EE 046211 - Technion - Deep Learning

Tal Daniel (https://taldatech.github.io)

Tutorial 08 - Training Methods

Agenda

- Feature Scaling: Normalization and Standartization
- Types of Layer Normalizations in Neural Networks
  - Batch Normalization
  - Layer Normalization
  - Instance Normalization
  - Group Normalization
- Vanishing/Exploding Gradients and Gradient Clipping
- Skip-Connections
- Hyperparameter Tuning
  - Hyperparameter Tuning with Optuna and PyTorch
- Recommended Videos
- Credits

In [1]:

```python
# imports for the tutorial
import numpy as np
import matplotlib.pyplot as plt
import plotly
import time
import os

# pytorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data
from torchvision import datasets
from torchvision import transforms

# optuna
import optuna
```

Feature Scaling: Normalization and Standartization

- Feature scaling is a fundamental part of the data pre-processing stage.
- It can improve the performance of some machine learning algorithms, but may also harm others.
- It is especially important for **Gradient Descent-based** algoirthms such as linear regression, logistic regression, neural networks, and etc.
- The range of features also significantly affects **distance-based** algorithms such as KNN, SVM and K-Means.

- For example, let's take a look at the general formulation of Gradient Descent for linear regression:

$$\theta_j^{(t+1)} \leftarrow \theta_j^{(t)} - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

- The presence of feature value X in the formula will affect the step size of the gradient descent!
  - The difference in ranges of features will cause different step sizes for each feature.
- To ensure that the gradient descent moves smoothly towards the minima and that the steps for gradient descent are updated at the same rate for all the features, we scale the data before feeding it to the model.
- **Having features on a similar scale can help the gradient descent converge more quickly towards the minima.**

Normalization - MinMax Scaling

- Normalization is a scaling technique in which values are shifted and rescaled so that they end up **ranging between 0 and 1**.
  - It is also known as Min-Max scaling.

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

  - X_{max} and X_{min} are the maximum and the minimum values of the feature respectively.
- Normalization is good to use when you know that the distribution of your data does not follow a Gaussian distribution.
- This can be useful in algorithms that do not assume any distribution of the data like K-NN and Neural Networks.
- For example, when we worked with images, we used the `ToTensor()` transformation that normalized pixel values to [0, 1].
  - For some architectures/tasks (e.g., Generative Adversarial Networks ~ GAN), sometimes it is useful to normalize to [-1, 1].

Standartization

- Standardization is a scaling technique where the values are centered around the mean with a unit standard deviation.
- This means that the mean of the features becomes zero and the resultant distribution has a unit standard deviation.

$$X_{scaled} = \frac{X - \mu}{\sigma}$$

  - μ is the (empirical) mean of the feature values and σ is the (empirical) standard deviation of the feature values.
  - Note that in this case, the values are not restricted to a particular range.
- Standardization can usually be helpful in cases where the data follows a Gaussian distribution (but can work otherwise).
- Unlike normalization, standardization does not have a bounding range. So, even if you have outliers in your data, they will not be affected by standardization.

Feature Scaling In Practice

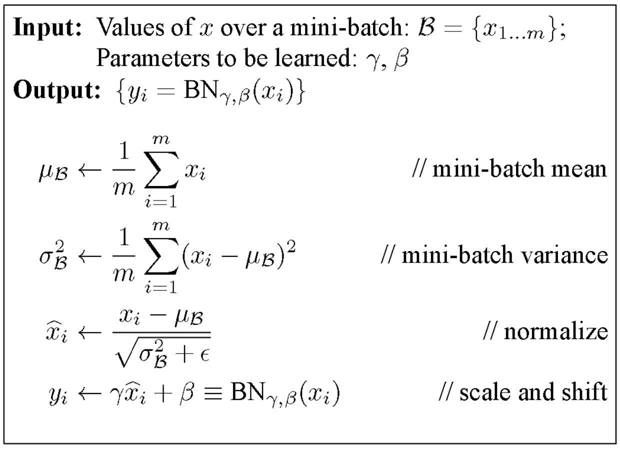
- At the end of the day, the choice of using normalization or standardization will depend on your problem and the machine learning algorithm you are using.
- You can always start by fitting your model to raw, normalized and standardized data and compare the performance for the best results.
- It is a good practice to fit the scaler on the training data and then use it to transform the testing data.
  - This would avoid any **data leakage** during the model testing process.
- Scaling of target values (or labels) is usually not required.

📏 Types of Layer Normalizations in Neural Networks

- The basic idea behind normalization layers is to normalize the output of an activation layer to improve the convergence during training.
  - Getting normalization right can be a crucial factor in getting your model to train effectively.
- We first cover **Batch Normalization** (BN) which is different than the rest of normalizations: unlike batch normalization, these normalizations **do not work on batches**, instead they normalize the activations of a **single sample**, making them suitable for recurrent neual networks as well.

Batch Normalization

- Batch Normalization is a technique for improving the speed, performance, and stability of deep neural networks.
  - The reasons behind its effectiveness remain under discussion.
- It is used to normalize the input layer by adjusting and scaling the activations.
- Formally:
  - **Input**: $x \in \mathcal{R}^{N \times D}$
  - **Learnable Parameters (scale and shift)**: $\gamma, \beta \in \mathcal{R}^{D}$
  - **Intermediates**: $\mu, \sigma \in \mathcal{R}^{D}$, $\hat{x} \in \mathcal{R}^{N \times D}$
  - **Output**: $y \in \mathcal{R}^{N \times D}$
- In PyTorch:
  - 1D: `torch.nn.BatchNorm1d()`
  - 2D (images): `torch.nn.BatchNorm2d()`
- In CNNs, we work with inputs of shape $[N, C, H, W]$, where N is the batch size, C is the number of channels and H, W are the height and width of the feature map respectively. BatchNorm in this case is performed **channel-wise**, i.e., on the channel dimension C such that $\gamma, \beta \in \mathcal{R}^{C}$.
- BN behaves differently during train (`model.train()`) and evaluation (`model.eval()`).
  - `model.train()` : BN parameters are calculated over the batch.
  - `model.eval()` : BN uses the *learned* `running_mean` and `running_std` to normalize the data (calculated as a moving average during training).



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

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

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

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

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

In [5]:

```
data = torch.randn([2, 3, 8, 8])  # a batch of 2 RGB (ch=3) images of size 8x8 (h, w)
# batch normalization
bn = torch.nn.BatchNorm2d(3, affine=False, momentum=0, track_running_stats=False)
d_out = bn(data)
print(d_out[0, 0, :, :])
```

```
tensor([[ 0.2939, -0.9834, -0.8119, -0.4152,  0.8119,  1.2622,  0.0931,  0.2426],
        [-0.9494, -0.3924,  1.5611, -0.0653,  1.0349,  0.4875,  0.7305, -0.3094],
        [-1.1611,  0.4114, -1.1368,  1.8561, -0.1057, -0.2810, -0.5084, -0.2459],
        [ 1.3733,  1.2246, -1.9154,  1.0214,  1.6371, -0.1558,  1.5632, -0.3496],
        [ 0.1105, -1.1744, -0.2636,  0.0429,  0.8881,  0.6445,  0.9153,  0.0470],
        [-0.6980, -0.5053, -1.6660, -0.6632, -0.9265,  0.8809, -0.2508,  1.0830],
        [ 0.7676,  1.0431,  0.4111, -0.4280, -0.1839,  0.8650,  0.6845, -0.3476],
        [ 0.5521, -1.3925,  0.9824, -2.5798, -1.0396,  1.4582,  1.4256, -0.3339]])
```

```
In [8]: # BN under the hood
        mean = data.mean(dim=(0, 2, 3), keepdim=True)  # we consider the [h, w] over all of the batch, the "avg. c
        hannel"
        var = data.var(dim=(0, 2, 3), unbiased=False, keepdim=True)
        eps = 1e-05
        print(f'bn values shape: mean: {mean.shape}, std: {var.shape}')
        d_n = (data - mean) / (var + eps).sqrt()
        print(d_n[0, 0, :, :])
```

```
bn values shape: mean: torch.Size([1, 3, 1, 1]), std: torch.Size([1, 3, 1, 1])
tensor([[ 0.2939, -0.9834, -0.8119, -0.4152,  0.8119,  1.2622,  0.0931,  0.2426],
        [-0.9494, -0.3924,  1.5611, -0.0653,  1.0349,  0.4875,  0.7305, -0.3094],
        [-1.1611,  0.4114, -1.1368,  1.8561, -0.1057, -0.2810, -0.5084, -0.2459],
        [ 1.3733,  1.2246, -1.9154,  1.0214,  1.6371, -0.1558,  1.5632, -0.3496],
        [ 0.1105, -1.1744, -0.2636,  0.0429,  0.8881,  0.6445,  0.9153,  0.0470],
        [-0.6980, -0.5053, -1.6660, -0.6632, -0.9265,  0.8809, -0.2508,  1.0830],
        [ 0.7676,  1.0431,  0.4111, -0.4280, -0.1839,  0.8650,  0.6845, -0.3476],
        [ 0.5521, -1.3925,  0.9824, -2.5798, -1.0396,  1.4582,  1.4256, -0.3339]])
```

Why Not Batch Normalization?

