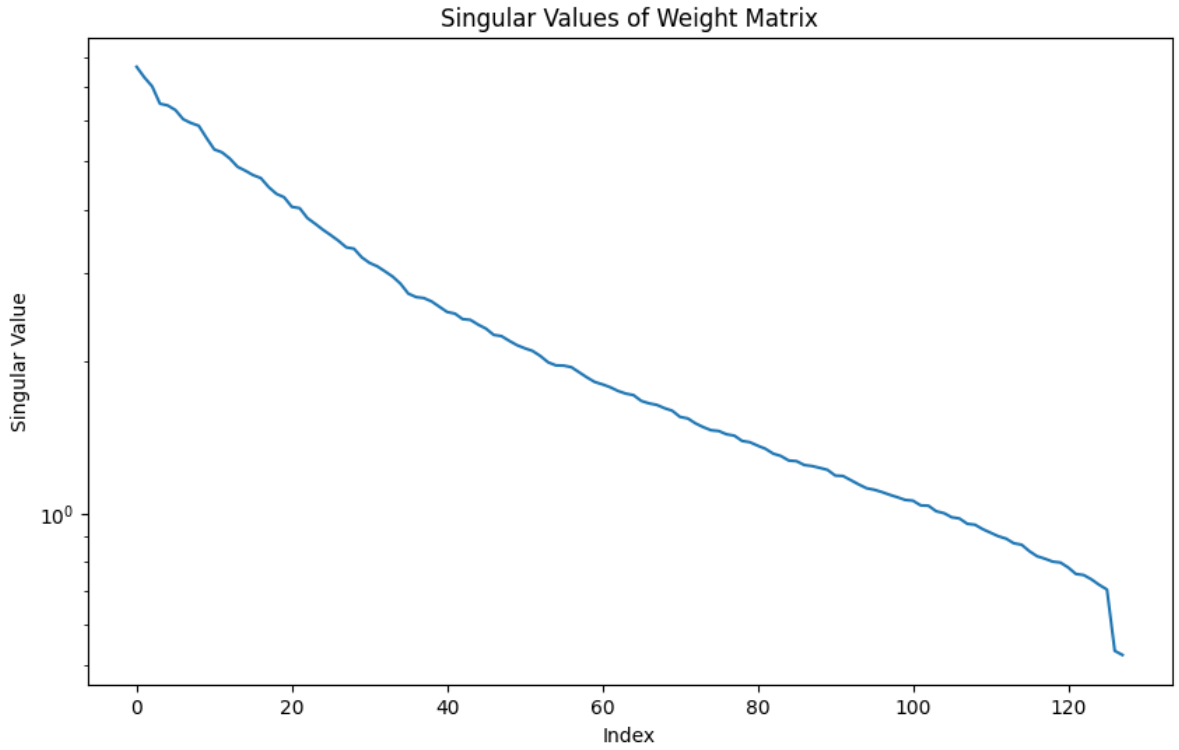
In this example, we take the following steps (also commented in the code):

- Compute total variance, which is the sum of squared singular values
- Calculate the cumulative variance of each singular value

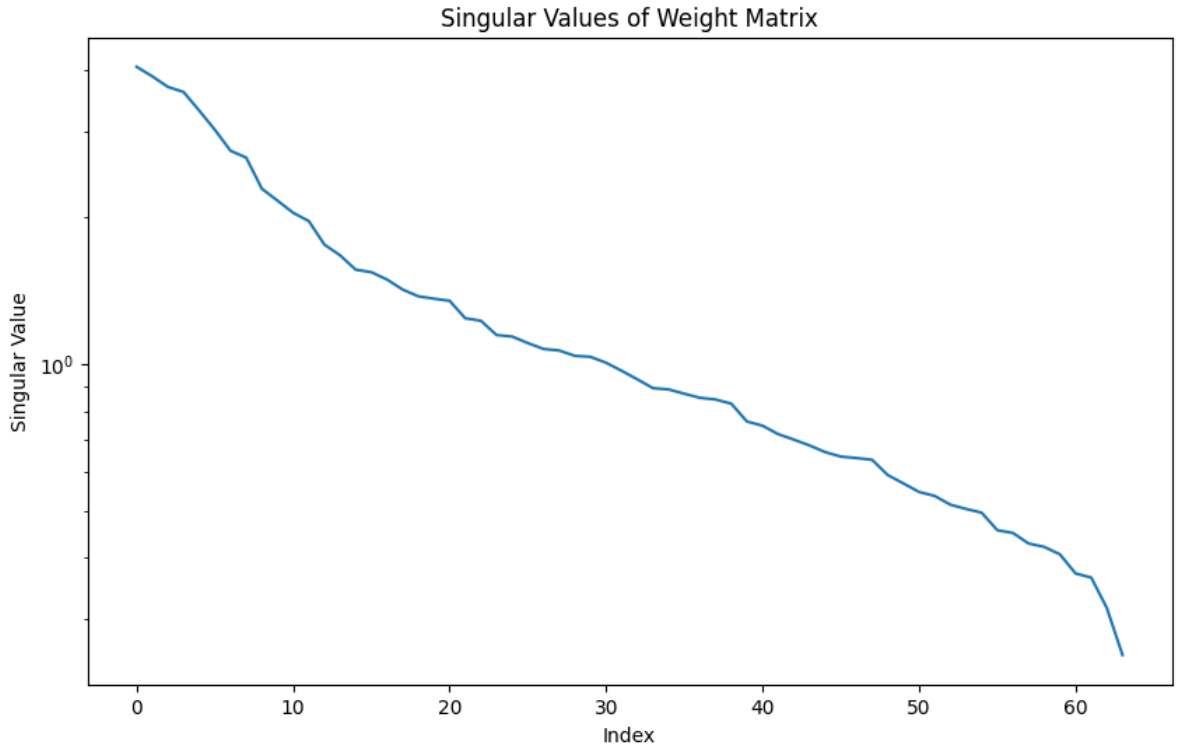
2.0.3.1 Interpretation

The 'effective rank' is a measure of superposition. A lower effective rank indicated less of the phenonemon. A higher effective rank suggests more, implying the weight matrix requires more dimensions to be accurately represented.

```python
import numpy as np

def analyze_superposition(weight_matrix):
    # Here's the SVD calculation. The 'S' array contains the
    # singular values in descending order.
    U, S, Vt = np.linalg.svd(weight_matrix.cpu().detach().numpy())

    # Plot singular values
    plt.figure(figsize=(10, 6))
    plt.plot(S)
    plt.title("Singular Values of Weight Matrix")
    plt.xlabel("Index")
    plt.ylabel("Singular Value")
    plt.yscale('log')
    plt.show()

    # Calculate 'effective rank', which is a measure of superposition.
    # This computes the total variance (sum of squared values), then
    # calculates the cumulative variance explained by each singular value.
    total_variance = np.sum(S**2)
    cumulative_variance = np.cumsum(S**2) / total_variance
    effective_rank = np.sum(cumulative_variance < 0.99)  # 99% of variance

    print(f"Effective Rank: {effective_rank}")

analyze_superposition(model.fc1.weight)
analyze_superposition(model.fc2.weight)
```




```
Effective Rank: 110
```




```
Effective Rank: 54
```

