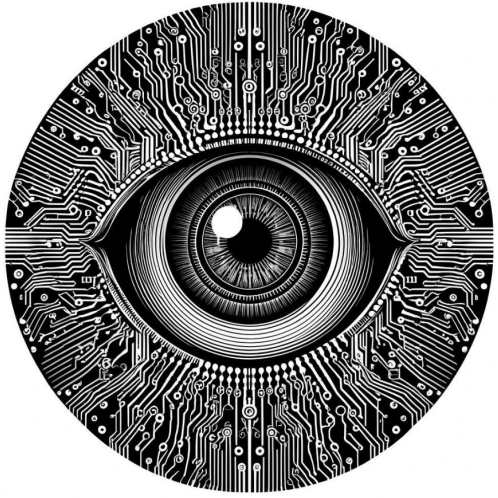
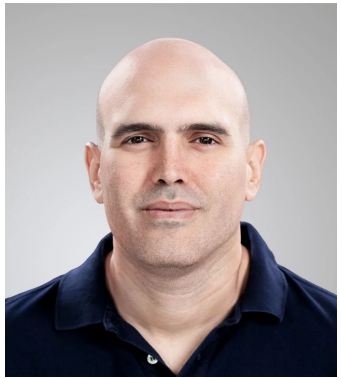
Workshop 5 - Training Techniques for Convolutional Networks



Antonio Rueda-Toicen

About me

- AI Researcher at Hasso Plattner Institute, AI Engineer & DevRel for Voxel51
- Organizer of the Berlin Computer Vision Group
- Instructor at Nvidia's Deep Learning Institute and Berlin's Data Science Retreat
- Preparing a MOOC for OpenHPI (to be published in May)



LinkedIn

Agenda

- Normalizing input values and inference with pretrained models
- Regularization with dropout and batch normalization
- Skip connections in neural networks

Notebooks

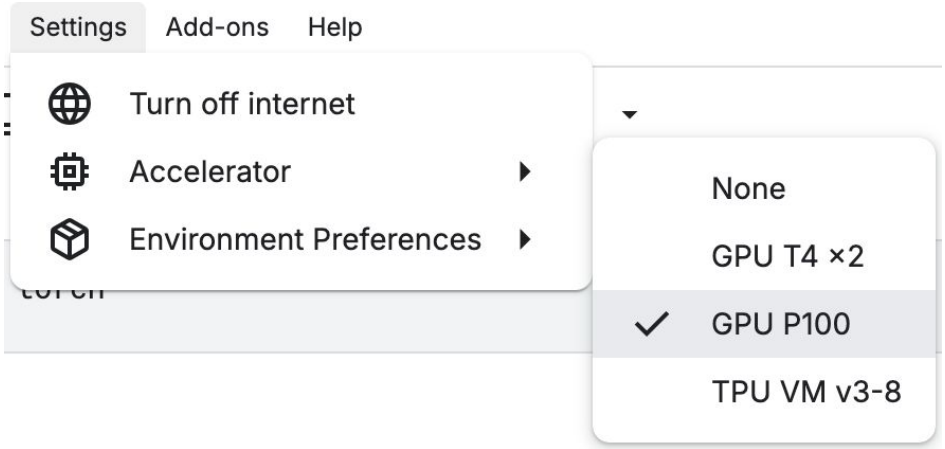
- Fine tuning and comparing Resnet18 and VGG19 on a classification task (fruits) (Kaggle notebook)
- Inspecting the fruits dataset on FiftyOne (Google Colab)

Setup for today

- Setting up a Weights and Biases API token
- Setting up free GPU and TPU access in Kaggle Notebooks

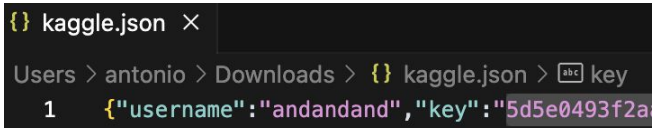
Kaggle Runtime Setup for Training

- Follow the Kaggle GPU Access Guide on our repository
- Settings -> Accelerator -> GPU P100



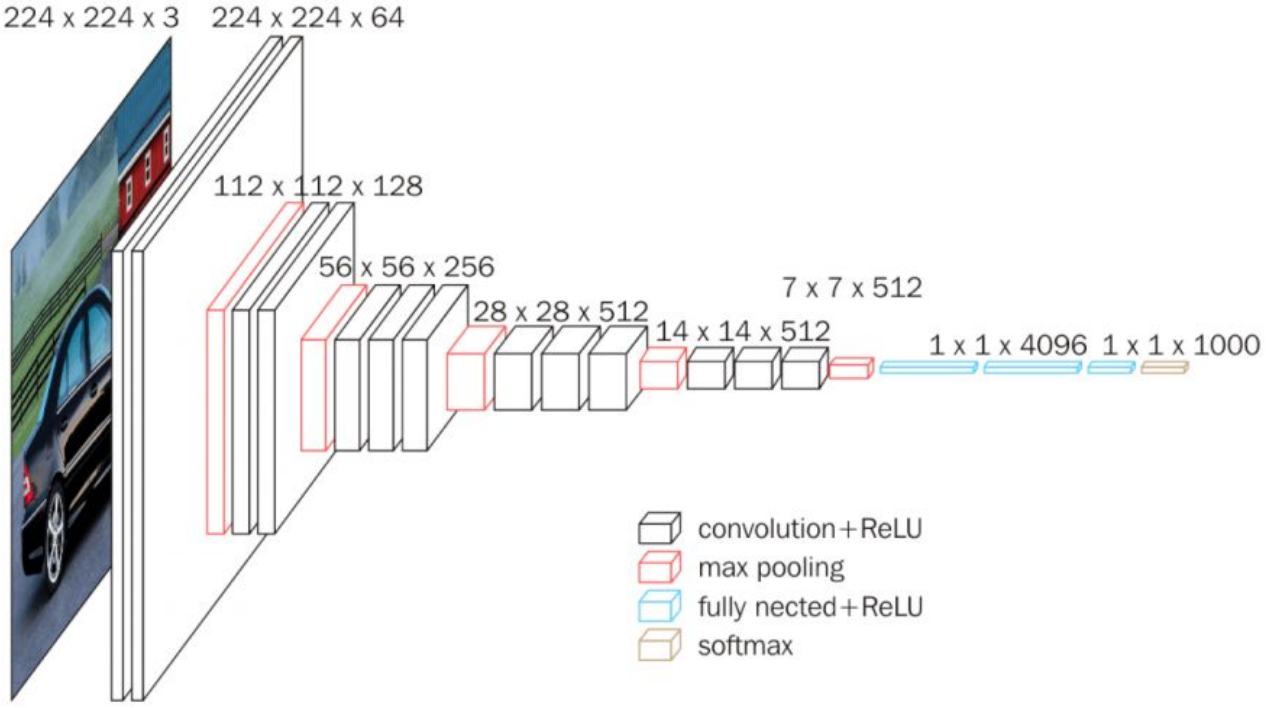
Google Colab Setup for Kaggle API keys

- Go to https://www.kaggle.com/settings and “Create New Token” on your API key
- Download and open the kaggle.json file



- In Google Colab, go to Secrets -> “Add New Secret”
- Create and enable KAGGLE_USERNAME and KAGGLE_KEY variables with the content of the JSON file

VGG-19



Architecture from [Very Deep Convolutional Networks for Large-Scale Image Recognition]