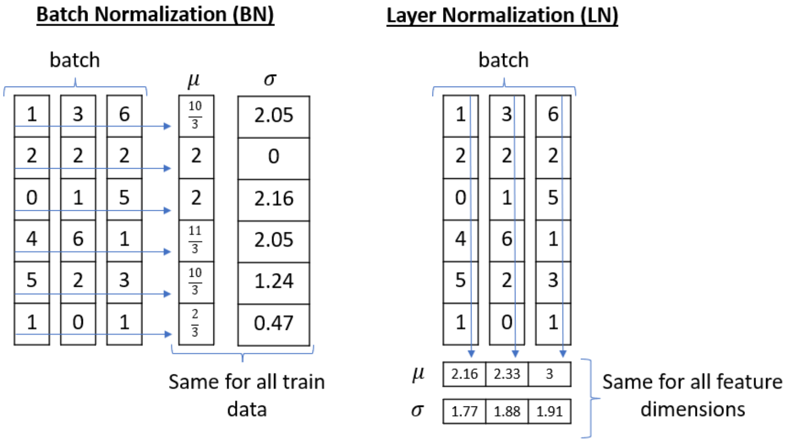
- In normalization, we ideally want to use the **global** mean and variance to standardize our data.
- Computing this for each layer is far too expensive though, so we need to approximate using some other measures.
- In batch normalization, this measure is the mean/variance of the **mini-batch**.
- Reasons for BN to not perform well:
  - **Small batch size** - If the batch size is 1, the variance is 0 so batch normalization cannot be applied. Slightly larger mini-batch sizes won't have this problem, but small mini-batches make our estimates very noisy and can negatively impact training, meaning batch normalization imposes a certain lower bound on our batch size.
  - **Recurrent connections in a recurrent neural network (RNN)** - In an RNN, the recurrent activations of each time-step will have different statistics. This means that we have to fit a separate batch normalization layer for each time-step. This makes the model more complicated and - more importantly - it forces us to store the statistics for each time-step during training.

Layer Normalization

- Recall that a mini-batch consists of multiple examples with the same number of features. Mini-batches are tensors where one axis corresponds to the batch and the other axis - or axes - correspond to the feature dimensions.
- The key feature of layer normalization is that it **normalizes the inputs across the features**.
  - In **batch normalization**, the statistics are computed **across the batch** and are the same for each example in the batch.
  - In **layer normalization**, the statistics are computed **across each feature** and are independent of other examples.
- The equations of layer normalization are very similar to that of batch normalization, where the difference is the dimensions of the computed statistics:

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

- As in BN, γ and β are learned parameters.
- In PyTorch: [`torch.nn.LayerNorm()` (https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html#torch.nn.LayerNorm)](https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html#torch.nn.LayerNorm)



In [6]:

```
x = torch.randn(20, 5, 10, 10)
# With Learnable Parameters
m = nn.LayerNorm(x.size()[1:])
# Without Learnable Parameters
# m = nn.LayerNorm(x.size()[1:], elementwise_affine=False)
# Normalize over last two dimensions
# m = nn.LayerNorm([10, 10])
# Normalize over last dimension of size 10
# m = nn.LayerNorm(10)
# Activating the module
output = m(x)
print(list(m.parameters())[0].shape) # the shape of gamma (and beta)
```

```
torch.Size([5, 10, 10])
```

In [ ]:

```
# NLP Example
batch, sentence_length, embedding_dim = 20, 5, 10
embedding = torch.randn(batch, sentence_length, embedding_dim)
layer_norm = nn.LayerNorm(embedding_dim)
# Activate module
layer_norm(embedding)

# Image Example
N, C, H, W = 20, 5, 10, 10
x_in = torch.randn(N, C, H, W)
# Normalize over the last three dimensions (i.e. the channel and spatial dimensions)
layer_norm = nn.LayerNorm([C, H, W])
output = layer_norm(x_in)
```

Instance Normalization

- Instance normalization is similar to layer normalization but it goes one step further--it computes the mean and std and **normalize across each channel** in each training example.
- It is mainly designed for visual tasks, such as style transfer, and the problem it tries to address is that the network should be agnostic to the contrast of the original image.
  - Thus, it is better to use with CNNs and not RNNs.
- The equations are the same as before, but the dimensions are different.
- Instance Normalization and Layer Normalization are very similar, but have some subtle differences. IN is applied on each channel of channeled data like RGB images, but LN is usually applied on entire sample and often in NLP tasks.
  - Additionally, LN applies elementwise affine transform, while **IN usually doesn't apply affine transform**.
- In PyTorch: `torch.nn.InstanceNorm2d()` (https://pytorch.org/docs/stable/generated/torch.nn.InstanceNorm2d.html#torch.nn.InstanceNorm2d) (there are also 1d and 3d variations)

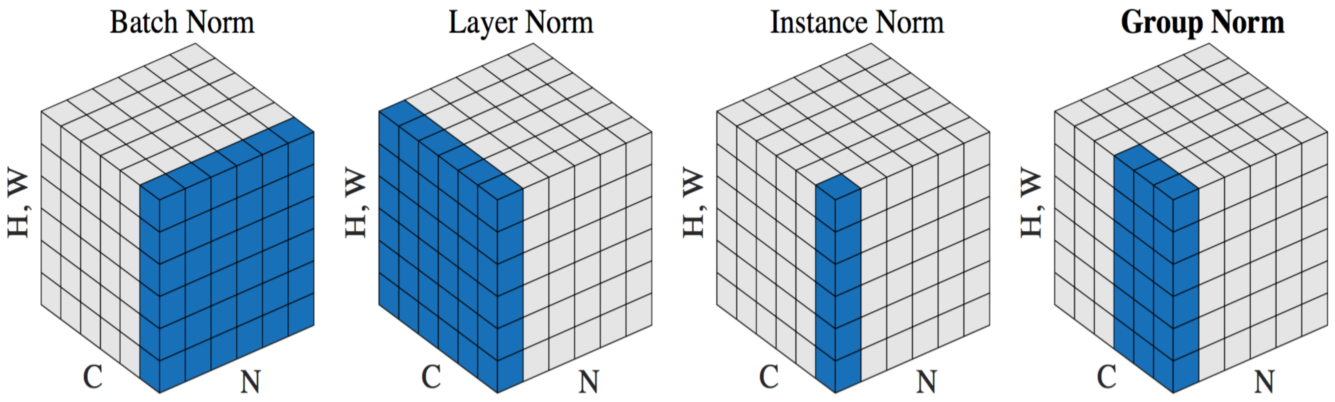
Group Normalization

- Group normalization computes the mean and std over **groups of channels** for each training example.
- In a way, group normalization is a combination of layer normalization and instance normalization.
  - Indeed, when we put all the channels into a single group, group normalization becomes layer normalization and when we put each channel into a different group it becomes instance normalization.
- γ and β are learnable *per-channel* affine transform parameter vectors of size `num_channels` if `affine` is `True`.
- In PyTorch: `torch.nn.GroupNorm()` (https://pytorch.org/docs/stable/generated/torch.nn.GroupNorm.html#torch.nn.GroupNorm).