In our SimpleNet model, we defined the following architecture:

```python
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)
```

Let's compare the effective ranks we observed with the actual number of neurons in each layer:

First layer (fc1):

Total neurons: 128 Effective rank: 113 Ratio: 113 / 128 0.883 or 88.3%

Second layer (fc2):

Total neurons: 64 Effective rank: 54 Ratio: 54 / 64 0.844 or 84.4%

Interpretation:

First layer (fc1): The effective rank of 113 compared to 128 total neurons suggests that this layer is using about 88.3% of its capacity for unique features, corresponding to a high degree of superposition. So a large number of singular values are needed to explain the variance in the weight matricies.

Second layer (fc2): The effective rank of 54 vs 64 total neurons indicates that this layer is using about 84.4% of its capacity for unique features, showing a slight decrease that may indicate more specialization or feature abstraction in the second layer.

The effective rank gives us an idea of how many "effective features" the layer is representing. A higher effective rank compared to the actual number of neurons suggests neurons are representing multiple features simultaneously, indicating a higher degree of superposition.

2.0.4 Interpreting the Results

When looking at the results, focus on sparse activation patterns, which might indicate specialized neurons. Compare the number of neurons to the effective rank - a large discrepancy suggests a high degree of superposition. Observe how superposition changes across layers. Consider how different input patterns affect the activations and whether this reveals any superposed features.

2.0.5 More layers, more data: CIFAR 100

Let's explore whether larger datasets and more complex neural network architectures affect the degree of superposition.

We switch to the ResNet50 model, which has 50 layers, including 48 convolutional layers, 1 max pool and 1 average pool layer. It uses skip connections to address the vanishing gradient problem, enabling training of deeper networks.

We will use the CIFAR-100 dataset, which comprises 60,000, 32 x 32 color images in 100 classes.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import models, datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from tqdm import tqdm
```

```
# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Define transforms
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761)),
])

# Load CIFAR-100 dataset
train_dataset = datasets.CIFAR100(root='./data', train=True, download=True, transform=transfo
test_dataset = datasets.CIFAR100(root='./data', train=False, download=True, transform=transfo

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True, num_workers=4, pin_mem
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False, num_workers=4, pin_memo

# Load pre-trained ResNet50 model and modify for CIFAR-100
model = models.resnet50(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, 100)  # 100 classes in CIFAR-100
model = model.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs}"):
        inputs, labels = inputs.to(device), labels.to(device)
```

```
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch {epoch+1} loss: {running_loss/len(train_loader):.4f}")

    # Validation
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in tqdm(test_loader, desc="Validation"):
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

    print(f"Validation Accuracy: {100.*correct/total:.2f}%")

print("Training completed")
```

```
Using device: cuda
Downloading https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz to ./data/cifar-100-pyth

100%|      | 169001437/169001437 [00:04<00:00, 42237449.26it/s]

Extracting ./data/cifar-100-python.tar.gz to ./data
Files already downloaded and verified

/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This
  warnings.warn(_create_warning_msg(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The pa
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Argume
  warnings.warn(msg)
```

```
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /root/.cache/tor
100%|      | 97.8M/97.8M [00:00<00:00, 130MB/s]
Epoch 1/10: 100%|      | 391/391 [00:34<00:00, 11.18it/s]

Epoch 1 loss: 3.3070

Validation: 100%|      | 79/79 [00:03<00:00, 24.33it/s]

Validation Accuracy: 25.89%

Epoch 2/10: 100%|      | 391/391 [00:33<00:00, 11.58it/s]

Epoch 2 loss: 2.5782

Validation: 100%|      | 79/79 [00:03<00:00, 19.93it/s]

Validation Accuracy: 41.69%

Epoch 3/10: 100%|      | 391/391 [00:34<00:00, 11.28it/s]

Epoch 3 loss: 2.1787

Validation: 100%|      | 79/79 [00:03<00:00, 23.63it/s]

Validation Accuracy: 45.72%

Epoch 4/10: 100%|      | 391/391 [00:33<00:00, 11.62it/s]

Epoch 4 loss: 2.0079

Validation: 100%|      | 79/79 [00:03<00:00, 25.49it/s]

Validation Accuracy: 48.87%

Epoch 5/10: 100%|      | 391/391 [00:33<00:00, 11.72it/s]

Epoch 5 loss: 1.7884
```

```
Validation: 100%|      | 79/79 [00:05<00:00, 15.58it/s]

Validation Accuracy: 50.87%

Epoch 6/10: 100%|      | 391/391 [00:33<00:00, 11.65it/s]

Epoch 6 loss: 1.6728

Validation: 100%|      | 79/79 [00:03<00:00, 25.56it/s]

Validation Accuracy: 51.85%

Epoch 7/10: 100%|      | 391/391 [00:33<00:00, 11.60it/s]

Epoch 7 loss: 1.5569

Validation: 100%|      | 79/79 [00:03<00:00, 25.69it/s]

Validation Accuracy: 53.72%

Epoch 8/10: 100%|      | 391/391 [00:34<00:00, 11.41it/s]

Epoch 8 loss: 1.4626

Validation: 100%|      | 79/79 [00:04<00:00, 15.99it/s]

Validation Accuracy: 53.80%

Epoch 9/10: 100%|      | 391/391 [00:34<00:00, 11.32it/s]

Epoch 9 loss: 1.4110

Validation: 100%|      | 79/79 [00:03<00:00, 24.77it/s]

Validation Accuracy: 54.18%

Epoch 10/10: 100%|      | 391/391 [00:33<00:00, 11.73it/s]
```

```
Epoch 10 loss: 1.3250

Validation: 100%|      | 79/79 [00:03<00:00, 24.64it/s]