Residual block in Resnet18

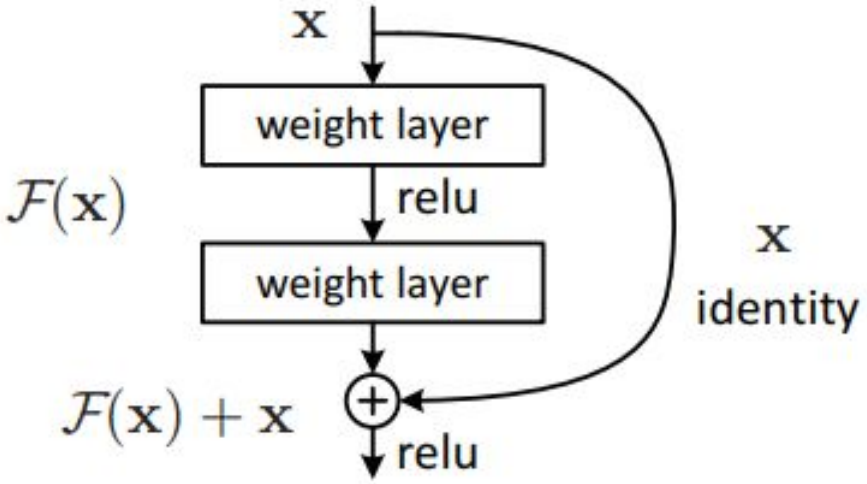
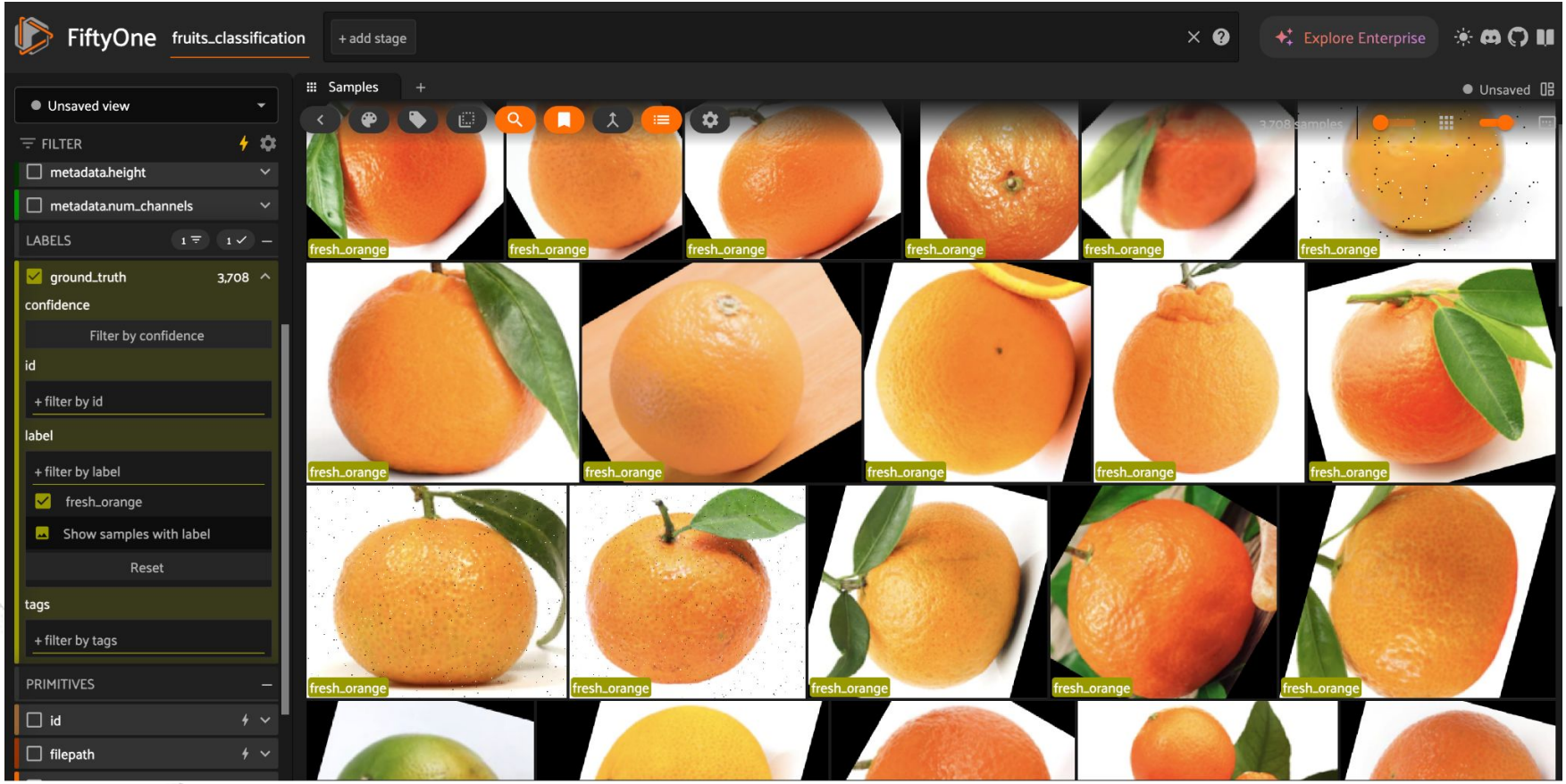
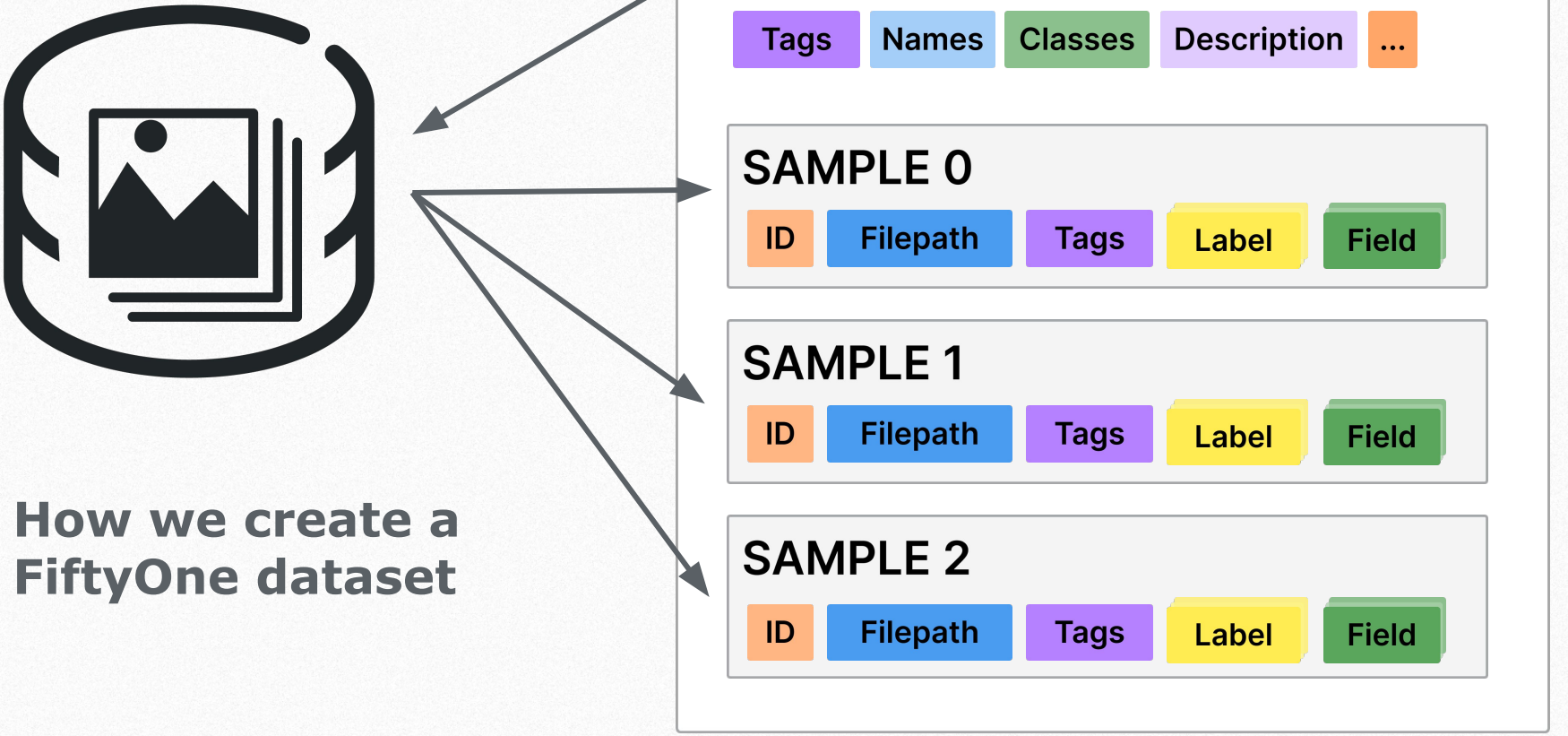


Diagram from [Deep Residual Learning for Image Recognition]

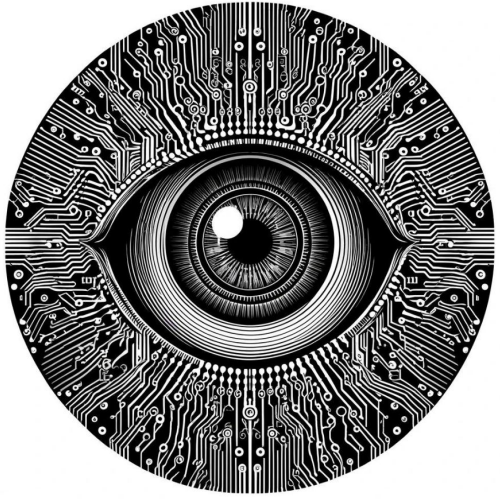


[Colab notebook]



How we create a FiftyOne dataset

Normalizing Input Values and Inference with Pretrained Models

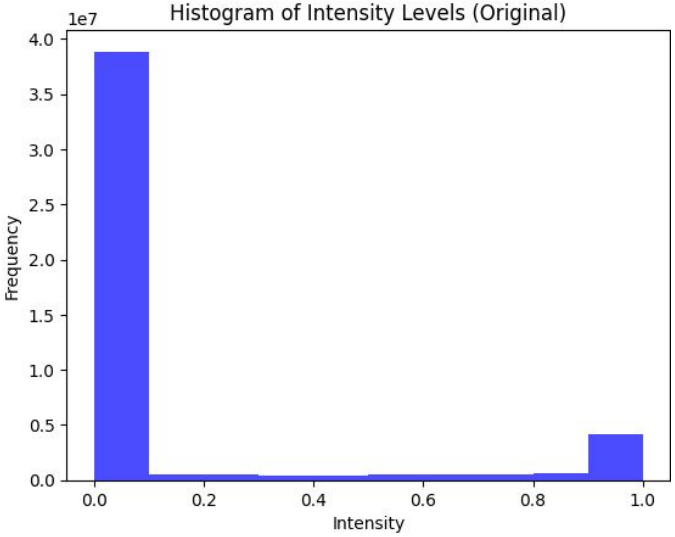


Learning goals

- Evaluate the effect of normalizing input values using statistics from the training set
- Describe effect of input normalization on the output of models pretrained on the Imagenet dataset

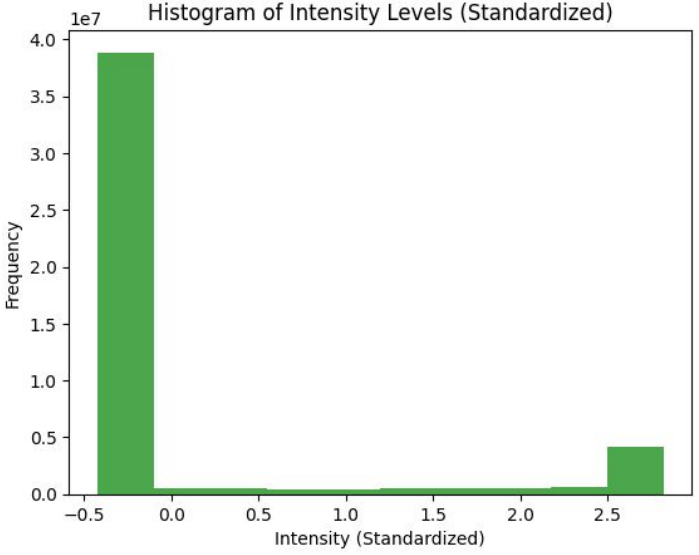
What is input scaling?

- Transforming input values to a range different than the original while preserving the shape of the original distribution
- The standard scaler centers the distribution on zero while making the variance equal to one



$$\mu = 0.1307, \quad \sigma = 0.3081$$

$$z_i = \frac{x_i - \mu}{\sigma}$$



$$\mu = 0.0, \quad \sigma = 1.0$$

Why do we do scaling?

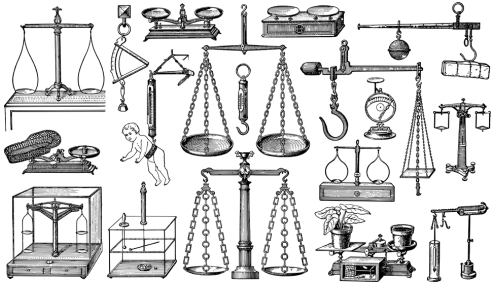
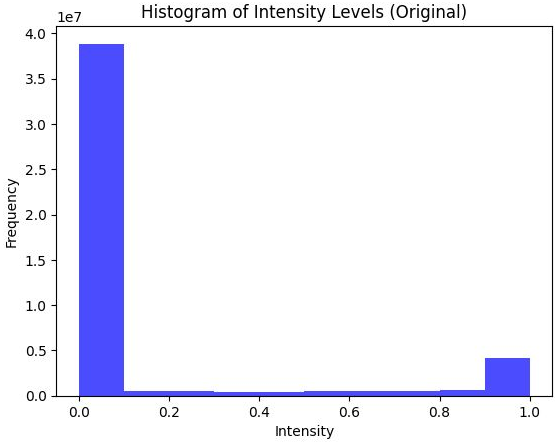
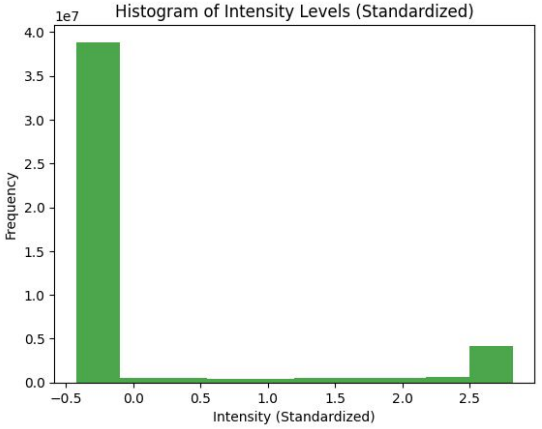


Image from Pexels

1. Has been found in experiments to speed up training.
2. We benefit from using it whenever we use a pretrained model that used it during training.