In [7]:

```
# GroupNorm example
x = torch.randn(20, 6, 10, 10)  # [batch_size, channels, h, w]
# Put all 6 channels into a single group (equivalent with LayerNorm)
m = nn.GroupNorm(1, 6)
# Separate 6 channels into 6 groups (equivalent with InstanceNorm)
m = nn.GroupNorm(6, 6)
# Separate 6 channels into 3 groups
m = nn.GroupNorm(3, 6)  # [n_groups, in_channels]

# Activating the module
output = m(x)
print(f'output: {output.shape}')
print(list(m.parameters())[0].shape)
```

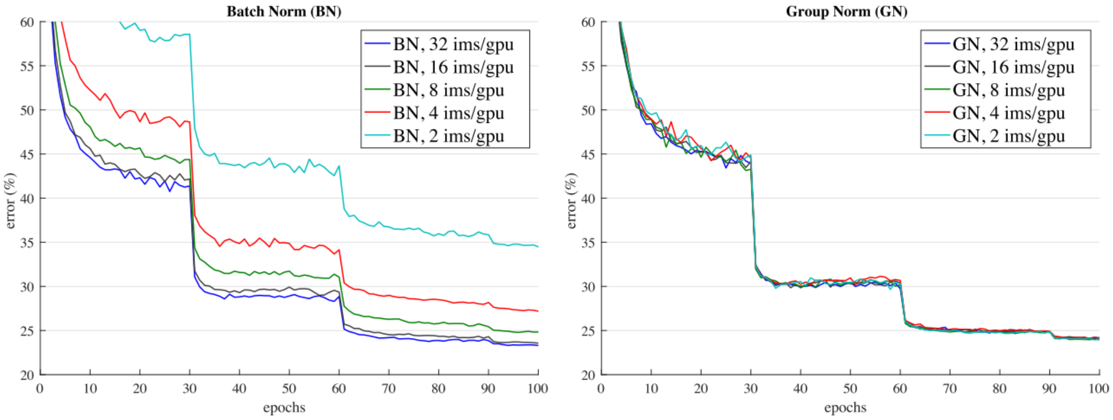
```
output: torch.Size([20, 6, 10, 10])
torch.Size([6])
```

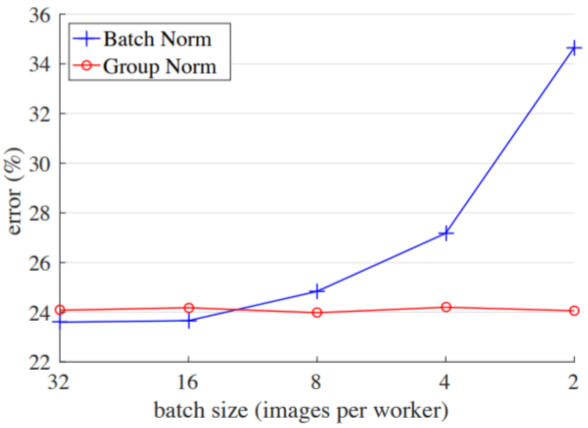


- Image Source (https://arxiv.org/pdf/1803.08494.pdf)

- Though **layer normalization** and **instance normalization** were both effective on RNNs and style transfer respectively, they were still inferior to **batch normalization** for image recognition tasks.
- **Group normalization** was able to achieve much closer performance to **batch normalization** with a batch size of 32 on ImageNet and outperformed it on smaller batch sizes.
- For tasks like object detection and segmentation that use much higher resolution images (and therefore cannot increase their batch size due to memory constraints), **group normalization** was shown to be a very effective normalization method.
- One of the implicit assumptions that **layer normalization** makes is that all channels are "equally important" when computing the mean.
    - This assumption is not always true in convolution layers. For instance, neurons near the edge of an image and neurons near the center of an image will have very different activation statistics.
    - This means that computing different statistics for different channels can give models much-needed flexibility. Channels in an image are not completely independent though, so being able to leverage the statistics of nearby channels is an advantage **group normalization** has over **instance normalization**.

The following figures illustrate how the batch size affects the performance of BatchNorm and GroupNorm on ImageNet classification:





Images Source (https://amaarora.github.io/2020/08/09/groupnorm.html)

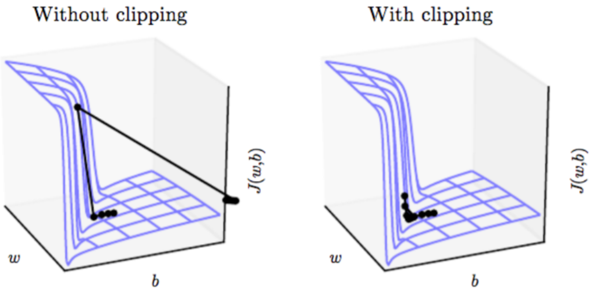
Vanishing/Exploding Gradients and Gradient Clipping

- Training a neural network can become unstable given the choice of error function and other hyper-parameters such as learning rate, or even the scale of the target variable.
- **Vanishing gradients**: When using certain activation functions, like the sigmoid function, in very deep neural networks, they squish a large input space into a small input space between 0 and 1. Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes smaller and smaller when backpropagating through the layers.
- **Exploding gradients**: large updates to weights during training can cause a numerical overflow or underflow often referred to as “exploding gradients.”
    - The problem of exploding gradients is more common with recurrent neural networks, such as LSTMs given the accumulation of gradients unrolled over hundreds of input time steps.
    - In practice, the weights can take on the value of an “NaN” or “Inf” when they overflow or underflow and for practical purposes the network will be useless from that point forward, forever predicting NaN values as signals flow through the invalid weights.

- The vanishing gradient problem is usually solved with changing the activation functions or using skip-connections (next section).
- For the **exploding gradient** problem, a common and relatively easy solution is to change the derivative of the error before propagating it backward through the network and using it to update the weights.
- Two approaches include **rescaling the gradients** given a chosen vector norm and **clipping gradient** values that exceed a preferred range. Together, these methods are referred to as “gradient clipping.”
- Basically, we prevent gradients from blowing up by rescaling them so that their norm is at most a particular value η. I.e., if $||g|| > \eta$, where g is the gradient, we set:

$$g \leftarrow \frac{\eta g}{||g||}.$$

- This biases the training procedure, since the resulting values won’t actually be the gradient of the cost function.
    - However, this bias can be worth it if it keeps things stable.



— Goodfellow et al., *Deep Learning*

- In PyTorch: [`torch.nn.utils.clip_grad_norm_` (https://pytorch.org/docs/stable/generated/torch.nn.utils.clip_grad_norm_.html)](https://pytorch.org/docs/stable/generated/torch.nn.utils.clip_grad_norm_.html)
    - This is an `inplace` operation as indicated by the `_` at the end of the function name.
- Usage example:

In [ ]:

```
clipping_value = 1 # arbitrary value of your choosing, typical values to check: [0.01, 0.05, 0.1, 0.5, 1.0]
outputs = model(data)
loss = loss_function(inputs, outputs)
optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), clipping_value) # gradient clipping, notice the location of this line
optimizer.step()
```

Skip Connections

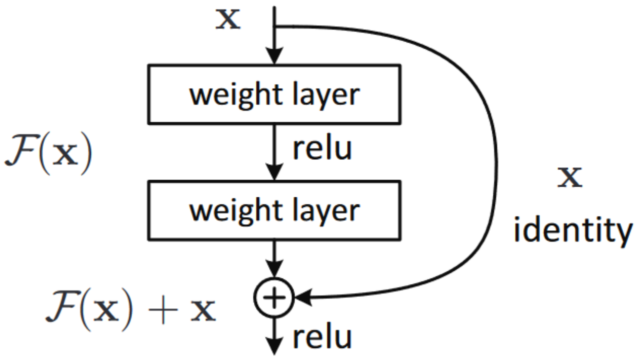
- **Skip connections**: skip some layers in the neural network and feeds the output of one layer as the input to the next layers (instead of only the next one).
- Skip connection is a standard module in many convolutional architectures.
- By using a skip connection, we provide an **alternative path for the gradient** (with backpropagation).
- It is experimentally validated that these additional paths are often beneficial for the **model convergence**.
- Using skip connections can mitigate the **vanishing gradient** problem (when the gradient becomes very small as we approach the earlier layers in a deep architecture).
- For *visual tasks*, such as semantic segmentation, optical flow estimation and etc.. there is some information that was captured in the initial layers and we would like to allow the later layers to also learn from them.
  - It has been observed that in earlier layers the learned features correspond to lower semantic information that is extracted from the input.
  - If we had not used the skip connection that information would have turned too abstract.

