Outline

- Python Operators
- Designing Training Procedures
- Introduction to Convolutional Neural Nets (CNNs)
- Exercise: MNIST Classification with CNN

PyTorchOperators and Layers

PyTorch Operators/Layers

- Activation Functions
- Normalization
- Dropout
- Loss Functions

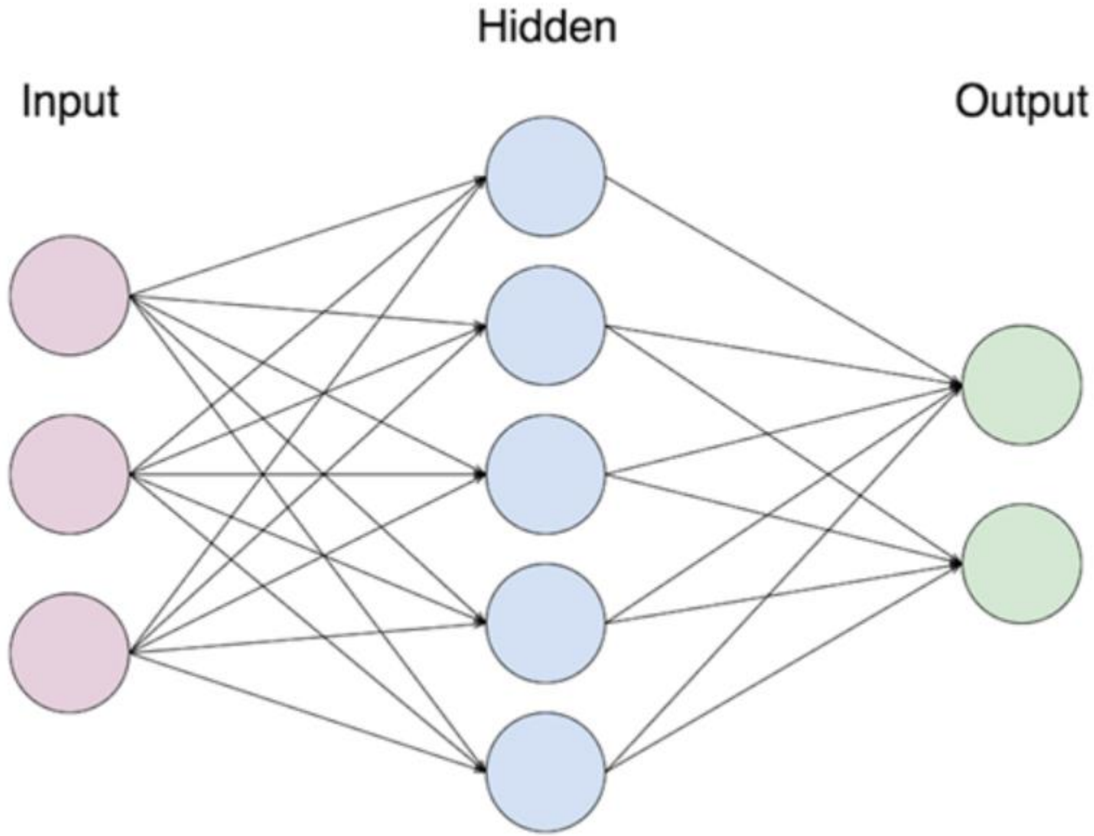
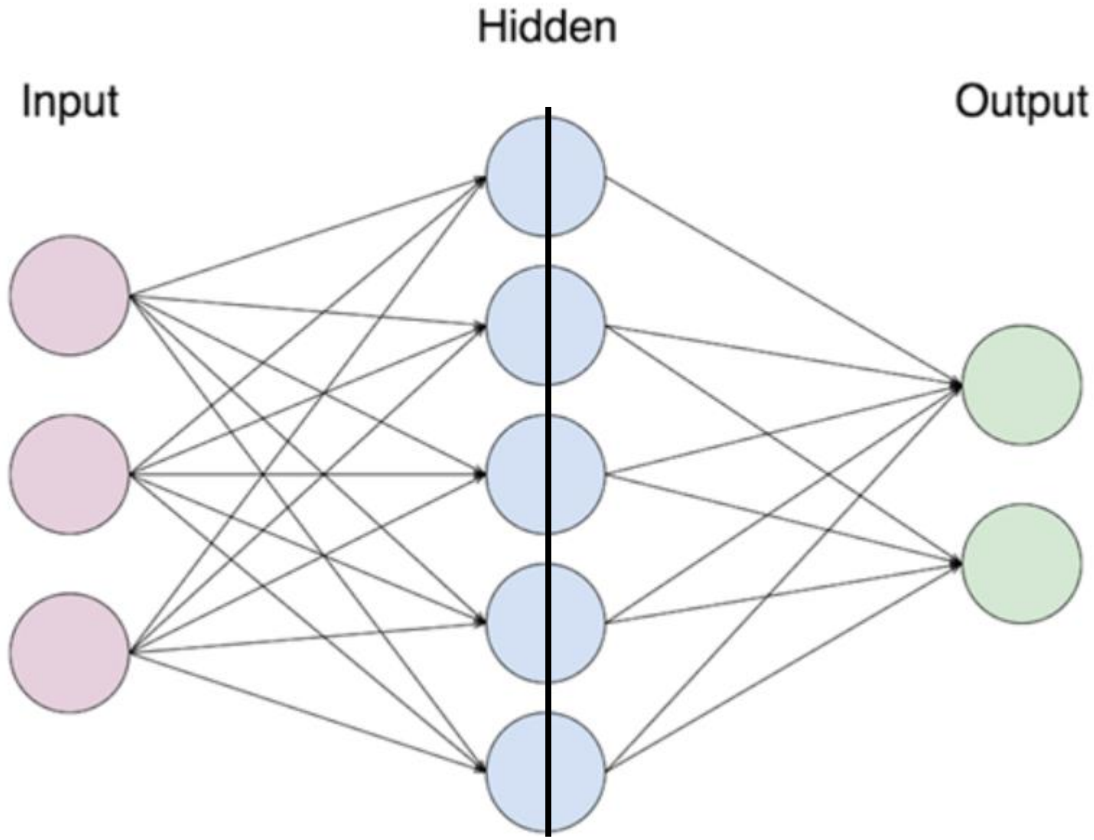