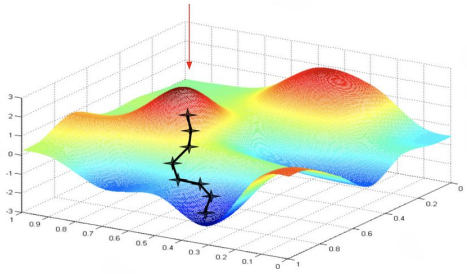
$$z_i = \frac{x_i - \mu}{\sigma}$$



4.3 Normalizing the Inputs

Convergence is usually faster if the average of each input variable over the training set is close to zero. To see this, consider the extreme case where all the inputs are positive. Weights to a particular node in the first weight layer are updated by an amount proportional to δx where δ is the (scalar) error at that node and x is the input vector (see equations (5) and (10)). When all of the components of an input vector are positive, all of the updates of weights that feed into a node will be the same sign (i.e. sign(δ)). As a result, these weights can only all decrease or all increase *together* for a given input pattern. Thus, if a weight vector must change direction it can only do so by zigzagging which is inefficient and thus very slow.

In the above example, the inputs were all positive. However, in general, any shift of the average input away from zero will bias the updates in a particular direction and thus slow down learning. Therefore, it is good to shift the inputs so that the average over the training set is close to zero. This heuristic should be applied at all layers which means that we want the average of the *outputs* of a node to be close to zero because these outputs are the inputs to the next layer [19]. This problem can be addressed by coordinating how the inputs are



Excerpt from '[Efficient Backprop: Tricks of the Trade](#)' by LeCun, Bottou, Orr, and Müller (1998)

The standard scaler for normalization

Mean (μ):

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Standardized Value (z_i):

$$z_i = \frac{x_i - \mu}{\sigma}$$

Standard Deviation (σ):

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

```
computed_mean = torch.mean(train_dataset_elements)
computed_std  = torch.std(train_dataset_elements)

normalization_transform = transforms.Normalize(
                            (computed_mean),
                             (computed_std)
                          )
```

The Imagenet dataset

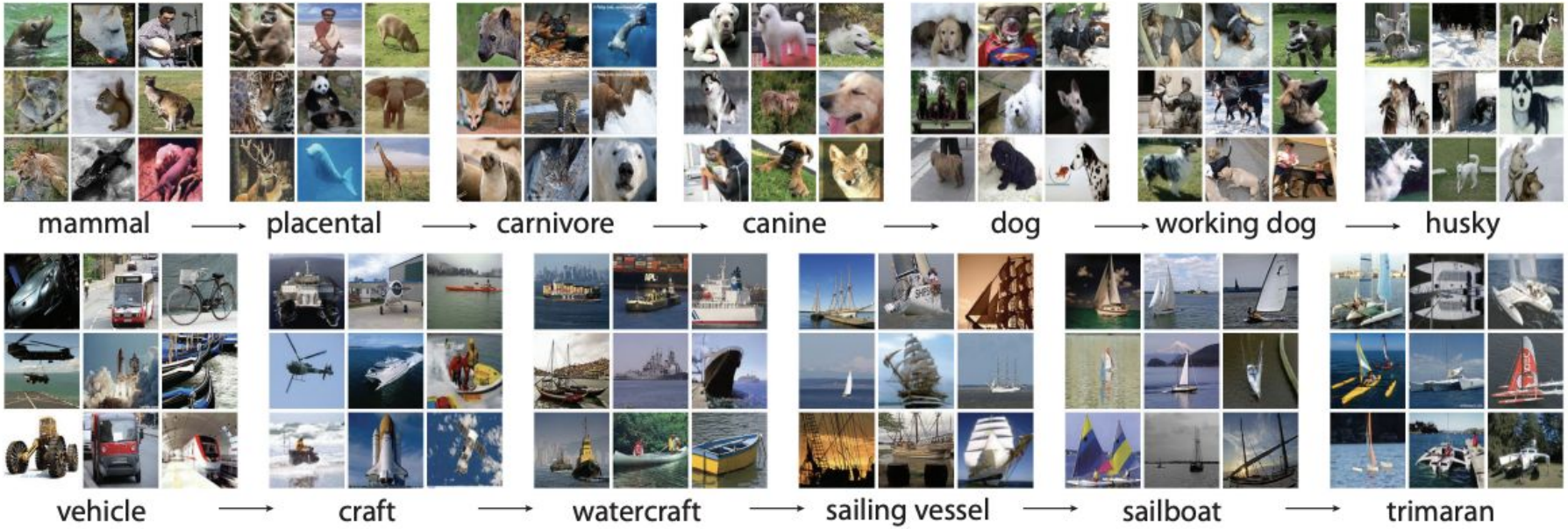
14,197,122 images, 21841 synsets indexed

Home Download Challenges About

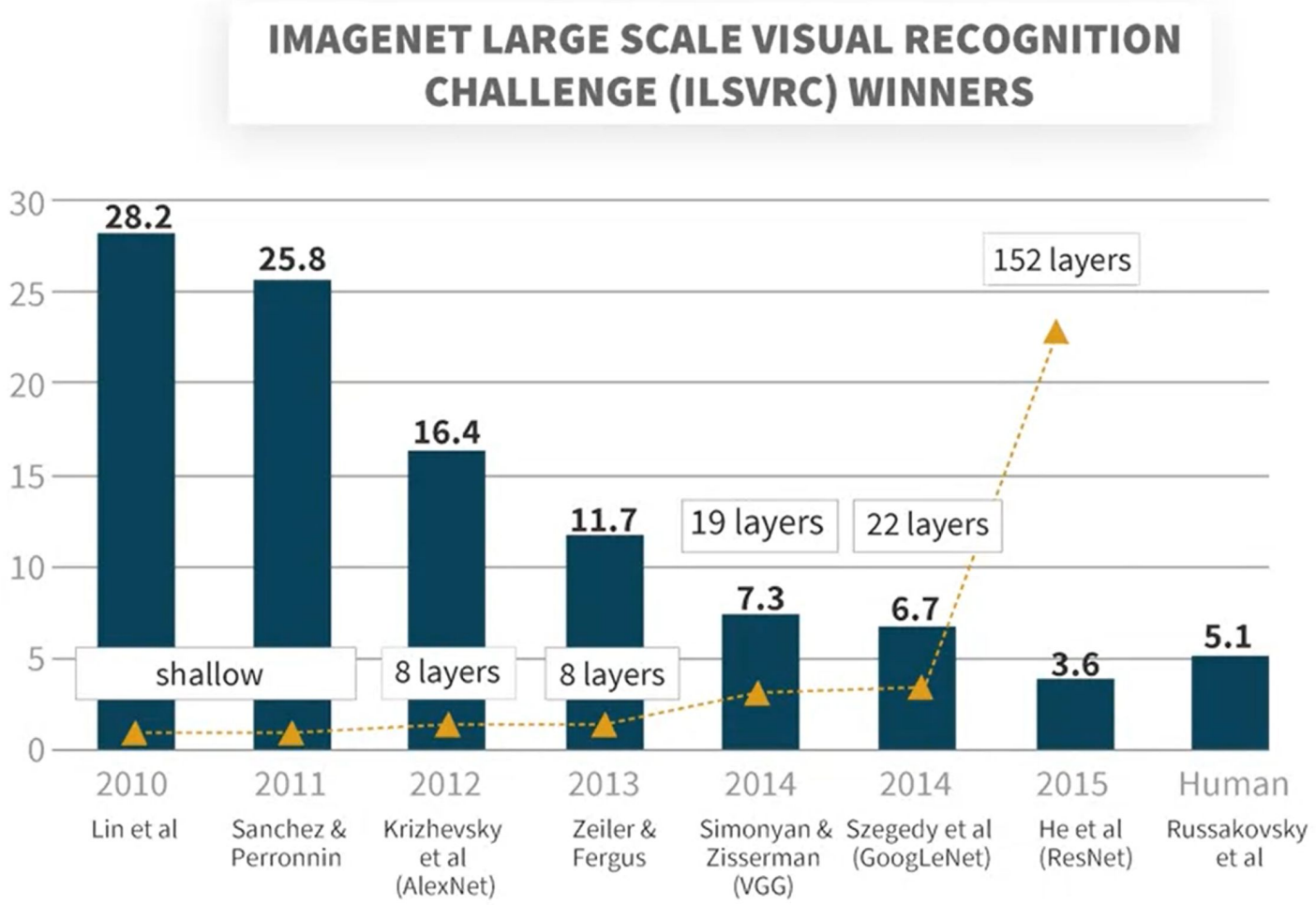
Not logged in. Login | Signup

ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images. The project has been instrumental in advancing computer vision and deep learning research. The data is available for free to researchers for non-commercial use.

https://www.image-net.org/

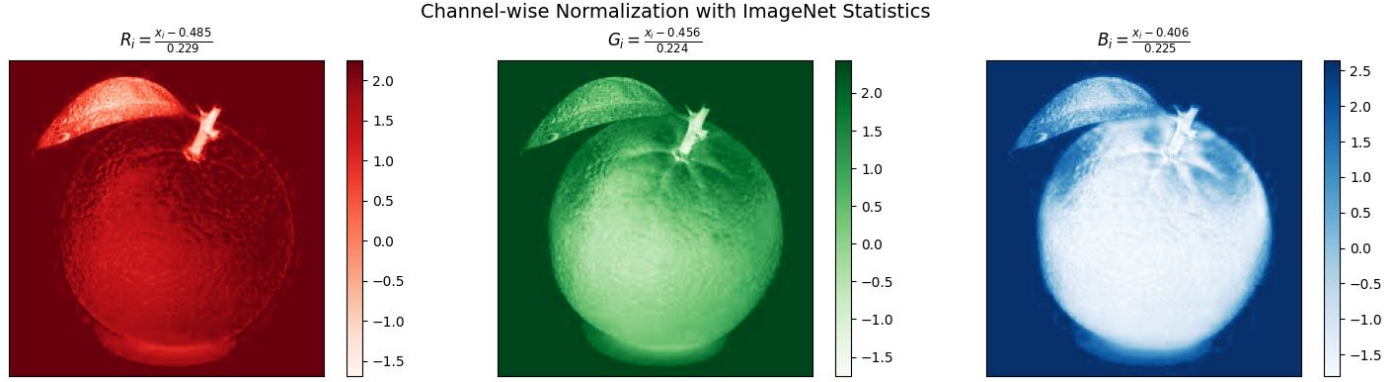


Source: [Introduction to Imagenet]