Validation Accuracy: 55.82%
Training completed
```

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import OrderedDict

def get_activations(model, loader, num_batches=10):
    activations = OrderedDict()

    def hook_fn(name):
        def hook(module, input, output):
            activations[name] = output.cpu().detach()
        return hook

    # Register hooks for convolutional layers
    for name, module in model.named_modules():
        if isinstance(module, nn.Conv2d):
            module.register_forward_hook(hook_fn(name))

    model.eval()
    with torch.no_grad():
        for i, (inputs, _) in enumerate(loader):
            if i >= num_batches:
                break
            inputs = inputs.to(device)
            _ = model(inputs)

    return activations

def analyze_superposition(activation, layer_name):
    reshaped = activation.reshape(activation.shape[1], -1).numpy()
    U, S, Vt = np.linalg.svd(reshaped, full_matrices=False)
```

```
    total_variance = np.sum(S**2)
    cumulative_variance = np.cumsum(S**2) / total_variance
    effective_rank = np.sum(cumulative_variance < 0.99)  # 99% of variance

    return {
        'layer_name': layer_name,
        'total_channels': activation.shape[1],
        'effective_rank': effective_rank,
        'ratio': effective_rank / activation.shape[1]
    }

# Get activations
activations = get_activations(model, test_loader)

# Analyze superposition for each layer
results = []
for name, activation in activations.items():
    results.append(analyze_superposition(activation, name))

# Create DataFrame
df = pd.DataFrame(results)

# Plot effective rank vs layer
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x=df.index, y='effective_rank', marker='o')
plt.title('Effective Rank vs Layer')
plt.xlabel('Layer Index')
plt.ylabel('Effective Rank')
plt.xticks(df.index, df['layer_name'], rotation=45, ha='right')
plt.tight_layout()
plt.savefig('effective_rank_vs_layer.png')
plt.close()

# Plot ratio of effective rank to total channels
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x=df.index, y='ratio', marker='o')
plt.title('Ratio of Effective Rank to Total Channels vs Layer')
plt.xlabel('Layer Index')
plt.ylabel('Ratio')
plt.xticks(df.index, df['layer_name'], rotation=45, ha='right')
plt.tight_layout()
plt.savefig('effective_rank_ratio_vs_layer.png')
```

```
plt.close()

print(df)
print("\nAnalysis completed. Check the generated PNG files for visualizations.")
```

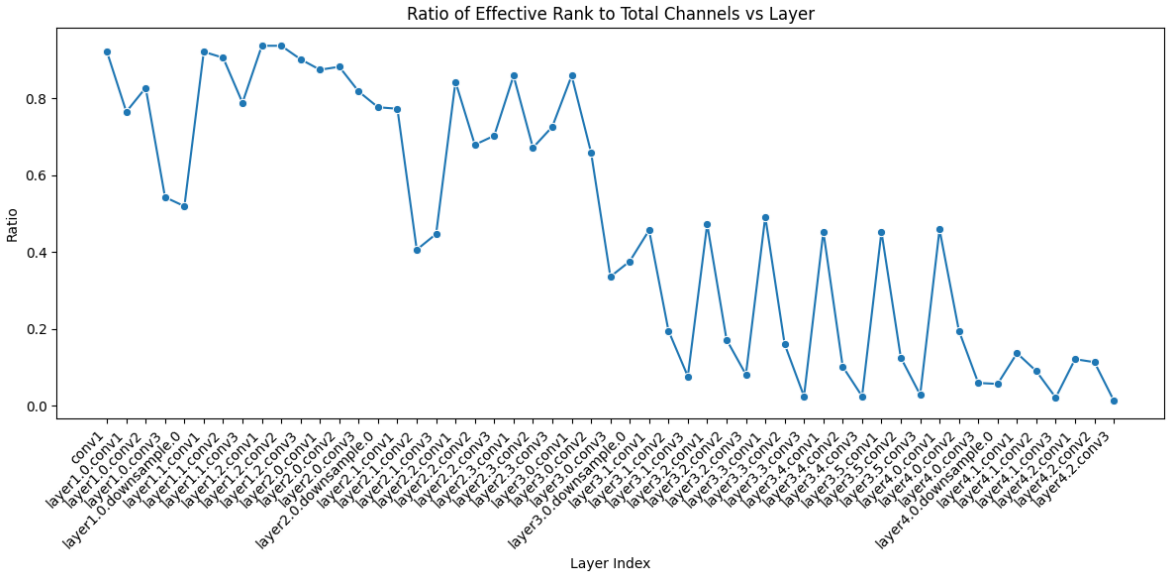
	layer_name	total_channels	effective_rank	ratio
0	conv1	64	59	0.921875
1	layer1.0.conv1	64	49	0.765625
2	layer1.0.conv2	64	53	0.828125
3	layer1.0.conv3	256	139	0.542969
4	layer1.0.downsample.0	256	133	0.519531
5	layer1.1.conv1	64	59	0.921875
6	layer1.1.conv2	64	58	0.906250
7	layer1.1.conv3	256	202	0.789062
8	layer1.2.conv1	64	60	0.937500
9	layer1.2.conv2	64	60	0.937500
10	layer1.2.conv3	256	231	0.902344
11	layer2.0.conv1	128	112	0.875000
12	layer2.0.conv2	128	113	0.882812
13	layer2.0.conv3	512	419	0.818359
14	layer2.0.downsample.0	512	398	0.777344
15	layer2.1.conv1	128	99	0.773438
16	layer2.1.conv2	128	52	0.406250
17	layer2.1.conv3	512	229	0.447266
18	layer2.2.conv1	128	108	0.843750
19	layer2.2.conv2	128	87	0.679688
20	layer2.2.conv3	512	360	0.703125
21	layer2.3.conv1	128	110	0.859375
22	layer2.3.conv2	128	86	0.671875
23	layer2.3.conv3	512	372	0.726562
24	layer3.0.conv1	256	220	0.859375
25	layer3.0.conv2	256	169	0.660156
26	layer3.0.conv3	1024	344	0.335938
27	layer3.0.downsample.0	1024	385	0.375977
28	layer3.1.conv1	256	117	0.457031
29	layer3.1.conv2	256	50	0.195312
30	layer3.1.conv3	1024	78	0.076172
31	layer3.2.conv1	256	121	0.472656
32	layer3.2.conv2	256	44	0.171875
33	layer3.2.conv3	1024	83	0.081055
34	layer3.3.conv1	256	126	0.492188
35	layer3.3.conv2	256	41	0.160156

```
36         layer3.3.conv3           1024              25  0.024414
37         layer3.4.conv1            256             116  0.453125
38         layer3.4.conv2            256              26  0.101562
39         layer3.4.conv3           1024              25  0.024414
40         layer3.5.conv1            256             116  0.453125
41         layer3.5.conv2            256              32  0.125000
42         layer3.5.conv3           1024              29  0.028320
43         layer4.0.conv1            512             235  0.458984
44         layer4.0.conv2            512             100  0.195312
45         layer4.0.conv3           2048             121  0.059082
46  layer4.0.downsample.0           2048             116  0.056641
47         layer4.1.conv1            512              70  0.136719
48         layer4.1.conv2            512              46  0.089844
49         layer4.1.conv3           2048              43  0.020996
50         layer4.2.conv1            512              62  0.121094
51         layer4.2.conv2            512              58  0.113281
52         layer4.2.conv3           2048              27  0.013184

Analysis completed. Check the generated PNG files for visualizations.
```