In general, there are two fundamental ways that one could use skip connections through different non-sequential layers:

- Addition as in residual architectures (the ResNet family).
- Concatenation as in densely connected architectures (the DenseNet family).

ResNet: Skip Connections via Addition

- The core idea is to backpropagate through the **identity function**, by just using a tensor addition.
- The gradient would simply be multiplied by one and its value will be maintained in the earlier layers.
- **Residual Networks (ResNets)**: stack these skip residual blocks and use the identity function to preserve the gradient.
- ResNet uses **short** skip-connections (they do not change the input dimension of the consecutive layer).


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

- Image Source (https://arxiv.org/abs/1512.03385)

Mathematically, we can represent the residual block, and calculate its partial derivative (gradient), given the loss function:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H}\frac{\partial H}{\partial x} = \frac{\partial L}{\partial H}\left(\frac{\partial F}{\partial x} + 1\right) = \frac{\partial L}{\partial H}\frac{\partial F}{\partial x} + \frac{\partial L}{\partial H}.$$

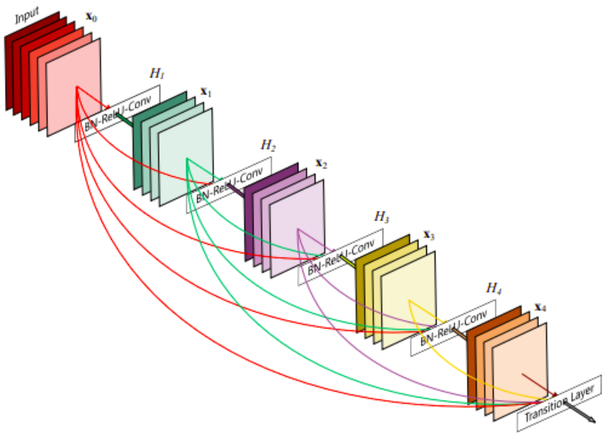
- Notice how the gradient is preserved ($\frac{\partial L}{\partial H}$).
- There is now an alternative path for the gradients.

In [ ]:

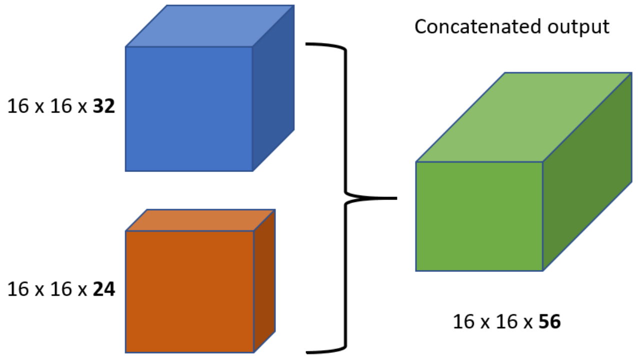
```
# Residual block
# full example:
# https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/02-intermediate/deep_residual_network/main.py
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample
        # downsample is a function/layer that matches the the dimensions of the input and output of the layer
        # an example of a downsampling function (channel-based downsample, can also use different padding/stride)
        downsample = nn.Sequential(nn.Conv2d(in_channels, out_channels, kernel_size=3,
                                             stride=stride, padding=1, bias=False),
                                   nn.BatchNorm2d(out_channels))

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out
```

DenseNet: Skip Connections via Concatenation

- For many tasks, there is low-level information shared between the input and output, and it would be desirable to pass this information directly across the net.
- The alternative way to achieve skip connections is by concatenation of previous feature maps.
- The most famous deep learning architecture that uses this technique is **DenseNet**.
- This architecture heavily uses feature concatenation so as to ensure **maximum information flow** between layers in the network.
- This is achieved by connecting (via concatenation) all layers directly with each other, as opposed to ResNets.
- Practically, what you are basically doing is concatenating the feature channel dimension, which leads to:
  - An enormous amount of feature channels on the last layers of the network.
  - More compact models.
  - Extreme feature reusability.
- [PyTorch Implementation (https://github.com/bamos/densenet.pytorch/blob/master/densenet.py)](https://github.com/bamos/densenet.pytorch/blob/master/densenet.py)



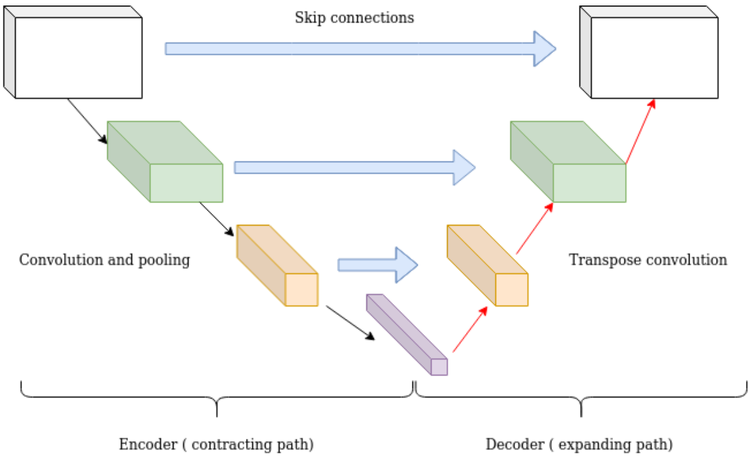
- [Image Source (https://arxiv.org/abs/1608.06993)](https://arxiv.org/abs/1608.06993)



- [Image Source (https://theaisummer.com/skip-connections)](https://theaisummer.com/skip-connections)

U-Nets: Long Skip Connections

- **Long skip connections** often exist in architectures that are symmetrical, where the spatial dimensionality is reduced in the *encoder* part and is gradually increased in the *decoder*.
- In the *decoder* part, one can increase the dimensionality of a feature map via transpose convolutional layers (also called up-convolution).
  - The transposed convolution operation forms the same connectivity as the normal convolution but in the backward direction.
- Mathematically, if we express convolution as a matrix multiplication, then transpose convolution is the reverse order multiplication ($B \times A$ instead of $A \times B$).
- The architecture of the encoder-decoder with long skip connections is often referred as **U-shape (Unet)**.
  - It is utilized for tasks that the prediction has the same spatial dimension as the input such as image segmentation, optical flow estimation, video prediction, etc.
- By introducing skip connections in the encoder-decoded architecture, fine-grained details can be recovered in the prediction.
- The skip-connection is **concatenation-based**.
- [PyTorch Implementation (https://github.com/milesial/Pytorch-UNet)](https://github.com/milesial/Pytorch-UNet)



- [Image Source (https://theaisummer.com/skip-connections)](https://theaisummer.com/skip-connections)

Hyperparmeter Tuning

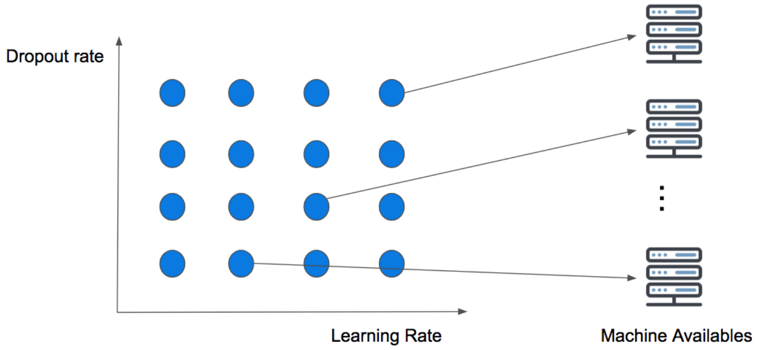
- Hyperparameters are external parameters set by the operator of the neural network – for example, selecting which activation function to use or the batch size used in training.
  - They are set manually with a pre-determined value before starting the training.
- Hyperparameters have a huge impact on the accuracy of a neural network, there may be different optimal values for different values, and it is non-trivial to discover those values.
- Common hyperparameters in deep learning include:
  - Number of hidden layers
  - Learning rate, learning rate schdeule
  - Activations function
  - Weights initialization
  - Dropout rate
  - Batch size
  - And many more...
- Hyperparameter tuning is always performed against an optimization metric or score, which is called the optimization objective. This is the metric you are trying to optimize when you try different hyperparameter values. Typically, the optimization metric is accuracy for classifications tasks, but it can also be the reconstruction error for autoencoders, or FID for GANs.
- We will discuss 4 tuning techniques and see how can we automate the process with **Optuna and PyTorch**.