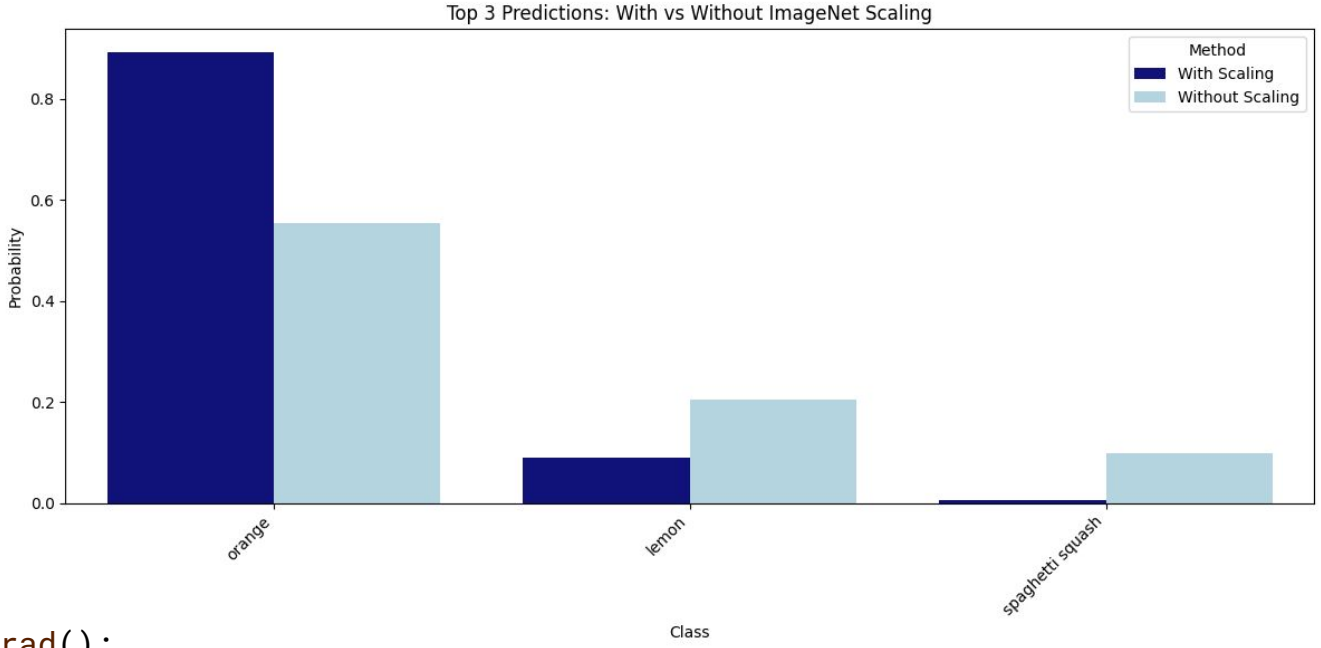

IMAGENET LARGE SCALE VISUAL RECOGNITION CHALLENGE (ILSVRC) WINNERS
30
25
20
15
10
5
0
28.2
25.8
16.4
11.7
7.3
6.7
3.6
5.1
shallow
8 layers
8 layers
19 layers
22 layers
152 layers
2010
Lin et al
2011
Sanchez & Perronnin
2012
Krizhevsky et al (AlexNet)
2013
Zeiler & Fergus
2014
Simonyan & Zisserman (VGG)
2014
Szegedy et al (GoogLeNet)
2015
He et al (ResNet)
Human
Russakovsky et al

The Imagenet transformations

```
# ImageNet scaling transformation
        transform = transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
              # This will convert the PIL image to a torch tensor with the same uint8 type
            transforms.ToImage(),
            # We convert the uint8 tensor to a float32 tensor and divide by 255
            transforms.ToDtype(torch.float32, scale=True),
            # We apply the standard scaler
            transforms.Normalize(
                mean=[0.485, 0.456, 0.406], #Red, Green, Blue
                std =[0.229, 0.224, 0.225]  #These stats come from the Imagenet's training set
            )
        ])
```



Predicting with and without scaling




```
with torch.no_grad():
     output = model(img_tensor)

 probabilities = torch.nn.functional.softmax(output[0], dim=0)
 top3_prob, top3_idx = torch.topk(probabilities, 3)
```

Accessing the transformations from torchvision's pretrained models

```python
import torchvision.transforms as transforms
from torchvision.models import ConvNeXt_Tiny_Weights

model =
torchvision.models.convnext_tiny(weights="IMAGENET1K_V1")

# Access the weights object
weights = ConvNeXt_Tiny_Weights.IMAGENET1K_V1

# Get the transform functions from the weights object
transforms_train = weights.transforms()
transforms_train
```


```
ImageClassification(
    crop_size=[224]
    resize_size=[236]
    mean=[0.485, 0.456, 0.406]
    std=[0.229, 0.224, 0.225]
    interpolation=InterpolationMode.BILINEA
)
```

Summary

Input normalization speeds up training

- Centers data around zero (typically mean = 0)
- Scales features to have similar variances (often standard deviation = 1)

ImageNet preprocessing pipeline

- Includes resizing, center cropping, and using Imagenet's training set statistics for mean and standard deviation of its training set of about 1 million images.

Impact on model performance and pretraining

- Proper scaling improves model predictions, particularly when using pre-trained models
- PyTorch provides easy access to the transformation that was used to train a model

Further reading and references

Efficient Backprop

- https://cseweb.ucsd.edu/classes/wi08/cse253/Handouts/lecun-98b.pdf

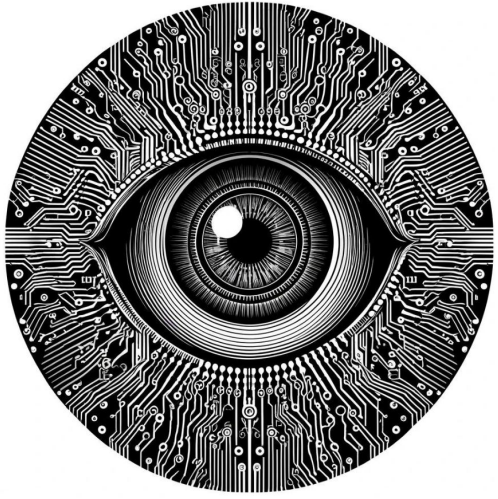
An Introduction to Imagenet

- https://blog.roboflow.com/introduction-to-imagenet/

Documentation of Convnext on PyTorch

- https://pytorch.org/vision/main/models/generated/torchvision.models.convnext_base.html#torchvision.models.convnext_base

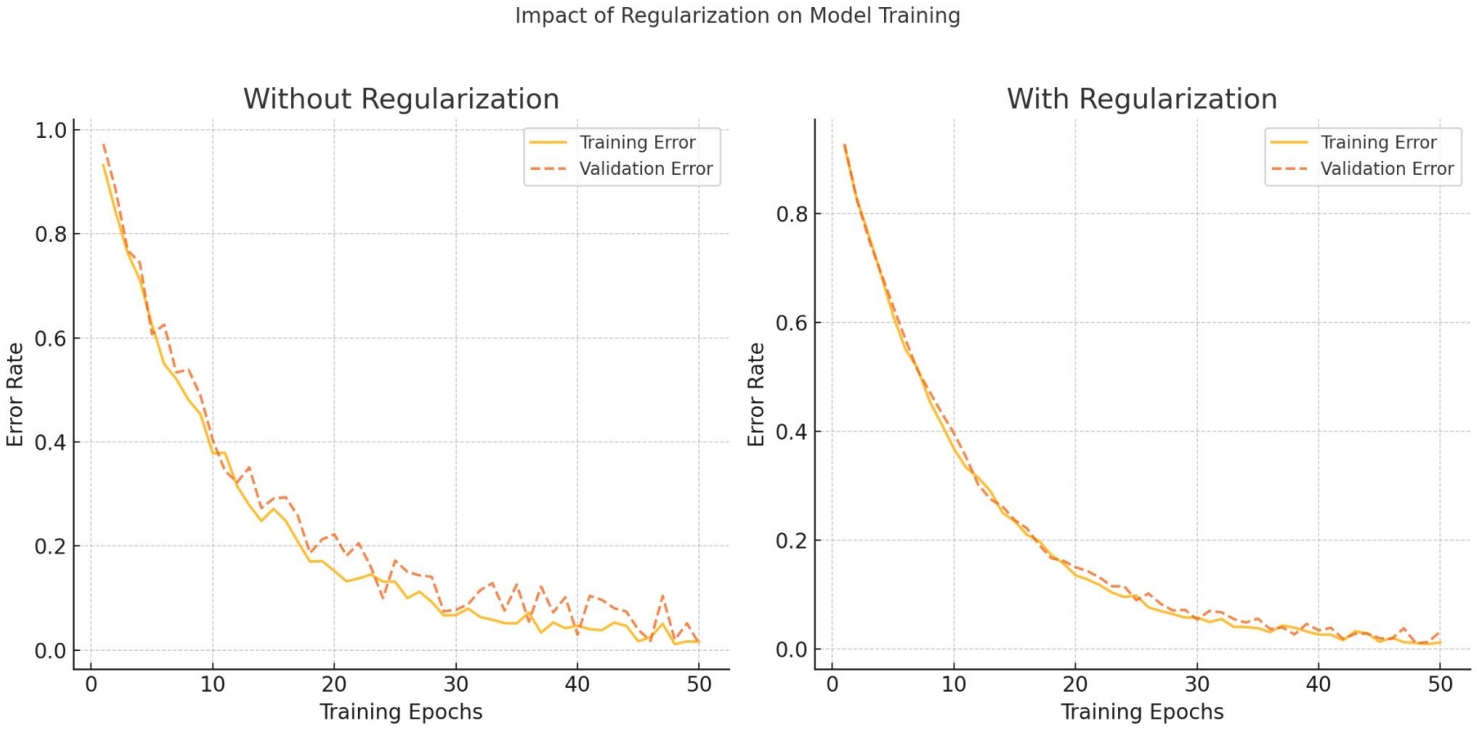
Regularization with Dropout and Batch Normalization



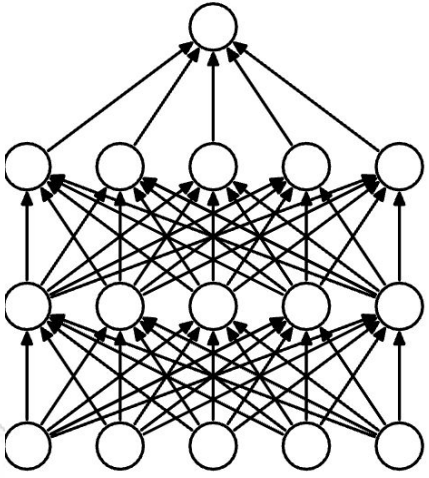
Learning goals

- Use dropout and batch normalization during the training process as regularization techniques
- Understand the behavior of batch normalization and dropout during training and inference with model.train() and model.eval()

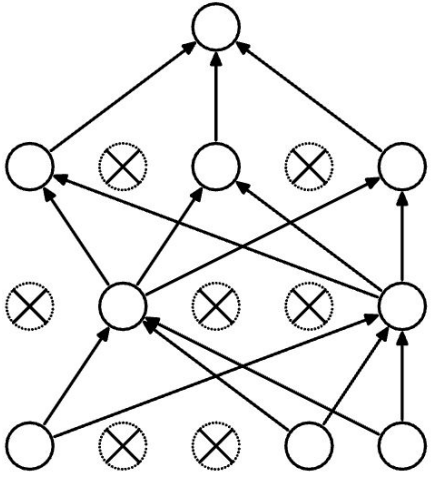
Overfitting vs underfitting - training vs validation



Dropout



(a) Standard Neural Net



(b) After applying dropout.