2.0.6 Interpretation

With some consistency, we see effective rank ratio decrease as the later network layers learn more abstract representations.

```
# Display the image in the notebook
image_path = "/content/effective_rank_ratio_vs_layer.png"
display(Image(filename=image_path))
```


Ratio of Effective Rank to Total Channels vs Layer
Ratio
0.0
0.2
0.4
0.6
0.8
Layer Index
conv1
layer1.0.conv1
layer1.0.conv2
layer1.0.conv3
layer1.0.downsample.0
layer1.1.conv1
layer1.1.conv2
layer1.1.conv3
layer1.2.conv1
layer1.2.conv2
layer1.2.conv3
layer2.0.conv1
layer2.0.conv2
layer2.0.conv3
layer2.0.downsample.0
layer2.1.conv1
layer2.1.conv2
layer2.1.conv3
layer2.2.conv1
layer2.2.conv2
layer2.2.conv3
layer2.3.conv1
layer2.3.conv2
layer2.3.conv3
layer3.0.conv1
layer3.0.conv2
layer3.0.conv3
layer3.0.downsample.0
layer3.1.conv1
layer3.1.conv2
layer3.1.conv3
layer3.2.conv1
layer3.2.conv2
layer3.2.conv3
layer3.3.conv1
layer3.3.conv2
layer3.3.conv3
layer3.4.conv1
layer3.4.conv2
layer3.4.conv3
layer3.5.conv1
layer3.5.conv2
layer3.5.conv3
layer4.0.conv1
layer4.0.conv2
layer4.0.conv3
layer4.0.downsample.0
layer4.1.conv1
layer4.1.conv2
layer4.1.conv3
layer4.2.conv1
layer4.2.conv2
layer4.2.conv3

3 Interpreting Superposition Analysis Results

Let's analyze a result for one single convolutional layer

```
Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False):
Total channels: 512
Effective Rank: 67
Ratio: 0.13
```

1. Layer description: `Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)`
    - This is a 2D convolutional layer
    - It takes 2048 input channels and outputs 512 channels
    - It uses a 1x1 kernel size, which is essentially a pointwise convolution
2. Total channels: 512
    - In a convolutional neural network, each channel can be thought of as a "neuron" or a feature detector
    - So this layer has 512 "neurons" or feature detectors
3. Effective Rank: 67
    - This is significantly lower than the number of channels (512), indicating a small subset of the available dimensions are needed to explain most of the variance in the weight matrix. This suggests low superposition, since the majority of channels are not contributing to complex representations.
4. Ratio: 0.13 (67 / 512)
    - This ratio indicates that only about 13% of the available dimensions are being effectively utilized

In summary, this result shows a lower degree of superposition, with the layer using only about 67 effective dimensions to represent information, despite having 512 channels available.

4 Superposition in language models

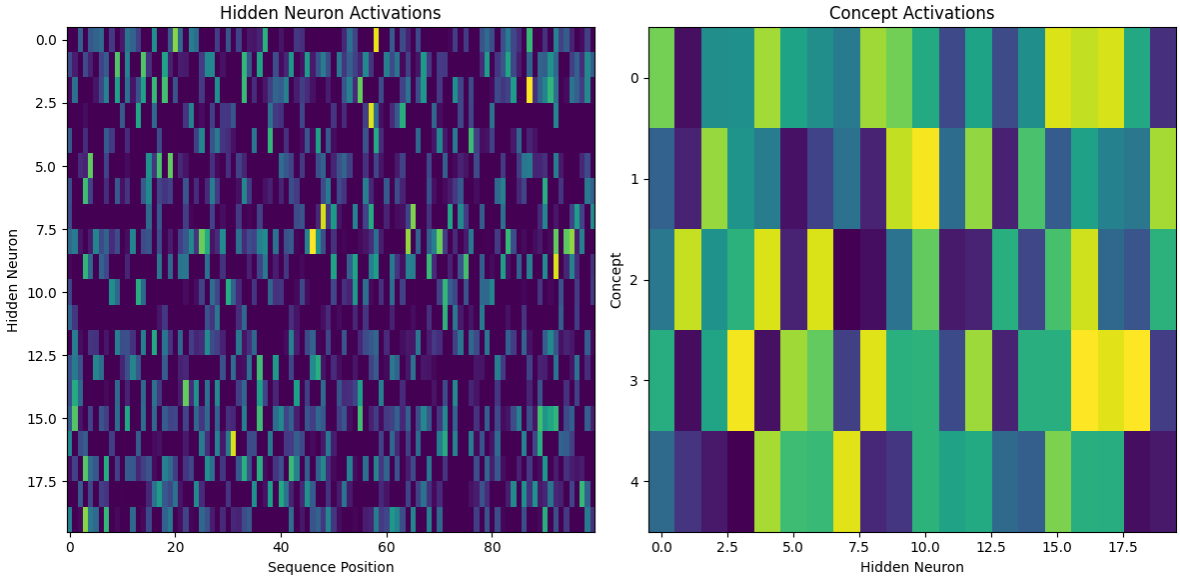
In this section, we will explore how language models encode a variety of concepts in the same sets of neurons.

Let's start with a simple language model. We can use a correlation matrix to see a heatmap of activations for each hidden neuron in the input sequence.

```
!pip install transformers datasets torchviz
```

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

# Set random seed for reproducibility
torch.manual_seed(42)

# Define a simple language model
class SimpleLanguageModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.linear = nn.Linear(embedding_dim, hidden_dim)
        self.output = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x):
        x = self.embedding(x)
        x = torch.relu(self.linear(x))
        return self.output(x)

# Parameters
vocab_size = 1000
embedding_dim = 50
hidden_dim = 20
num_concepts = 5

# Create model
```

```
model = SimpleLanguageModel(vocab_size, embedding_dim, hidden_dim)

# Generate random input
input_ids = torch.randint(0, vocab_size, (100,))

# Get hidden layer activations
with torch.no_grad():
    embeddings = model.embedding(input_ids)
    hidden_activations = torch.relu(model.linear(embeddings))

# Simulate concept activations
concept_activations = torch.rand((num_concepts, hidden_dim))

# Visualize superposition
plt.figure(figsize=(12, 6))

# Plot hidden neuron activations
plt.subplot(1, 2, 1)
plt.imshow(hidden_activations.T, aspect='auto', cmap='viridis')
plt.title('Hidden Neuron Activations')
plt.xlabel('Sequence Position')
plt.ylabel('Hidden Neuron')

# Plot concept activations
plt.subplot(1, 2, 2)
plt.imshow(concept_activations, aspect='auto', cmap='viridis')
plt.title('Concept Activations')
plt.xlabel('Hidden Neuron')
plt.ylabel('Concept')

plt.tight_layout()
plt.show()