Babysitting - Manual Hyperparameter Tuning

- Babysitting is also known as Trial & Error. This approach is 100% manual and the most widely adopted by researchers, students, and hobbyists.
- This is still commonly done, and experienced operators can “guess” parameter values that will achieve very high accuracy for deep learning models.
- **Pros**: Very simple and effective with skilled engineers.
- **Cons**: Not scientific, unknown if you have fully optimized hyperparameters.

Grid Search

- Grid search involves systematically testing multiple values of each hyperparameter, by automatically retraining the model for each value of the parameter.
  - For example, you can perform a grid search for the optimal batch size by automatically training the model for batch sizes between 10-100 samples, in steps of 20. The model will run 5 times and the batch size selected will be the one which yields highest accuracy.
- The process:
  - Define a grid on n dimensions, where each of these maps for an hyperparameter. e.g. $n = (\text{learning\_rate}, \text{dropout\_rate}, \text{batch\_size})$
  - For each dimension, define the range of possible values: e.g. $\text{batch\_size} = [4, 8, 16, 32, 64, 128, 256]$
  - Search for all the possible configurations and wait for the results to establish the best one.
- **Pros**: Maps out the problem space and provides more opportunity for optimization. Can be parallelized.
- **Cons**: Can be slow to run for large numbers of hyperparameter values. Doesn't take history into account. Curse of dimensionality (the more dimensions we add, the more the search will explode in time complexity). Doesn't focus on areas of higher benefit.



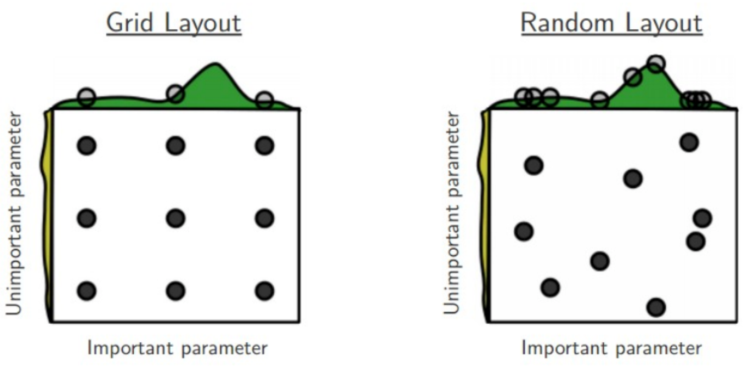
- [Image Source (https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/)](https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/)

In [ ]:

```
# example skeleton for grid search
lrs = [1e-1, 1e-2, 1e-3, 1e-4]  # learning rate
bss = [32, 64 ,128, 256]  # batch size
hidden_unitss = [64, 126, 256, 512]
n_layerss = [1, 2, 4, 8, 16]
for lr in lrs:
    for bs in bss:
        for hidden_units in hidden_unitss:
            for n_layers in n_layerss:
                # train model and save results
```

Random Search

- Random search - instead of testing systematically to cover “promising areas” of the problem space, it is preferable to test random values drawn from the entire problem space.
    - It was found that testing randomized values of hyperparameters is actually more effective than manual search or grid search.
- **Pros**: According to the study, provides higher accuracy with less training cycles, for problems with high dimensionality.
- **Cons**: Results are unintuitive, difficult to understand “why” hyperparameter values were chosen.
- A rule of thumb: **DON'T use Grid Search** if your searching space contains more than 3 to 4 dimensions. Instead, use Random Search, which provides a really good baseline for each searching task.



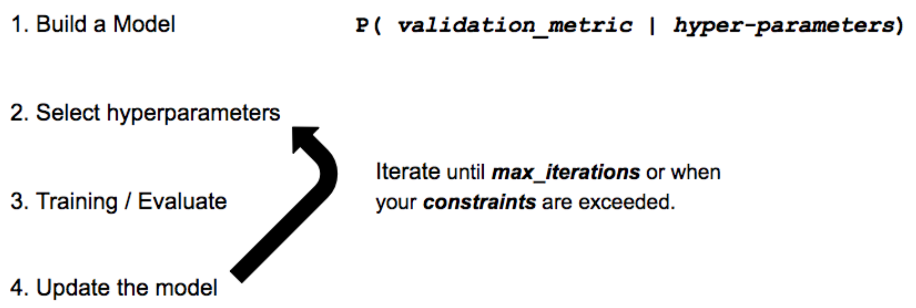
- [Image Source (http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf)](http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf)

In [ ]:

```
# example skeleton for random serach
lrs = [1e-1, 1e-2, 1e-3, 1e-4]  # learning rate
bss = [32, 64 ,128, 256]  # batch size
hidden_unitss = [64, 126, 256, 512]
n_layerss = [1, 2, 4, 8, 16]
max_experiments = 100
for i in range(max_experiments):
    lr = np.random.choice(lrs, p=None)  # p=None -> uniform distribution sampling
    bs = np.random.choice(bss, p=[0.4, 0.4, 0.1, 0.1])
    hidden_units = np.random.choice(hidden_unitss, p=None)
    n_layers = np.random.choice(n_layerss, p=None)
    # train model and save results
```

Bayesian Optimization

- Bayesian optimization is a technique which tries to approximate the trained model with different possible hyperparameter values. **It tries to predict the metrics we care about from the hyperparameters configuration**: $P(\mathrm{val\_acc}|\mathrm{hyper\_params})$.
- To simplify, Bayesian optimization trains the model with different hyperparameter values, and observes the function generated for the model by each set of parameter values. It does this over and over again, each time selecting hyperparameter values that are slightly different and can help plot the next relevant segment of the problem space.
- Similar to sampling methods in statistics, the algorithm ends up with a list of possible hyperparameter value sets and model functions, from which it predicts the optimal function across the entire problem set.
- **Pros**: The original study and practical experience from the industry shows that Bayesian optimization results in significantly higher accuracy compared to random search.
- **Cons**: Like random search, results are not intuitive and difficult to improve on, even by trained operators.
- Optuna can perform this kind of tuning as we will soon see.



1. Build a Model

P(*validation_metric* | *hyper-parameters*)

2. Select hyperparameters

3. Training / Evaluate

4. Update the model

Iterate until ***max_iterations*** or when your ***constraints*** are exceeded.

- [Image Source (https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/)](https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/)

Approach	ML	DL	Cost (Complexity)	History
Babysitting	Yes	Not Recommended	Low	Yes
Grid Search	Yes	Not Recommended	High	No
Random Search	Yes	Yes	Medium	No
Bayesian	Yes	Yes	Low-medium	Yes

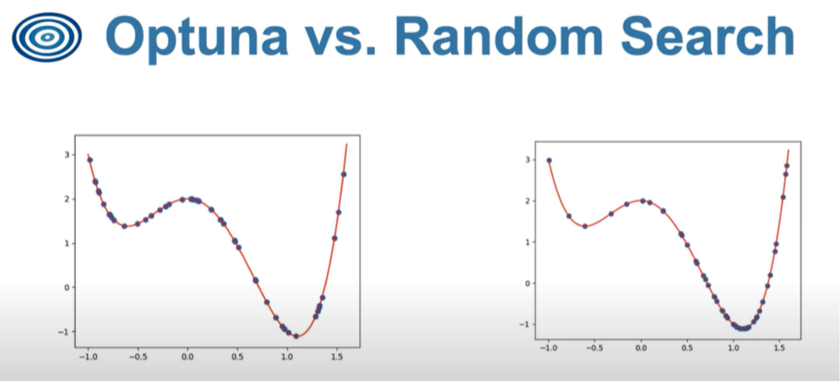
Hyperparameter Tuning with Optuna and PyTorch

- [Optuna (https://optuna.org/)](https://optuna.org/) is an open source hyperparameter optimization framework to automate hyperparameter search.
- It allows easy parallelization and quick visualization of the hyperparameter space.
- Installation:
  - Anaconda: `conda install -c conda-forge optuna`
  - pip: `pip install optuna`

- Optuna uses two strategies:
  - **Sampling Startegy** -what areas of hyperparameters to sample from, or, where to look?
  - **Pruning Strategy** - if a particular trial is not looking very promising, Optuna can terminate it early to provide more time to better trials.