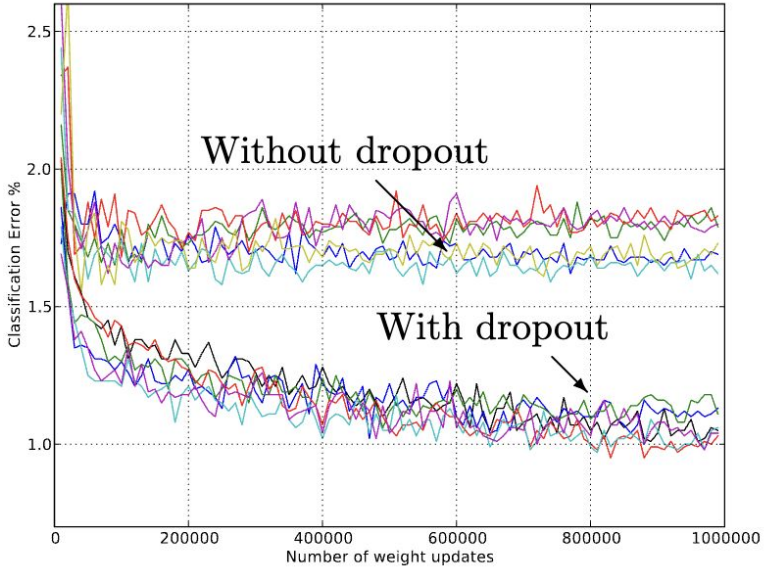
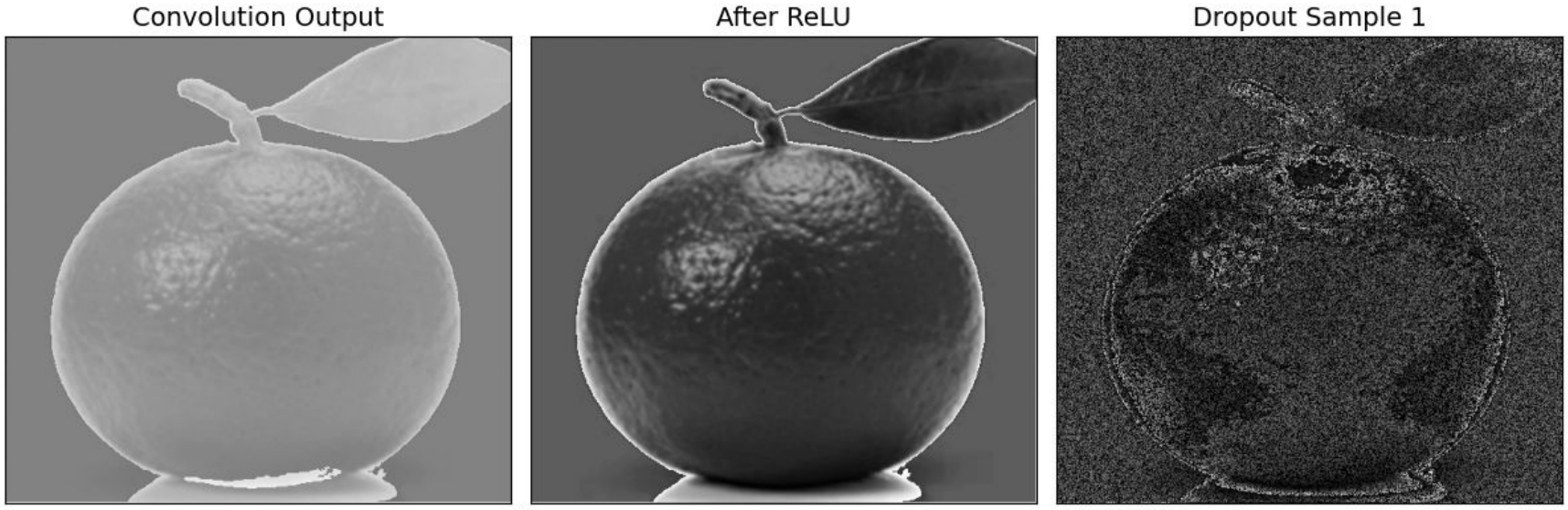


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Visualizing dropout




```
nn.Sequential(
    nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
    nn.Dropout(p=0.5)  # Dropout after ReLU
)
```

Batch Normalization

```
import torch.nn as nn
model = nn.Sequential(
        nn.Conv2d(in_channels=3,
                  out_channels=16,
                   kernel_size=3,
                   padding=1),
        nn.BatchNorm2d(num_features=16),
        nn.ReLU(),
        )
```

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

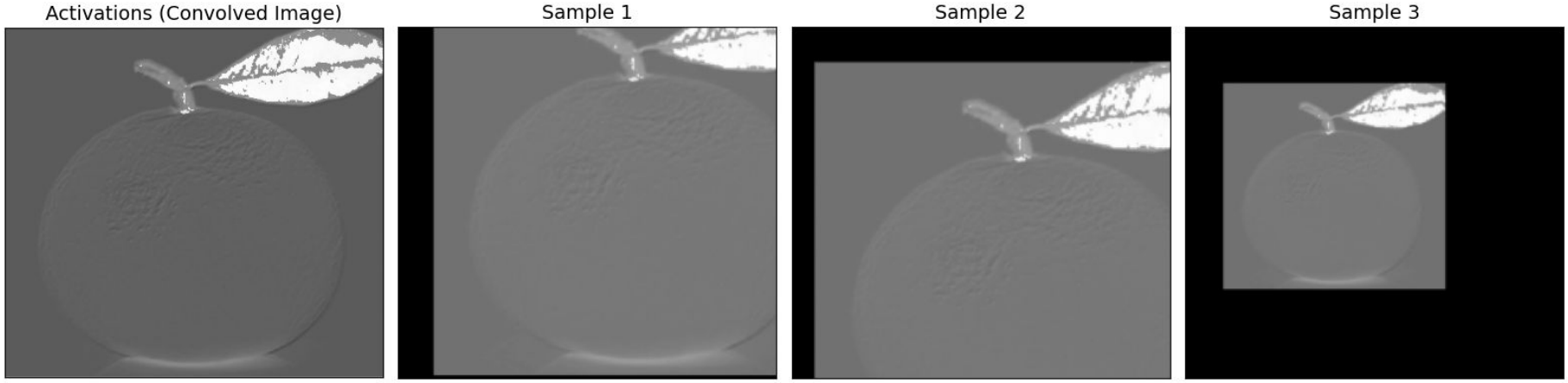
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

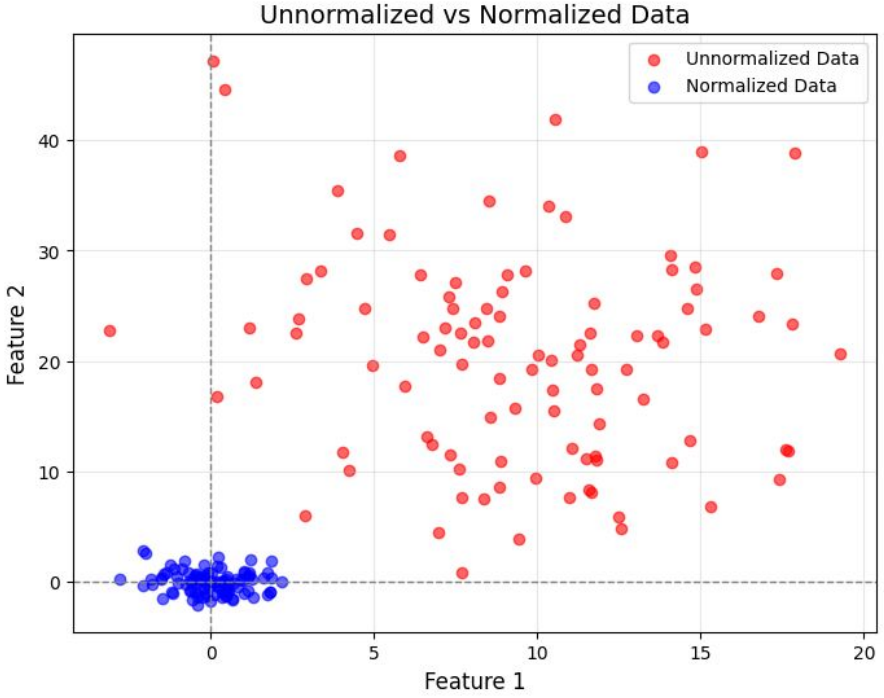
m corresponds to the batch size that we have defined in our DataLoader

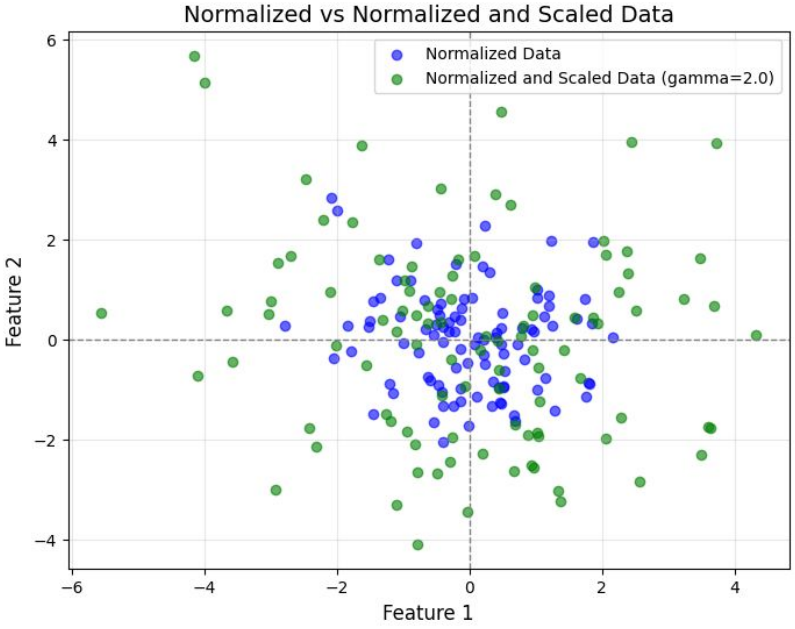
Regularization in batch normalization

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{\gamma_x}{\sqrt{\sigma_x^2+\varepsilon}} & 0 \\ 0 & \frac{\gamma_y}{\sqrt{\sigma_y^2+\varepsilon}} \end{pmatrix}}_{\text{the “scale” matrix}} \begin{pmatrix} x - \mu_x \\ y - \mu_y \end{pmatrix} + \underbrace{\begin{pmatrix} \beta_x - \frac{\gamma_x \mu_x}{\sqrt{\sigma_x^2+\varepsilon}} \\ \beta_y - \frac{\gamma_y \mu_y}{\sqrt{\sigma_y^2+\varepsilon}} \end{pmatrix}}_{\text{the “shift” vector}}.$$