# Print correlation matrix
correlation_matrix = torch.corrcoef(torch.cat([hidden_activations.mean(dim=0).unsqueeze(0),
print("Correlation Matrix:")
print(correlation_matrix)
```




```
Correlation Matrix:
tensor([[ 1.0000,  0.2627, -0.0627,  0.0544,  0.1577, -0.2853],
        [ 0.2627,  1.0000,  0.0604, -0.0051,  0.5944,  0.3043],
        [-0.0627,  0.0604,  1.0000,  0.0394,  0.1046, -0.1701],
        [ 0.0544, -0.0051,  0.0394,  1.0000, -0.2142,  0.0359],
        [ 0.1577,  0.5944,  0.1046, -0.2142,  1.0000, -0.1420],
        [-0.2853,  0.3043, -0.1701,  0.0359, -0.1420,  1.0000]])
```

4.0.1 Interpretation

In these visualizations, we see multiple concepts encoded in the same set of neurons.

In the first (left) subplot, we a heatmap of activations for each hidden neuron in the input sequence.

- Each row represents a single neuron in the hidden layer
- Each column represents a position in the input sequence
- The color intensity indicates the activation strength of each neuron at each position, with brighter colors showing higher activation

In the second, *concept activations* visualization, we see a heatmap showing activation patterns for different *concepts* across the hidden neurons.

- Rows represent different concepts
- Columns represent neurons in the hidden layer

- Color intensity indicates how strongly each concept is associated with each hidden neuron
- Brighter colors indicate stronger activations

4.0.2 The key idea

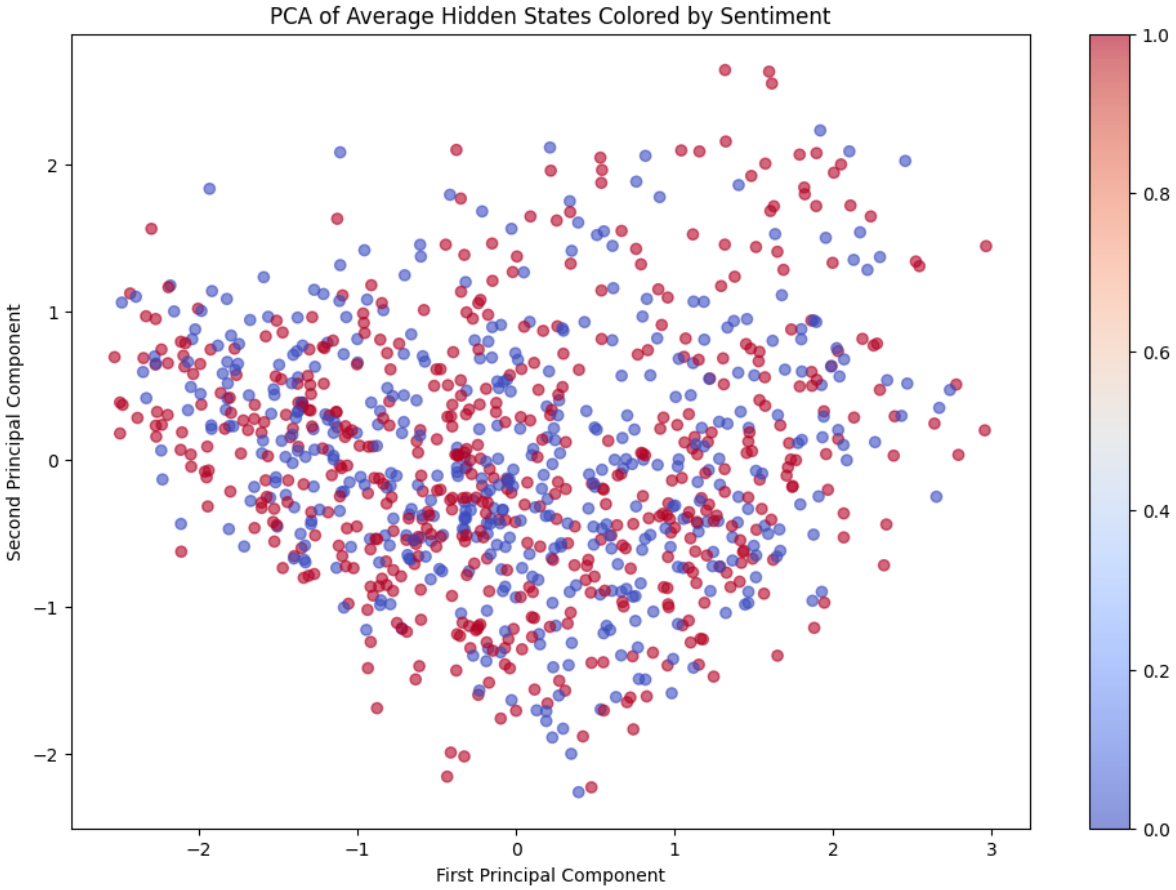
These two visualizations overlap in hidden neuron space, so the same set of neurons are encoding *both* sequence and concept information simultaneously. That these two aspects coexist in the same hidden layer demonstrates superposition.

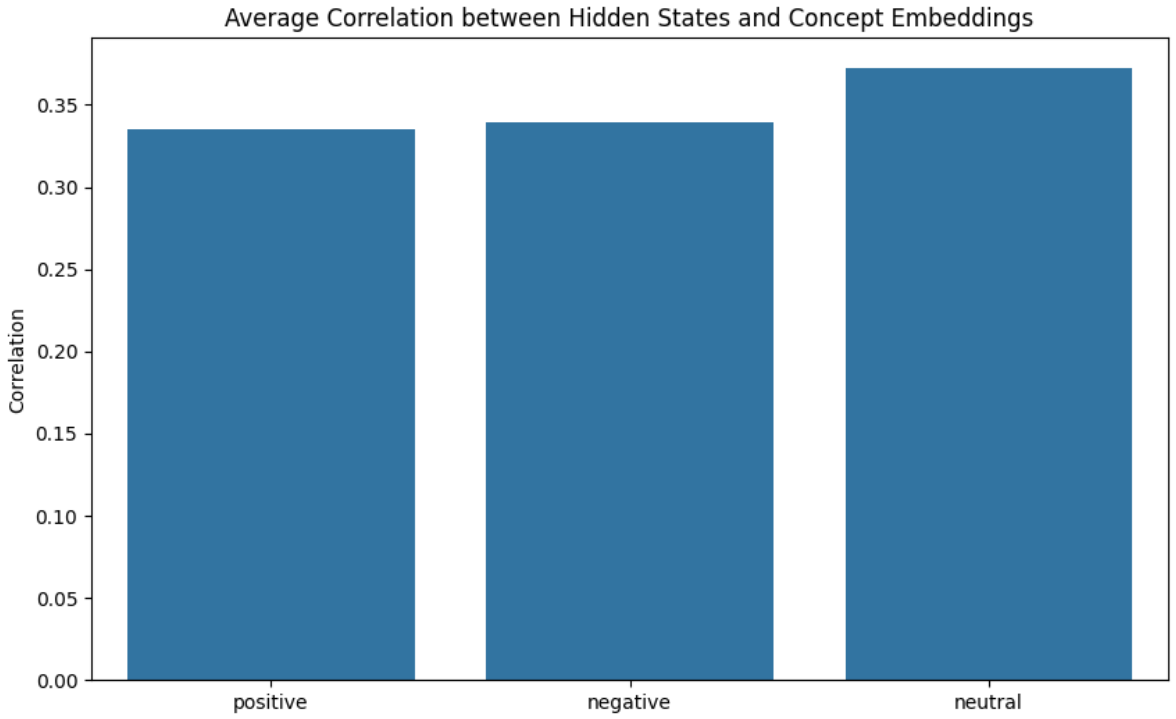
```
import torch
import torch.nn as nn
import numpy as np
from transformers import DistilBertTokenizer, DistilBertModel
from datasets import load_dataset
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA

# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Load pre-trained model and tokenizer
model_name = "distilbert-base-uncased"
tokenizer = DistilBertTokenizer.from_pretrained(model_name)
model = DistilBertModel.from_pretrained(model_name).to(device)

# Load a subset of the GLUE dataset (SST-2 for sentiment analysis)
dataset = load_dataset("glue", "sst2", split="train[:1000]")

# Tokenize the dataset
def tokenize_function(examples):
    return tokenizer(examples["sentence"], padding="max_length", truncation=True, max_length=

tokenized_dataset = dataset.map(tokenize_function, batched=True, remove_columns=dataset.colu
tokenized_dataset.set_format("torch")

# Define linguistic concepts (simple version)
concepts = {
    "positive": ["good", "great", "excellent", "wonderful", "fantastic"],
    "negative": ["bad", "terrible", "awful", "horrible", "poor"],
    "neutral": ["okay", "fine", "average", "mediocre", "so-so"]
```

```
}

# Function to get hidden states
def get_hidden_states(batch):
    inputs = {k: v.to(device) for k, v in batch.items() if k in ['input_ids', 'attention_mask
    with torch.no_grad():
        outputs = model(**inputs)
    return outputs.last_hidden_state.cpu().numpy()

# Get hidden states for the dataset
hidden_states = []
for i in range(0, len(tokenized_dataset), 32):
    batch = tokenized_dataset[i:i+32]
    hidden_states.append(get_hidden_states(batch))

hidden_states = np.concatenate(hidden_states, axis=0)

# Calculate average hidden state for each input
avg_hidden_states = np.mean(hidden_states, axis=1)

# Perform PCA
pca = PCA(n_components=2)
reduced_states = pca.fit_transform(avg_hidden_states)

# Visualize the reduced hidden states
plt.figure(figsize=(12, 8))
scatter = plt.scatter(reduced_states[:, 0], reduced_states[:, 1], c=dataset["label"], cmap="
plt.colorbar(scatter)
plt.title("PCA of Average Hidden States Colored by Sentiment")
plt.xlabel("First Principal Component")
plt.ylabel("Second Principal Component")
plt.show()

# Function to get concept embeddings
def get_concept_embeddings(concepts):
    concept_embeddings = {}
    for concept, words in concepts.items():
        embeddings = []
        for word in words:
            inputs = tokenizer(word, return_tensors="pt").to(device)
            with torch.no_grad():
                outputs = model(**inputs)
```

```
            embeddings.append(outputs.last_hidden_state.mean(dim=1).cpu().numpy())
        concept_embeddings[concept] = np.mean(embeddings, axis=0).flatten()
    return concept_embeddings

concept_embeddings = get_concept_embeddings(concepts)

# Calculate correlation between average hidden states and concept embeddings
correlations = {}
for concept, embedding in concept_embeddings.items():
    corr = np.mean([np.corrcoef(avg_state, embedding)[0, 1] for avg_state in avg_hidden_state
    correlations[concept] = corr

# Visualize correlations
plt.figure(figsize=(10, 6))
sns.barplot(x=list(correlations.keys()), y=list(correlations.values()))
plt.title("Average Correlation between Hidden States and Concept Embeddings")
plt.ylabel("Correlation")
plt.show()

print("Correlations:", correlations)
```

```
Using device: cuda
```


PCA of Average Hidden States Colored by Sentiment
Second Principal Component
First Principal Component
2
1
0
−1
−2
−2
−1
0
1
2
3
1.0
0.8
0.6
0.4
0.2
0.0




```
Correlations: {'positive': 0.33541225356469684, 'negative': 0.3389937145819827, 'neutral': 0
```

4.0.3 Interpretation

In the PCA plot, we see clusters of points with different colors, suggesting the model is distinguishing between different sentiments in its hidden representations.

In the correlation bar plot, the higher values suggest hidden states are more tightly correlated with that particular concept. The high correlations for multiple concepts suggests the model is encoding multiple concepts simultaneously in its hidden states.

5 Interpreting with Sparse Autoencoders

```
!pip install torch transformers
```

In this notebook, we will explore one of the cutting-edge approaches to interpreting superposition: *sparse autoencoders* (SAE).

SAEs are a type of neural network used in unsupervised learning and feature extraction. Autoencoders are generally useful for detecting patterns that are difficult for humans to discern. SAEs have the following characteristics:

- An encoder to compress data and a decoder to recontruct it
- The SAE enforces sparsity in the hidden layer, activating a small number of neurons for a given input
- This sparsity forces the network to learn more efficient and meaningful representations of the input data
- Trained to minimize reconstruction error
- Useful for feature learning, dimensionality reduction, data denoising

Let's import GPT-2 for examination and set up a simple SAE.