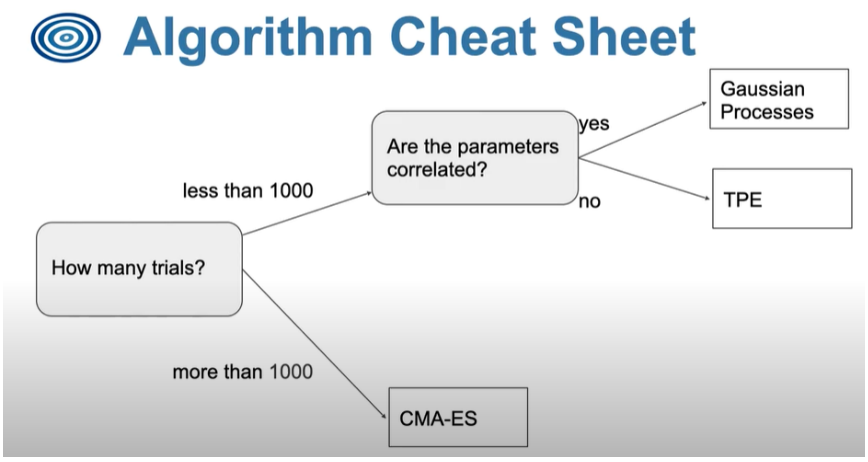
Samplers - Where to Look?

- Optuna focuses in on areas of interest using Bayesian fitting to find the places it has had the best results, and continue to look there.
- In the figure below, on the left you can see the areas Random Search chose to look, and on the right, the areas where Optuna chose to look.
    - Optuna chose to focus in on the area where the objective had the best results.



Optuna vs. Random Search

- Types of samplers:
    - Model-based samplers
        - **Tree-Structured Parzen Estimator (TPE)** - bayesian optimization based on kernel fitting (this is the default for Optuna).
        - **Gaussian Processes (GP)** - bayesian optimization based on Gaussian processes.
        - **Covariance matrix adaptation evolution strategy (CMA-ES)** - meta-heuristics algorithm for continuous space.
    - Other methods:
        - Random Search
        - Grid Search
        - User-defined algorithm



Algorithm Cheat Sheet

How many trials?

less than 1000

Are the parameters correlated?

yes

Gaussian Processes

no

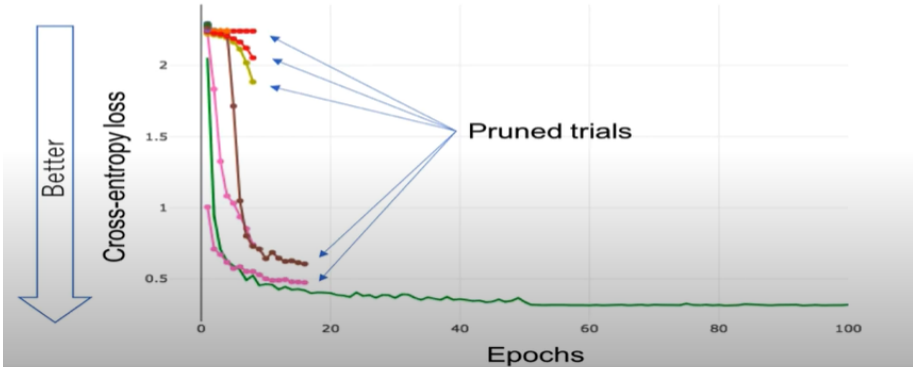
TPE

more than 1000

CMA-ES

Pruners - Stopping Trials Early

- Stop unpromising trials based on learning curves.
- Two startegies:
  - **Median Pruning**
  - **Sucessive Halving** (works better usually)



PyTorch Example

- Optuna main terms:
  - `study` - the experiment, e.g. "MNIST-FC".
  - `trial` - as the name suggests, a trial is a run of your training algorithm with the current selected hyperparameters.
  - `direction` - should the objective be maximized (e.g. accuracy) or minimized (e.g. error).
- A template code looks like this:

In [ ]:

```
def objective(trial):
    """
    YOUR CODE HERE
    """
    return evaluation_score

study = optuna.create_study(study_name="exp_name", direction="maximize"/"minimize", sampler=optuna.samplers.TPESampler())
study.optimize(objective, n_trials=NUMBER_OF_TRIALS)
```

In [2]:

```
# some definitions
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
batch_size = 128
classes = 10
epochs = 10
log_interval = 10
n_train_examples = batch_size * 30
n_valid_examples = batch_size * 10
```

In [3]:

```
def get_mnist():
    # Load MNIST dataset.
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./datasets', train=True, download=True, transform=transforms.ToTensor()),
        batch_size=batch_size,
        shuffle=True,
    )
    valid_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./datasets', train=False, transform=transforms.ToTensor()),
        batch_size=batch_size,
        shuffle=True,
    )

    return train_loader, valid_loader
```

- Next, we define a function that builds the model according to the current trial's hyperparameters.
- Based on the possible hyperparameter type (float, int, categorical...) we use `trial.suggest_float()` , `trial.suggest_int()` , to let optuna choose the hyperparameter for the current trial.
  - For each hyperparameter we need to specify a range of possible values for Optuna to choose from.
- We also give an informative name for each hyper-parameter in each trial.

In [4]:

```
def define_model(trial):
    # We optimize the number of layers, hidden units and dropout ratio in each layer.
    n_layers = trial.suggest_int("n_layers", 1, 3)  # number of layers will be between 1 and 3
    layers = []

    in_features = 28 * 28
    for i in range(n_layers):
        out_features = trial.suggest_int("n_units_l{}".format(i), 4, 128)  # number of units will be between 4 and 128
        layers.append(nn.Linear(in_features, out_features))
        layers.append(nn.ReLU())
        p = trial.suggest_float("dropout_l{}".format(i), 0.2, 0.5)  # dropout rate will be between 0.2 and 0.5
        layers.append(nn.Dropout(p))

        in_features = out_features
    layers.append(nn.Linear(in_features, classes))
    layers.append(nn.LogSoftmax(dim=1))

    return nn.Sequential(*layers)
```

- Next, we define the objective function that will run the model we defined above.