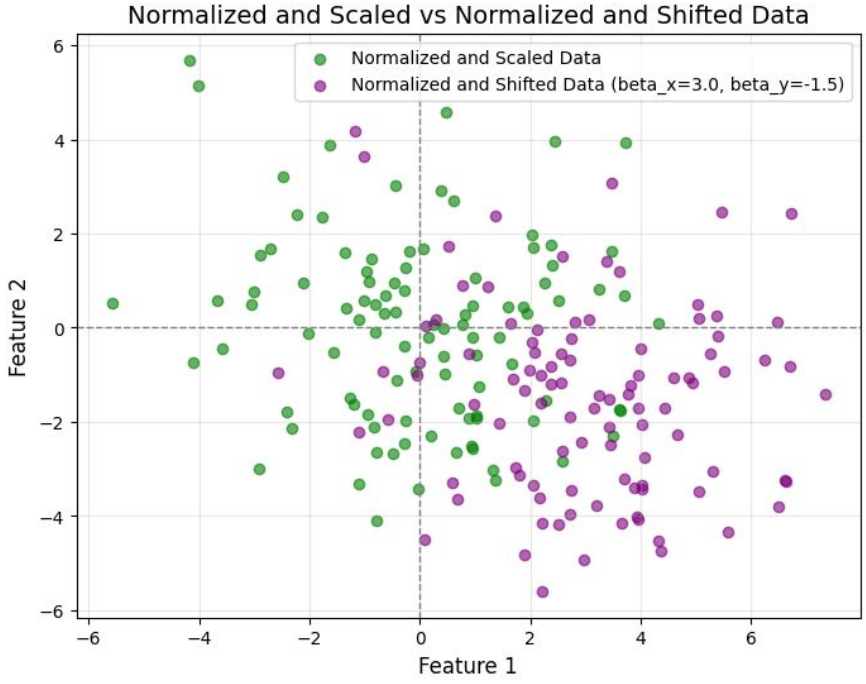


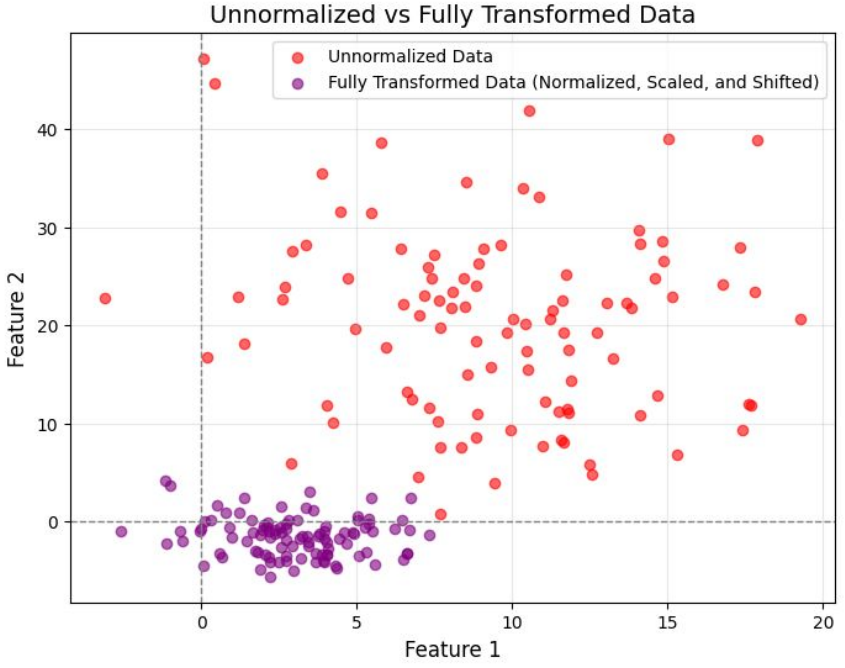
Visualizing batch normalization





Visualizing batch normalization

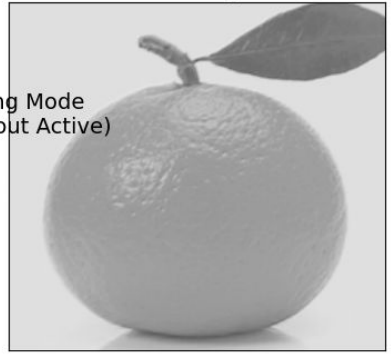




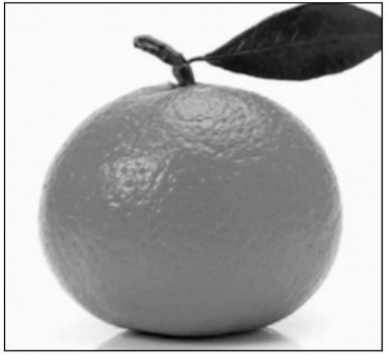
Dropout during model.train() and model.eval()

Training Mode
(Dropout Active)

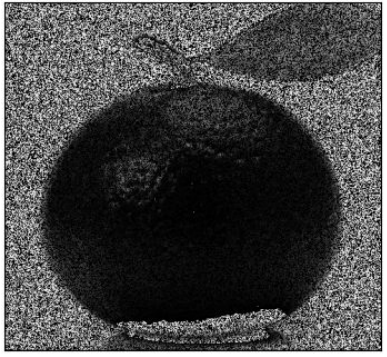
Conv Output



ReLU Output



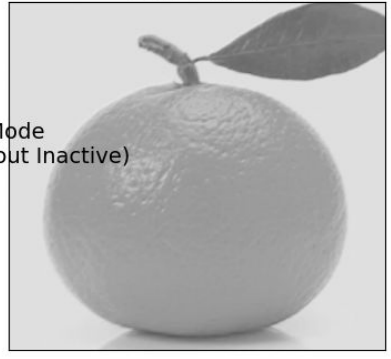
Train Mode
Dropout



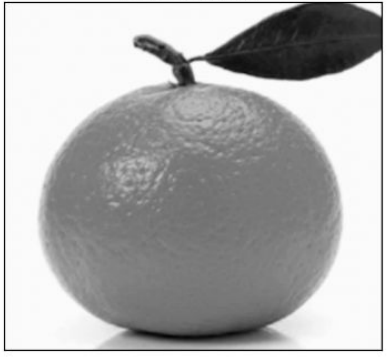
```
model.train()
```

Eval Mode
(Dropout Inactive)

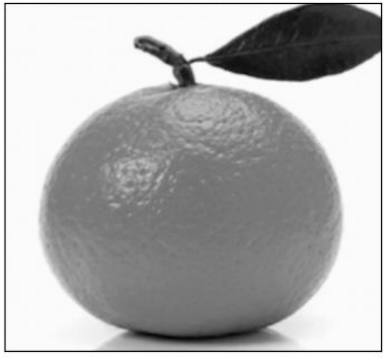
Conv Output



ReLU Output



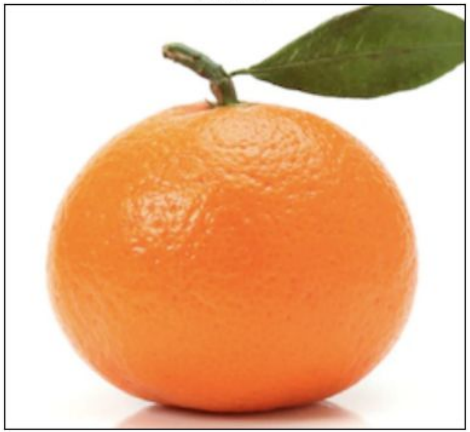
Eval Mode
Dropout



```
model.eval()
```

Dropout makes inference output non-deterministic

input

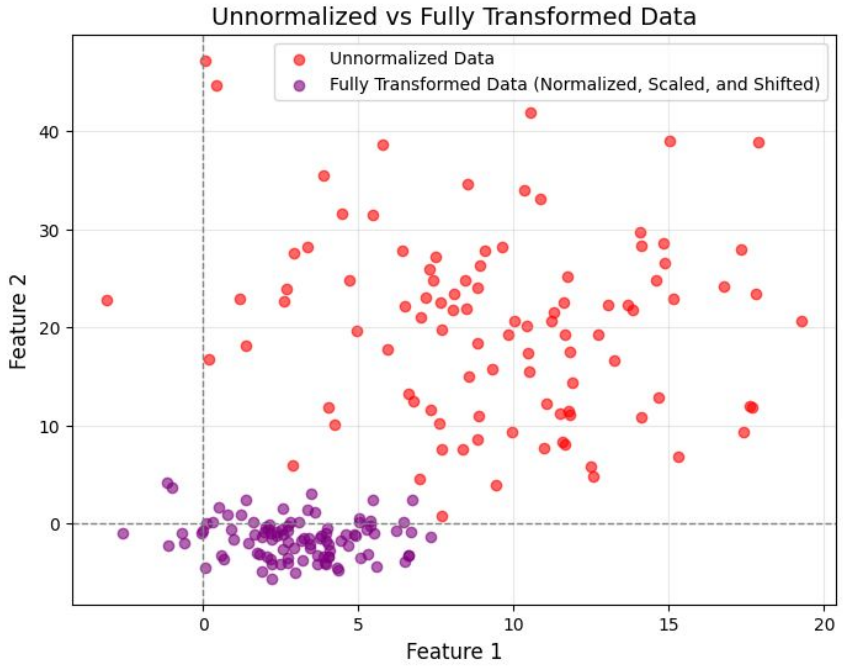


```
model.train()
```

orange: 0.9979	orange: 0.9703
lemon: 0.0014	lemon: 0.0151
croquet ball: 0.0002	Granny Smith: 0.0104
Granny Smith: 0.0001	pomegranate: 0.0019
banana: 0.0001	spaghetti squash: 0.0007

Evaluation on a pretrained VGG-16 network

BatchNorm during model.eval()



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

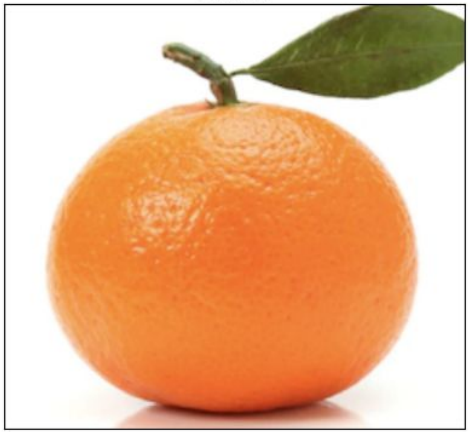
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Effect of BatchNorm in a pretrained resnet18

input



batch size = 1

```
model.eval()
```

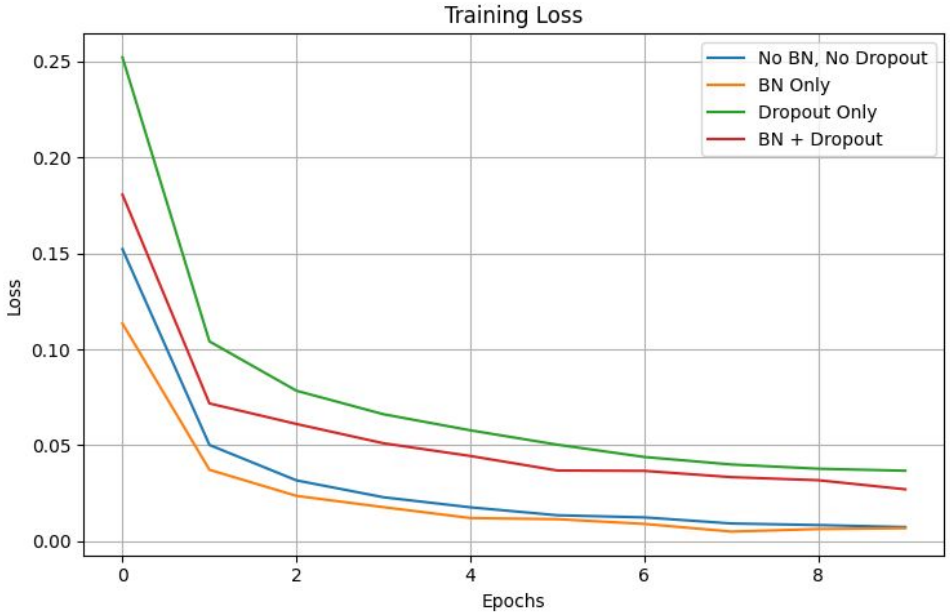
```
orange: 0.9701
lemon: 0.0286
Granny Smith: 0.0004
banana: 0.0003
pomegranate: 0.0001
```