```
import torch
import torch.nn as nn
import torch.optim as optim
from transformers import GPT2LMHeadModel, GPT2Tokenizer
import matplotlib.pyplot as plt
import seaborn as sns

# Set up device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load pre-trained GPT-2 model
model_name = 'gpt2'
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
gpt2_model = GPT2LMHeadModel.from_pretrained(model_name).to(device)
```

```python
# Define Sparse Autoencoder
class SparseAutoencoder(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(SparseAutoencoder, self).__init__()
        self.encoder = nn.Linear(input_dim, hidden_dim)
        self.decoder = nn.Linear(hidden_dim, input_dim)

    def forward(self, x):
        encoded = torch.relu(self.encoder(x))
        decoded = self.decoder(encoded)
        return encoded, decoded
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://hugg
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or
  warnings.warn(

tokenizer_config.json:   0%|          | 0.00/26.0 [00:00<?, ?B/s]

vocab.json:   0%|          | 0.00/1.04M [00:00<?, ?B/s]

merges.txt:   0%|          | 0.00/456k [00:00<?, ?B/s]

tokenizer.json:   0%|          | 0.00/1.36M [00:00<?, ?B/s]

config.json:   0%|          | 0.00/665 [00:00<?, ?B/s]

model.safetensors:   0%|          | 0.00/548M [00:00<?, ?B/s]

generation_config.json:   0%|          | 0.00/124 [00:00<?, ?B/s]
```

The following function enables the extraction of activations from GPT-2. The function is liberally commented to explain how to keep track of activations.

Here's how it works: * Tokenize the text * Ensure output is a PyTorch tensor * Return hidden states (activations) from all layers * `output_hidden_states` is a tuple containing the hidden states from all layers by `layer_idx`, which allows us to choose which layer's activations are of interest.

```
# Function to get GPT-2 activations
def get_gpt2_activations(text, layer_idx=-1):
    # Tokenize input text; ensure output is a PyTorch (".pt") tensor
    inputs = tokenizer(text, return_tensors="pt").to(device)
    # Model inference, without tracking gradients. This saves memory,
    # since we are only interested in inference, not training
    with torch.no_grad():
        # The output_hidden_states method returns the hidden states,
        # aka activations, from all layers, not just the final output
        outputs = gpt2_model(**inputs, output_hidden_states=True)
    # outputs.hidden_states is a tuple containing hidden states from
    # all layers of the model. squeeze() removes extra dimensions
    # of size 1 from the tensor.
    return outputs.hidden_states[layer_idx].squeeze()
```

```
def train_sparse_autoencoder(autoencoder, activations, num_epochs=500, sparsity_weight=0.01)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(autoencoder.parameters(), lr=0.001)
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)

    for epoch in range(num_epochs):
        optimizer.zero_grad()
        encoded, decoded = autoencoder(activations)

        recon_loss = criterion(decoded, activations)
        sparsity_loss = torch.mean(torch.abs(encoded))
        loss = recon_loss + sparsity_weight * sparsity_loss

        loss.backward()
        optimizer.step()
        scheduler.step()

        if (epoch + 1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

    return autoencoder
```

The following visualizations should help explain the findings of the sparse autoencoder. One will show the strength of the activations, the other, learned feature importance.

```python
def visualize_features(autoencoder, num_features, save_path='learned_features.png'):
    weights = autoencoder.encoder.weight.data.cpu().numpy()
    fig, axes = plt.subplots(4, 4, figsize=(15, 15))
    for i in range(num_features):
        ax = axes[i // 4, i % 4]
        sns.heatmap(weights[i].reshape(1, -1), ax=ax, cmap='viridis', cbar=False)
        ax.set_title(f'Feature {i+1}')
        ax.axis('off')
    plt.tight_layout()
    plt.savefig(save_path)
    plt.close()

def visualize_activation_strengths(encoded, num_features, save_path='activation_strengths.png
    plt.figure(figsize=(12, 6))
    plt.bar(range(num_features), encoded[:num_features])
    plt.title('Activation Strengths of Learned Features')
    plt.xlabel('Feature Index')
    plt.ylabel('Activation Strength')
    plt.xticks(range(0, num_features, 2))  # Label every other feature for readability
    plt.tight_layout()
    plt.savefig(save_path)
    plt.close()

def analyze_superposition(text, hidden_dim=16):
    # Get GPT-2 activations
    activations = get_gpt2_activations(text).to(device)
    input_dim = activations.shape[-1]

    # Initialize and train sparse autoencoder
    autoencoder = SparseAutoencoder(input_dim, hidden_dim).to(device)
    trained_autoencoder = train_sparse_autoencoder(autoencoder, activations)

    # Visualize learned features
    visualize_features(trained_autoencoder, hidden_dim)

    # Analyze activations
    encoded, _ = trained_autoencoder(activations)
    encoded = encoded.mean(dim=0).squeeze().cpu().detach().numpy()

    # Plot activation strengths
    visualize_activation_strengths(encoded, hidden_dim)
```

```
    print(f"Analysis complete. Check 'learned_features.png' and 'activation_strengths.png' fo

# Run the analysis with multiple inputs
texts = [
    "It was the best of times, it was the worst of times.",
    # "To be or not to be, that is the question.",
    # "In a hole in the ground there lived a hobbit.",
    # "It was the best of times, it was the worst of times."
]

for i, text in enumerate(texts):
    print(f"\nAnalyzing text {i+1}: '{text}'")
    analyze_superposition(text, hidden_dim=16)
```