In [5]:

```
def objective(trial):

    # Generate the model.
    model = define_model(trial).to(device)

    # Generate the optimizers.
    lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)  # log=True, will use log scale to interplolate between lr
    optimizer_name = trial.suggest_categorical("optimizer", ["Adam", "RMSprop", "SGD"])
    optimizer = getattr(optim, optimizer_name)(model.parameters(), lr=lr)
    # alternative version
    # optimizer = trial.suggest_categorical("optimizer", [optim.Adam, optim.RMSprop, optim.SGD])

    # Get the MNIST dataset.
    train_loader, valid_loader = get_mnist()

    # Training of the model.
    for epoch in range(epochs):
        model.train()
        for batch_idx, (data, target) in enumerate(train_loader):
            # Limiting training data for faster epochs.
            if batch_idx * batch_size >= n_train_examples:
                break

            data, target = data.view(data.size(0), -1).to(device), target.to(device)

            optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            loss.backward()
            optimizer.step()

        # Validation of the model.
        model.eval()
        correct = 0
        with torch.no_grad():
            for batch_idx, (data, target) in enumerate(valid_loader):
                # Limiting validation data.
                if batch_idx * batch_size >= n_valid_examples:
                    break
                data, target = data.view(data.size(0), -1).to(device), target.to(device)
                output = model(data)
                # Get the index of the max log-probability.
                pred = output.argmax(dim=1, keepdim=True)
                correct += pred.eq(target.view_as(pred)).sum().item()

        accuracy = correct / min(len(valid_loader.dataset), n_valid_examples)

        # report back to Optuna how far it is (epoch-wise) into the trial and how well it is doing (accuracy)
        trial.report(accuracy, epoch)

        # then, Optuna can decide if the trial should be pruned
        # Handle pruning based on the intermediate value.
        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()

    return accuracy
```

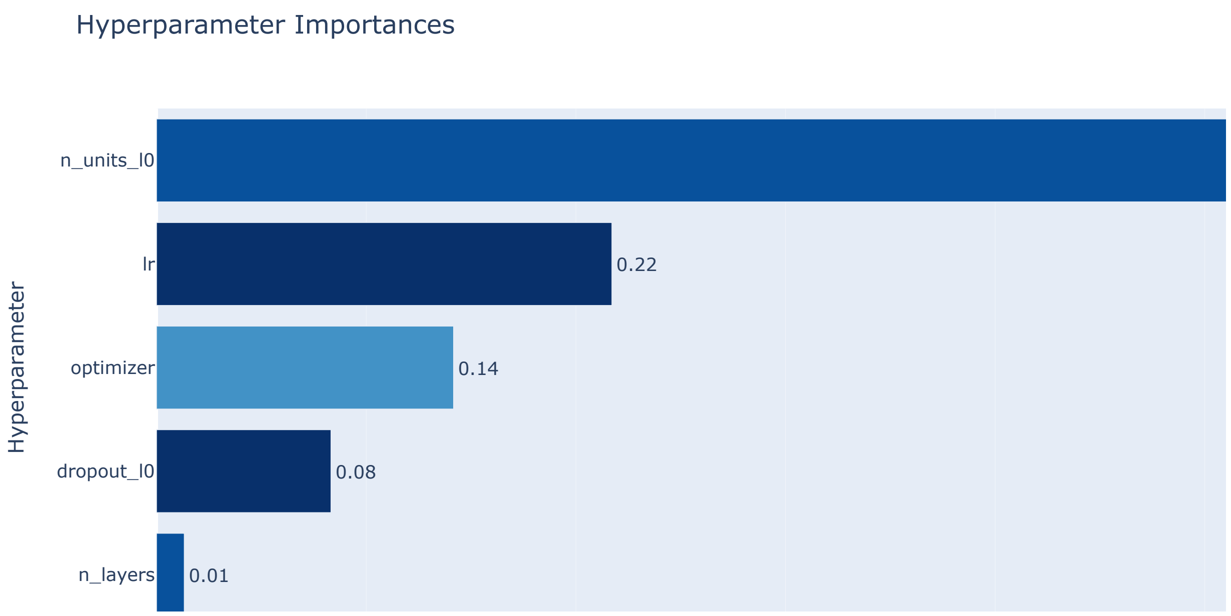
In [ ]:

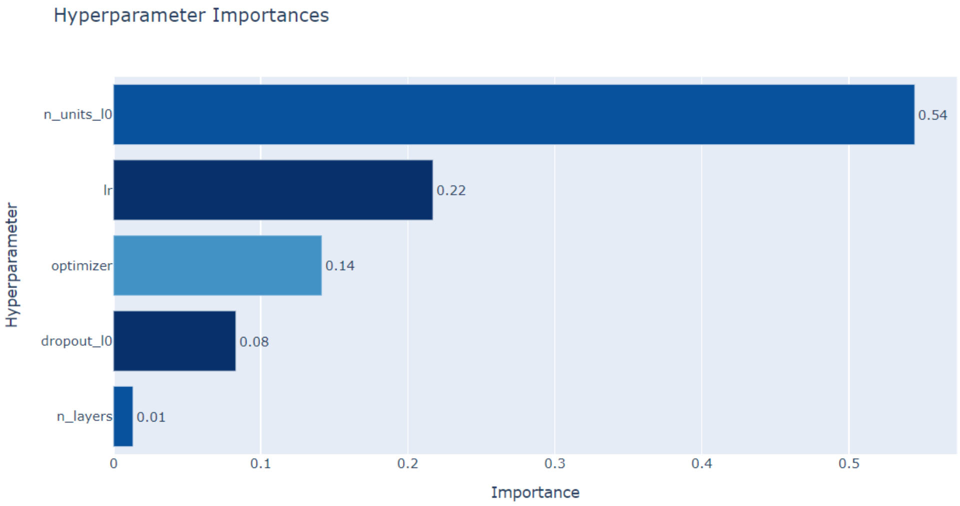
```
# now we can run the experiment
sampler = optuna.samplers.TPESampler()
study = optuna.create_study(study_name="mnist-fc", direction="maximize", sampler=sampler)
study.optimize(objective, n_trials=100, timeout=600)

pruned_trials = [t for t in study.trials if t.state == optuna.trial.TrialState.PRUNED]
complete_trials = [t for t in study.trials if t.state == optuna.trial.TrialState.COMPLETE]

print("Study statistics: ")
print("  Number of finished trials: ", len(study.trials))
print("  Number of pruned trials: ", len(pruned_trials))
print("  Number of complete trials: ", len(complete_trials))

print("Best trial:")
trial = study.best_trial

print("  Value: ", trial.value)

print("  Params: ")
for key, value in trial.params.items():
    print("    {}: {}".format(key, value))
```

Hyperparameter Importances

- You can actually get a list of the most effective hyperparameters based on completed trials in a given study.

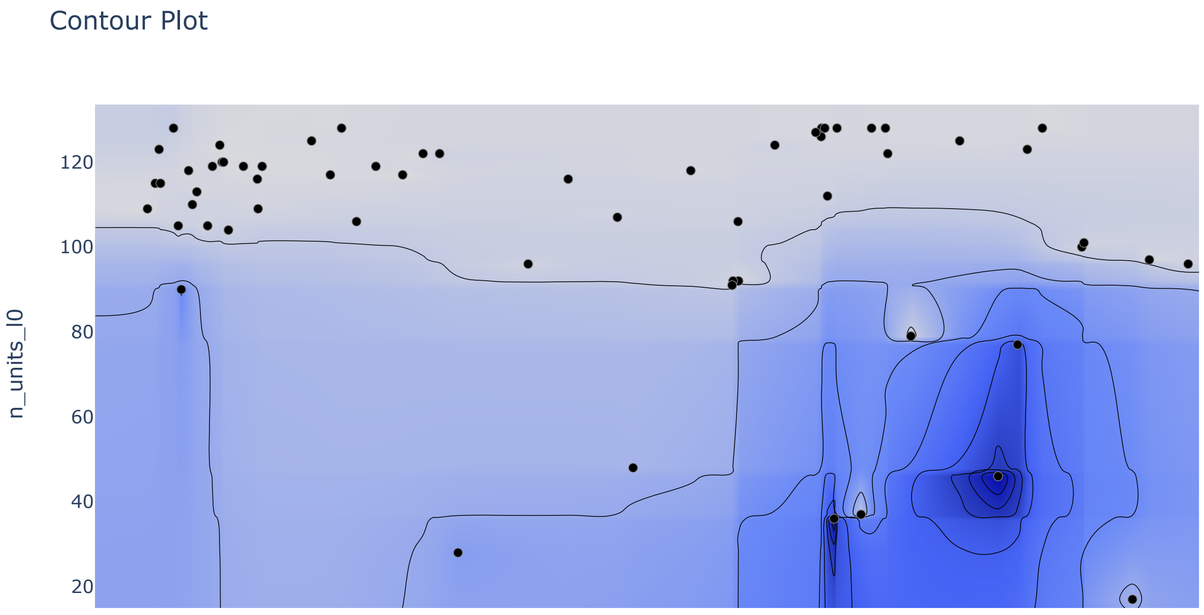
```
In [7]: optuna.visualization.plot_param_importances(study)
```