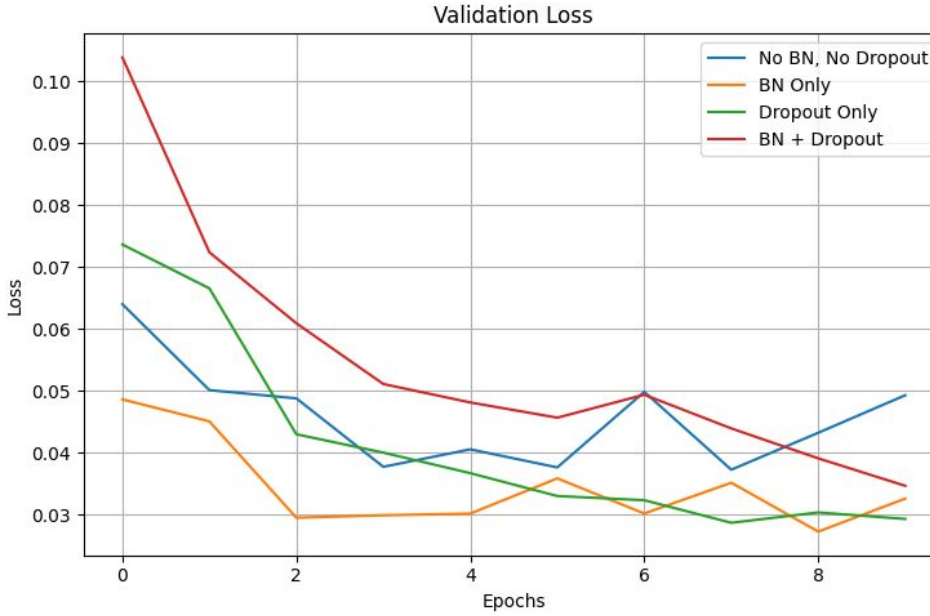
```
model.train()
```

```
bucket: 0.0086
plunger: 0.0065
hook: 0.0061
waste container: 0.005
ladle: 0.0051
```

Batch statistics affect the probability outputs

Do your own experiments





[Colab notebook]

Summary

Both dropout and batch normalization function as data augmentation

- They work on activations (e.g. convolution outputs) instead of input images

Dropout and batch normalization need to be controlled during inference

- `model.eval()` disables dropout producing deterministic output
- Batch normalization is still applied after `model.eval()`, however it uses then the mean and the standard deviation of the whole training set, instead of just the batch

Dropout and batch normalization may act against each other

- Both provide regularization effects, combining them might require careful tuning of their settings

Further reading and references

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

- https://arxiv.org/pdf/1502.03167

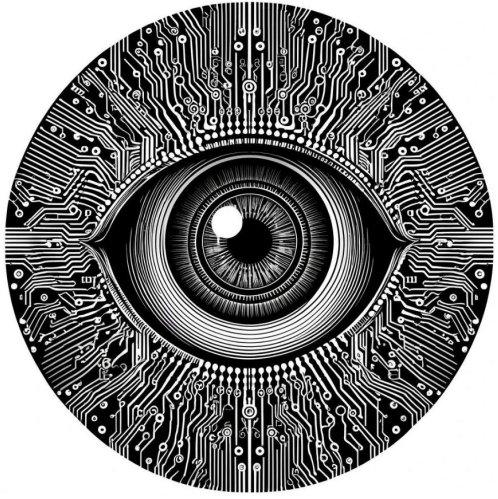
Dropout: A Simple Way to Prevent Neural Networks from Overfitting

- https://jmlr.org/papers/v15/srivastava14a.html

Where to use model.eval()? (PyTorch forum discussion)

- https://discuss.pytorch.org/t/where-to-use-model-eval/89200/2

Skip Connections



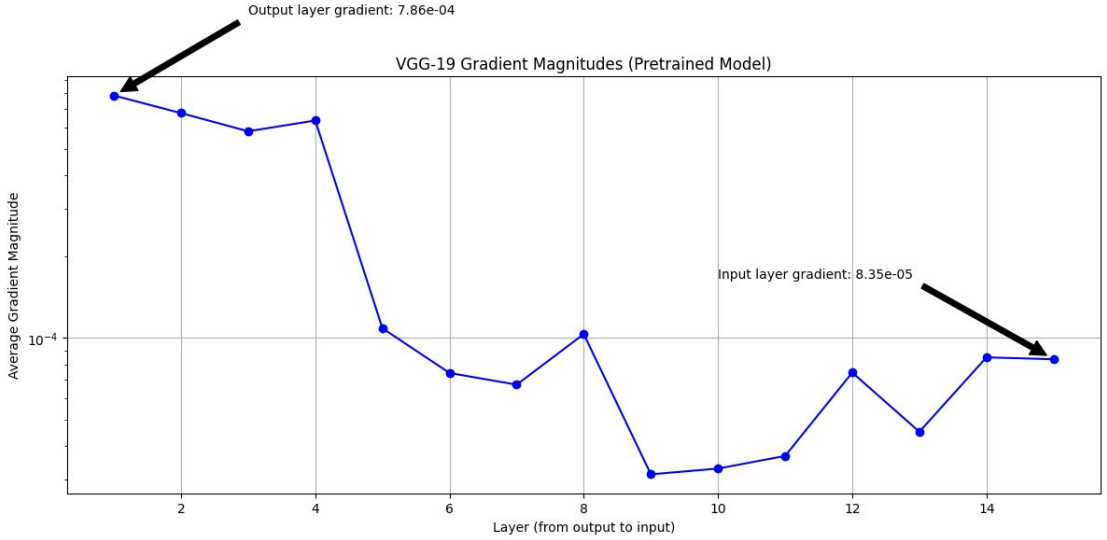
Antonio Rueda-Toicen

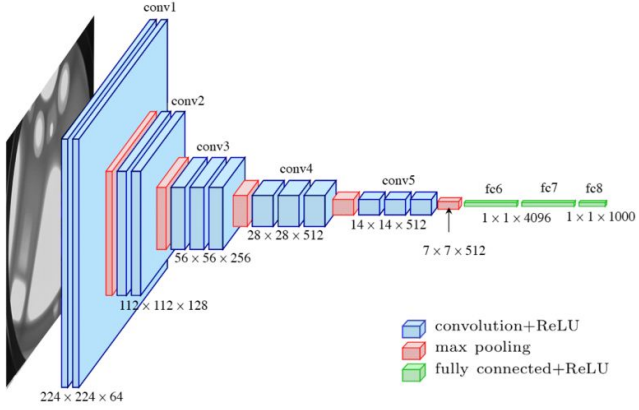
Learning goals

- Understand the vanishing gradient as a numerical problem
- Implement skip connections as element-wise addition or concatenation of activation maps

The vanishing gradient

$$w_{ij} = w_{ij} - (\text{learning rate} * \frac{dL}{dw_{ij}})$$

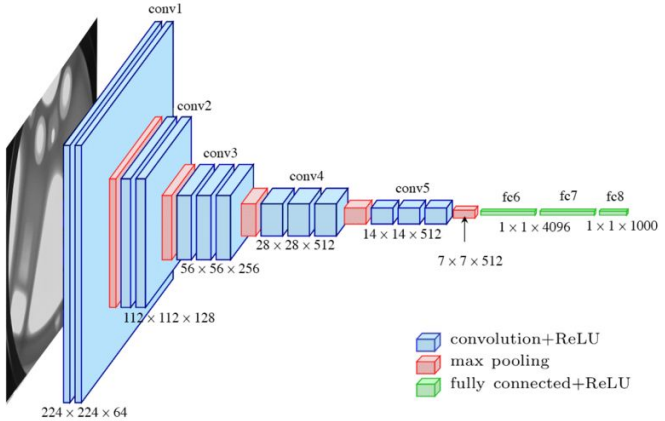




VGG-19 network (source)

Numerical underflow in neural networks

```python
import numpy as np
from scipy.signal import convolve2d

# Example data (any image or 2D array)
image = np.ones((8,8), dtype=np.float32)

# A 3×3 kernel with sum=0.8
kernel = np.array([[0.05, 0.10, 0.05],
                   [0.10, 0.20, 0.10],
                   [0.05, 0.10, 0.05]], dtype=np.float32)

for i in range(1000):
    image = convolve2d(image, kernel, mode='same', boundary='fill', fillvalue=
    # Underflow can show up when values drop below np.finfo(np.float32).tiny
    if (image > 0).sum() == 0:
        print("All values underflowed to 0 at iteration", i)
        break
```



VGG-19 network (source)

Numerical underflow

```python
import numpy as np

a = 1e-8 # Equal to 1 x 10 ** -8
b = 2

print(np.float32(a) ** b)  # Gives a value close to 1e-16
print(np.float16(a) ** b)  # Underflows to 0.0
```

$$(0.00000001)^2 = (1 \times 10^{-8})^2 = 10^{-16}$$

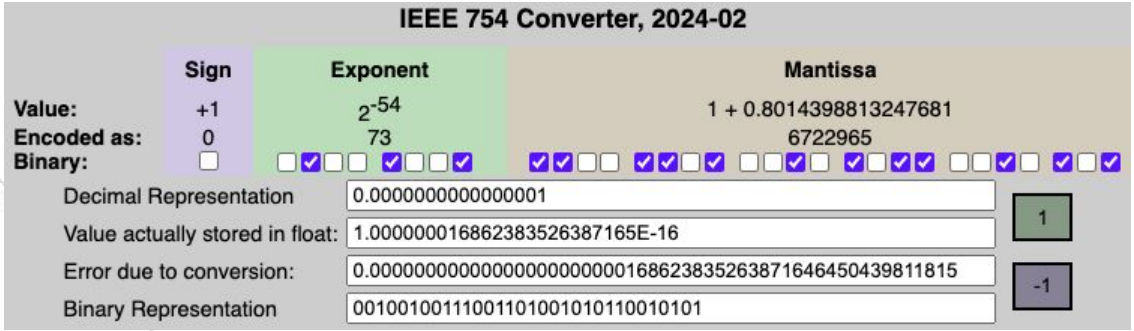
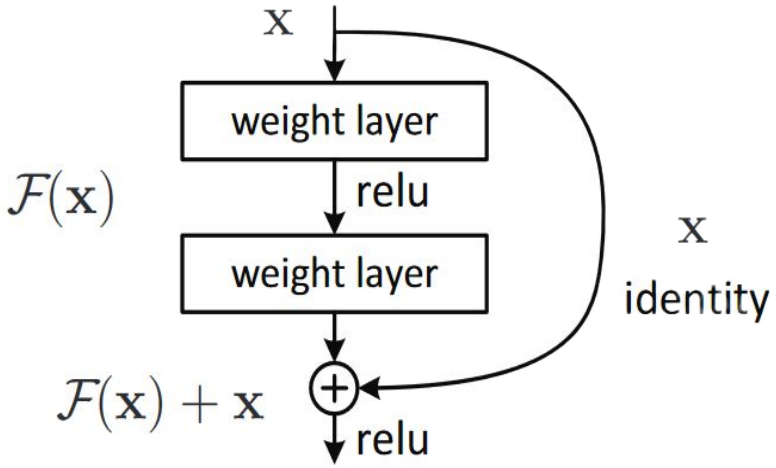


Image from [IEEE-754 Floating Point Converter]