```
Analyzing text 1: 'It was the best of times, it was the worst of times.'
Epoch [100/500], Loss: 10.9564
Epoch [200/500], Loss: 2.0957
Epoch [300/500], Loss: 1.4265
Epoch [400/500], Loss: 1.3351
Epoch [500/500], Loss: 1.3095
Analysis complete. Check 'learned_features.png' and 'activation_strengths.png' for visualizat
```

5.0.1 Learned feature importance

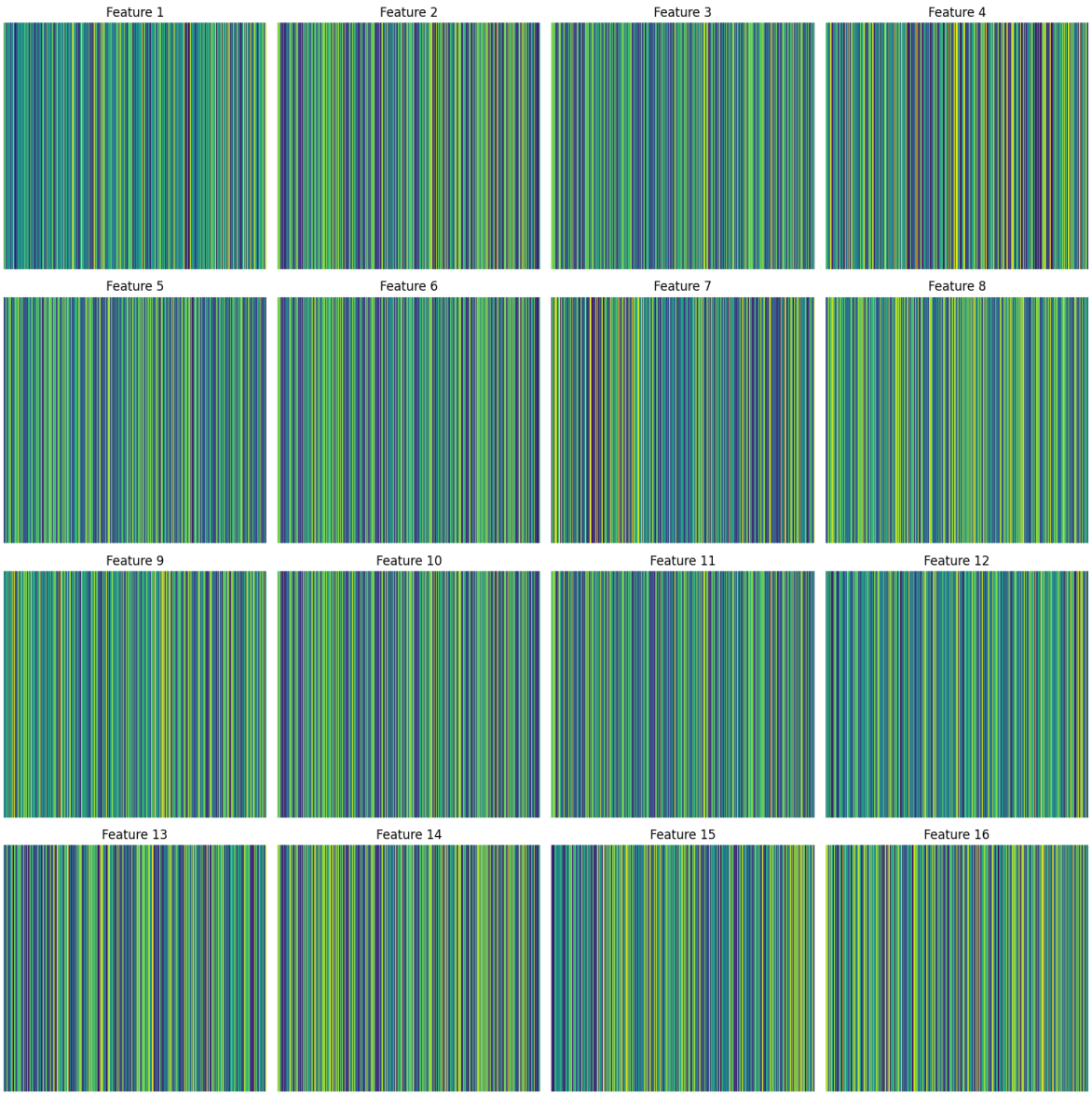
What do we mean by 'features' in this example?

Models such as GPT-2 create a distributed representation of words and concepts. Information about a single word is spread across many dimensions of the model's internal representations, and each dimension contributes to the representation of many different words or concepts.

GPT-2, like many transformer models, uses contextual embeddings, so the representation of a word will change based on its context, eg the word 'bark' in sentences such as 'A dog's bark' and 'The bark on the tree' will have different activations.

The sparse autoeocnder is trying to find efficient ways to encode the patterns in the activations, rather than to isolate individual words.

```
from IPython.display import Image, display

# Display the image in the notebook
image_path = "learned_features.png"
display(Image(filename=image_path))
```



5.0.2 Activation strengths

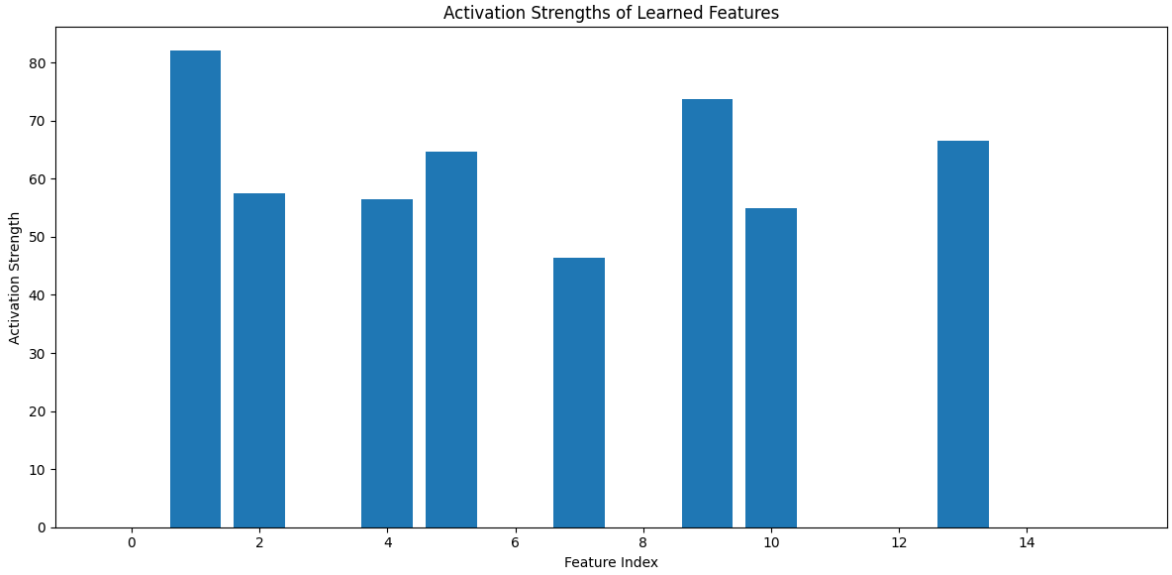
In this visualization, high activations for some features suggest those features are particularly relevant to representing the input sentence. The pattern of activations across all features shows how the model is representing the entire input.

Each bar respresents one of the features learned by the SAE, and the height indicates how strongly the feature was activated for the input text.

Look for sparsity: a successful sparse representation will have many features with low activation (shorter bars), only a few will have high activations (tall bars). The overall distribution can give clues about how the autoencoder is representing the information.

In the GPT-2 model, the input text is distributed across all dimensions of its activation vector, so the SAE tries to represent the same information with fewer specific features.

```
from IPython.display import Image, display

# Display the image in the notebook
image_path = "activation_strengths.png"
display(Image(filename=image_path))
```



In considering these two visualizations together, try to correlate the strong activations in the activation strengths with feature patterns in the heatmap.

Look for groups of features that seem to have similar patterns in the heatmap or similar activation levels, which many indicate related aspects of language that GPT-2 represents.

5.0.3 Summary

This example has shown how in the `learned_features` heatmap how information comprising different linguistic contexts (syntax, semantics etc) is superposed across the same set of neurons in the original model.

In the `activation_strengths` plot, we see how these disentangled features are used to respresent a specific input, with varying levels of activation indicating extracted feature relevance.

The activation of multiple features simultaneously shows how GPT-2 superposes different pieces of information in its dense representation, which the SAE has decomposed into more interpretable units.|

6 Summary

In summary, we covered the topic of superposition, its presence in toy models and in more sophisticated convolution neural networks and language models. We then explored using sparse autoencoders to interpret the features learned by the neurons in a network.

Further resources:

Coming soon