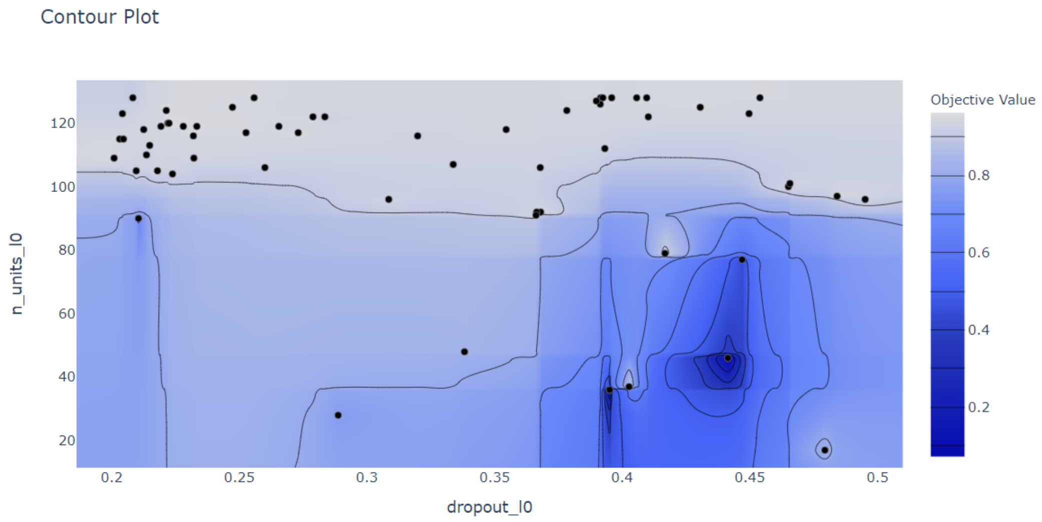




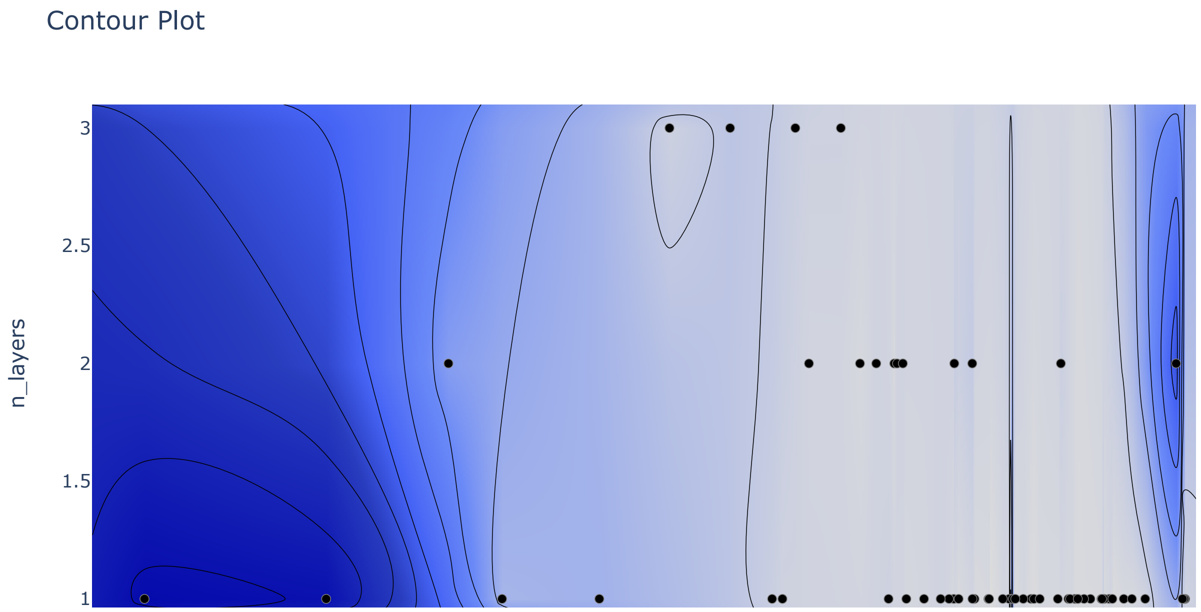
Visualizing the Search Space

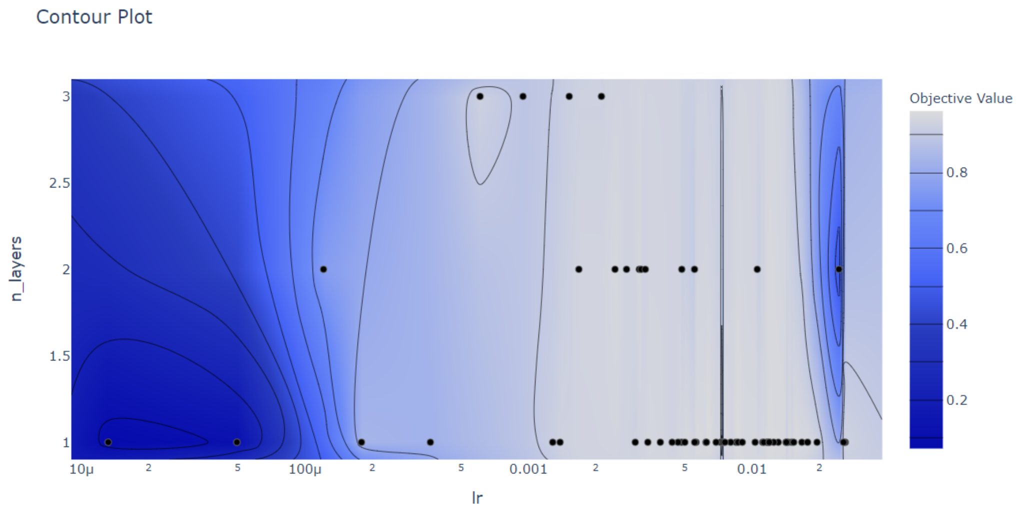
```
In [8]: optuna.visualization.plot_contour(study, params=["n_units_l0", "dropout_l0"])
```





```
In [9]: optuna.visualization.plot_contour(study, params=["n_layers", "lr"])
```





Recommended Videos

Warning!

- These videos do not replace the lectures and tutorials.
- Please use these to get a better understanding of the material, and not as an alternative to the written material.

Video By Subject

- Batch Normalization - Normalizing Activations in a Network (C2W3L04) (https://www.youtube.com/watch?v=tNIpEZLv_eg)
- Layer, Instance, Group Normalization - Layer, Instance, Group Normalization (https://www.youtube.com/watch?v=NE61nLoM-Fo)
- Vanishing/Exploding Gradients - Vanishing & Exploding Gradient explained (https://www.youtube.com/watch?v=qO_NLVjD6zE)
- Skip-Connections - C4W2L03 Resnets (https://www.youtube.com/watch?v=ZILIbUvp5lk)
- DenseNet - Henry AI Labs - DenseNets (https://www.youtube.com/watch?v=_8zx4T1Wcmg)
- Optuna - Auto-Tuning Hyperparameters with Optuna and PyTorch (https://www.youtube.com/watch?v=P6NwZVl8ttc)

Credits

- Icons made by Becris (https://www.flaticon.com/authors/becris) from www.flaticon.com (https://www.flaticon.com/)
- Icons from Icons8.com (https://icons8.com/) - https://icons8.com (https://icons8.com)

- Nikolas Adaloglou - Intuitive Explanation of Skip Connections in Deep Learning (https://theaisummer.com/skip-connections)
- An Overview of Normalization Methods in Deep Learning (https://mlexplained.com/2018/11/30/an-overview-of-normalization-methods-in-deep-learning/)
- Jason Brownlee - How to Avoid Exploding Gradients With Gradient Clipping (https://machinelearningmastery.com/how-to-avoid-exploding-gradients-in-neural-networks-with-gradient-clipping/)
- MissingLink - Hyperparameters: Optimization Methods and Real World Model Management (https://missinglink.ai/guides/neural-network-concepts/hyperparameters-optimization-methods-and-real-world-model-management/)
- FloydHub - Practical Guide to Hyperparameters Optimization for Deep Learning Models (https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/)