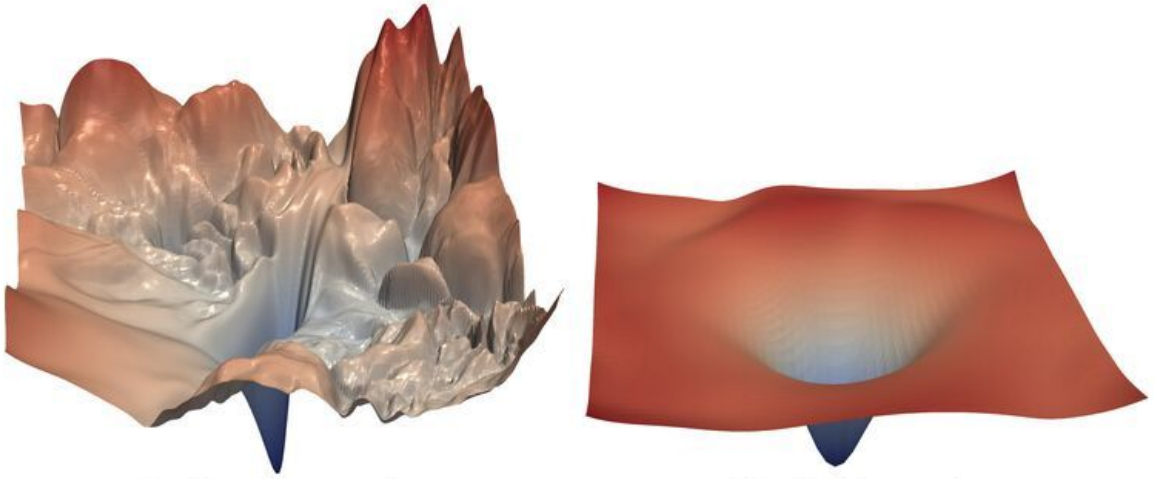
Skip connections on Resnet

```python
import torch.nn as nn

class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        # Main path - "city route"
        self.conv1 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(channels)
        self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(channels)
        self.relu = nn.ReLU()

    def forward(self, x):
        # Save input for skip connection - "highway route / checkpoint"
        identity = x

        # Main path through convolutions
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)

        # Add skip connection - "merging highway with city route (adding checkpoint)"
        out += identity

        # Final activation
        out = self.relu(out)

        return out
```



$$\underbrace{\begin{bmatrix} -0.12 & 0.24 \\ 0.35 & -0.25 \end{bmatrix}}_{\text{Identity Path (x)}} + \underbrace{\begin{bmatrix} 0.25 & -0.14 \\ -0.45 & 0.35 \end{bmatrix}}_{\text{Main Path (F(x))}} = \underbrace{\begin{bmatrix} 0.13 & 0.10 \\ -0.10 & 0.10 \end{bmatrix}}_{\text{Output (F(x) + x)}}$$

Effects on the loss landscape



(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

Image from Visualizing the Loss Landscape of Neural Nets

Relevance on current architectures

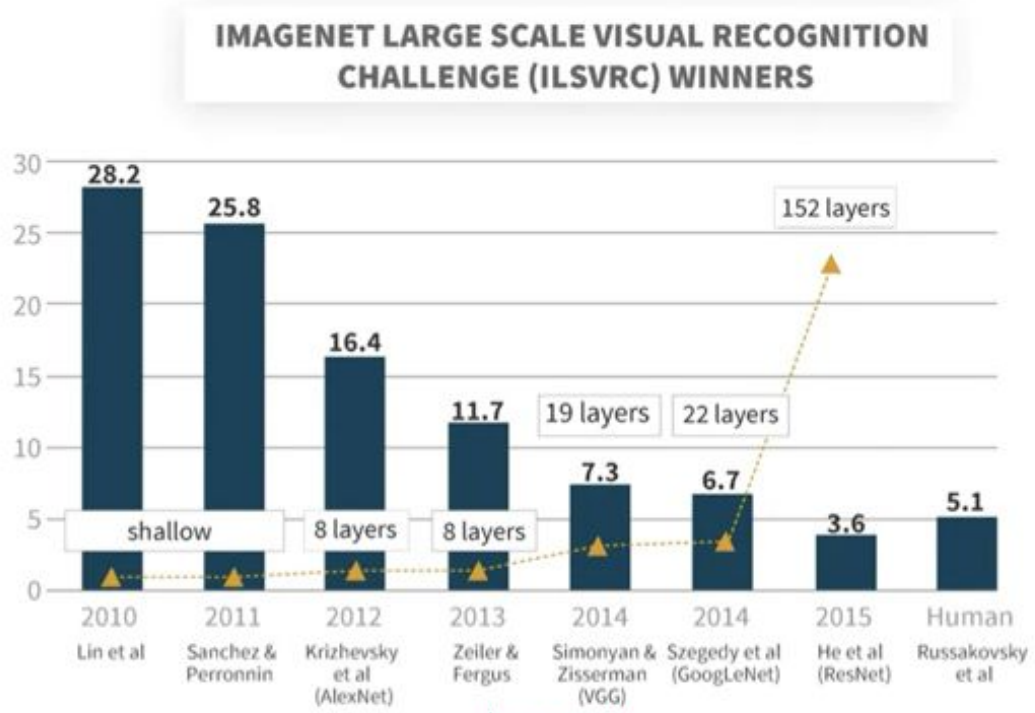
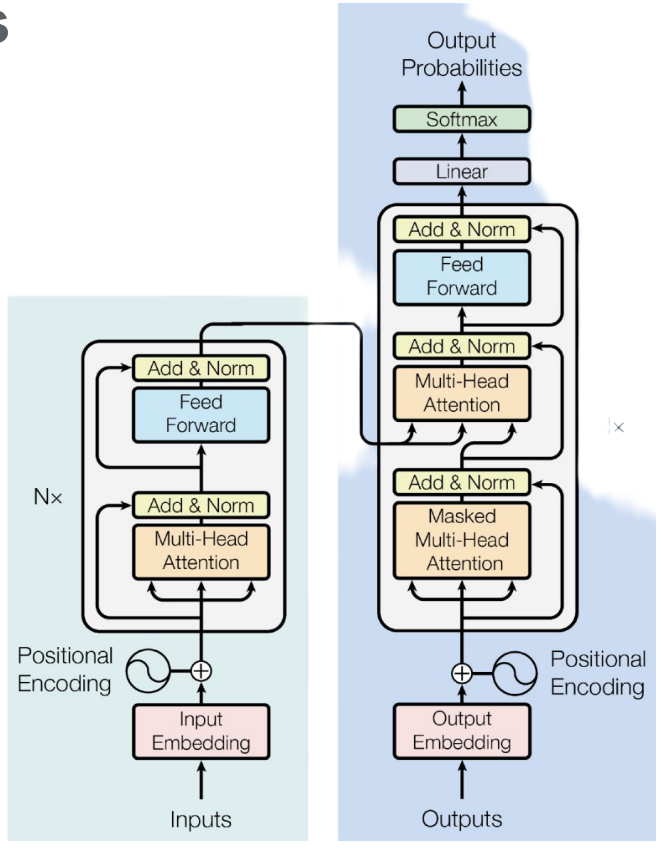


Image source



Transformer architecture from source

Skip connections on Densenet

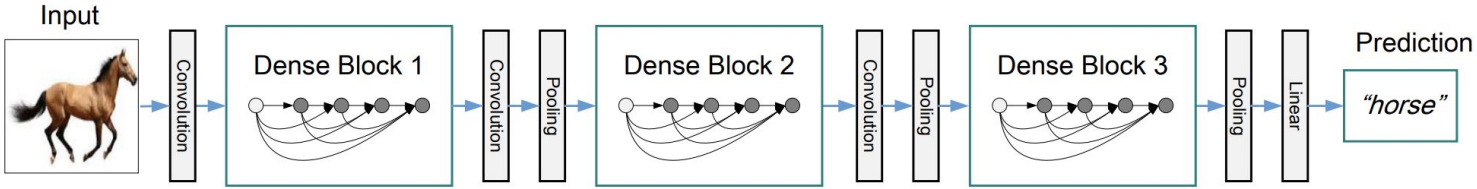
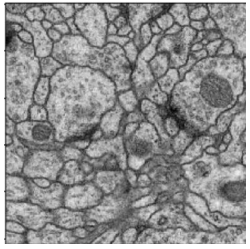
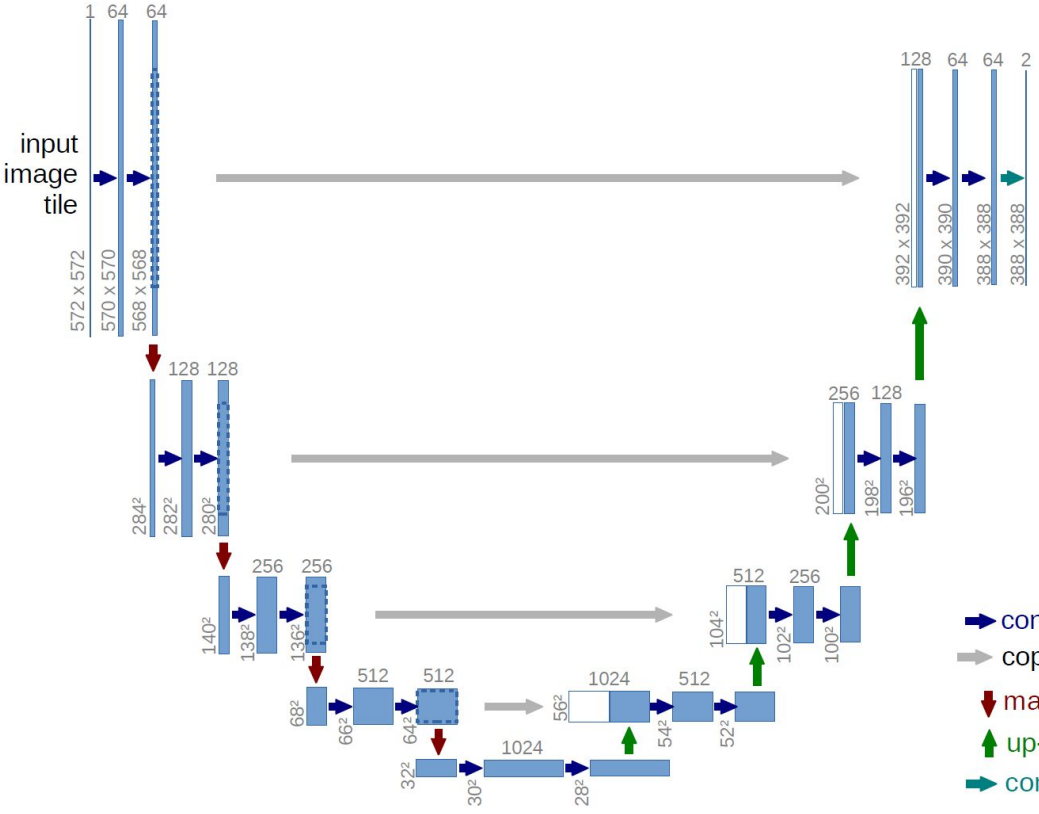


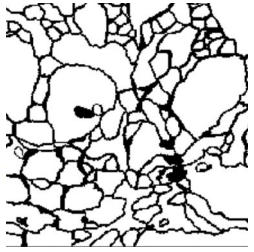
Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

```
# Feature maps are concatenated instead of added
# We can control the number of feature maps by using 1x1 convolutions
torch.cat(features, dim=1)
```

Skip connections on U-net







Summary

The vanishing gradient is a numerical problem

- Computers have limited precision to represent small numbers

Skip connections serve as “checkpoints” for what the model has learned

- A skip connection gives us the chance to preserve information that could have been destroyed due to numerical underflow
- Skip connections are what allow neural networks to be deep and increase their number of parameters while avoiding vanishing gradients

Two types of skip connections: addition and concatenation

- We use either element wise addition or concatenation of feature maps as skip connections

Further reading and references

Deep Residual Learning for Image Recognition

- https://arxiv.org/abs/1512.03385

Densely Connected Convolutional Networks

- https://arxiv.org/abs/1608.06993

Visualizing the Loss Landscape of Neural Nets

- https://arxiv.org/abs/1712.09913

Resources

- Github Repository
- YouTube playlist
- Discord channel
  #practical-computer-vision-workshops