Image source: https://towardsdatascience.com/step-by-step-guide-to-building-your-own-neural-network-from-scratch-df64b1c5ab6e

PyTorch Operators/Layers

- Activation Functions
- Normalization
- Dropout
- Loss Functions

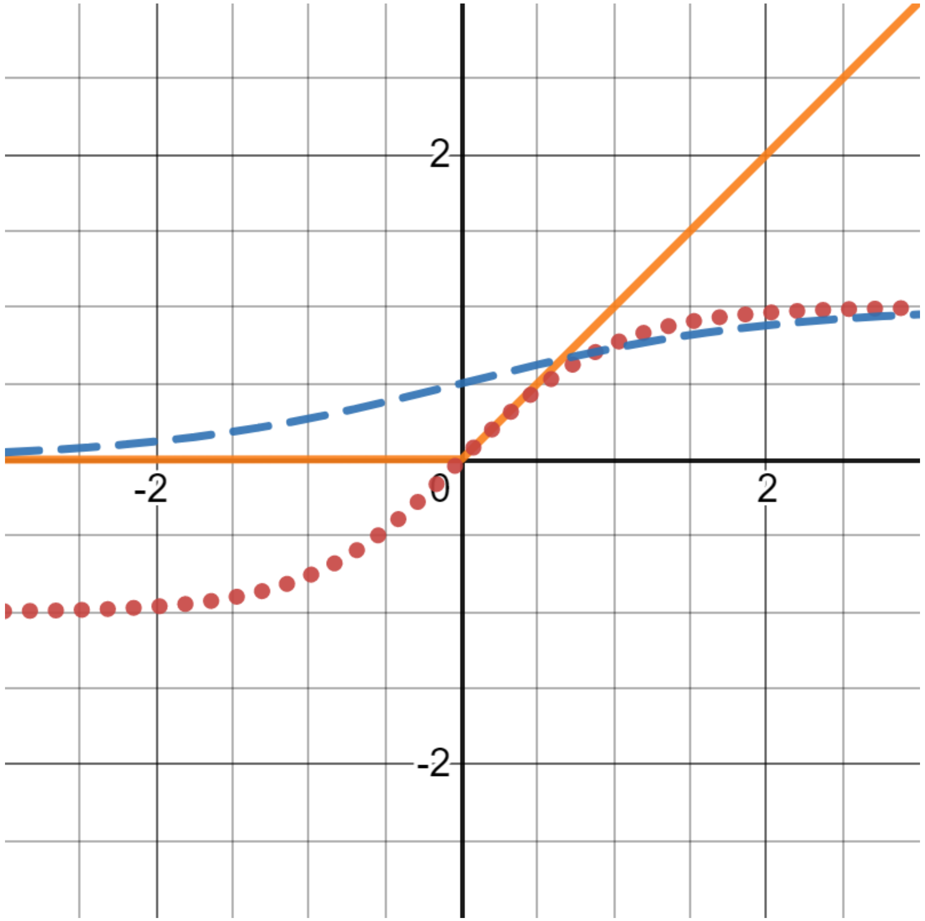


Activation Function

Image source: https://towardsdatascience.com/step-by-step-guide-to-building-your-own-neural-network-from-scratch-df64b1c5ab6e

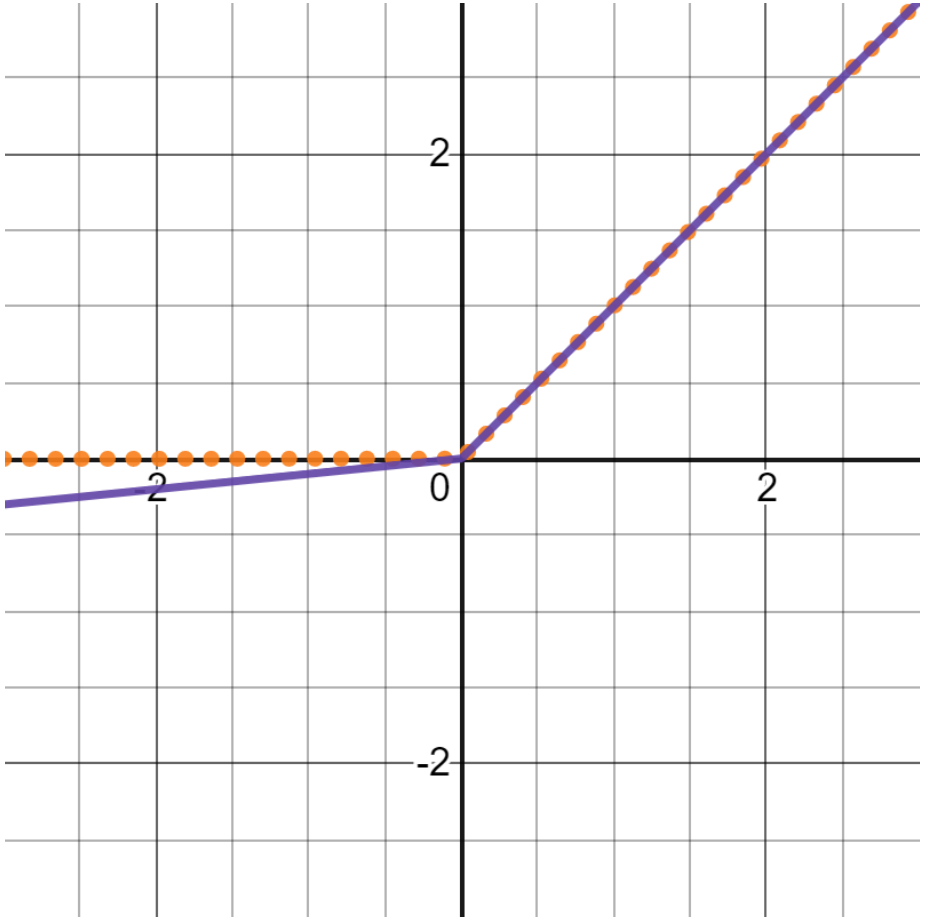
Activation Functions

- Non-linear functions performed by neurons
- ReLU - Rectified Linear Unit (nn.ReLU)
  - $y \geq 0$
- Tanh (nn.tanh)
  - $-1<y<1$
  - nn.Tanh
- Sigmoid (nn.Sigmoid)
  - $0<y<1$



Activation Functions

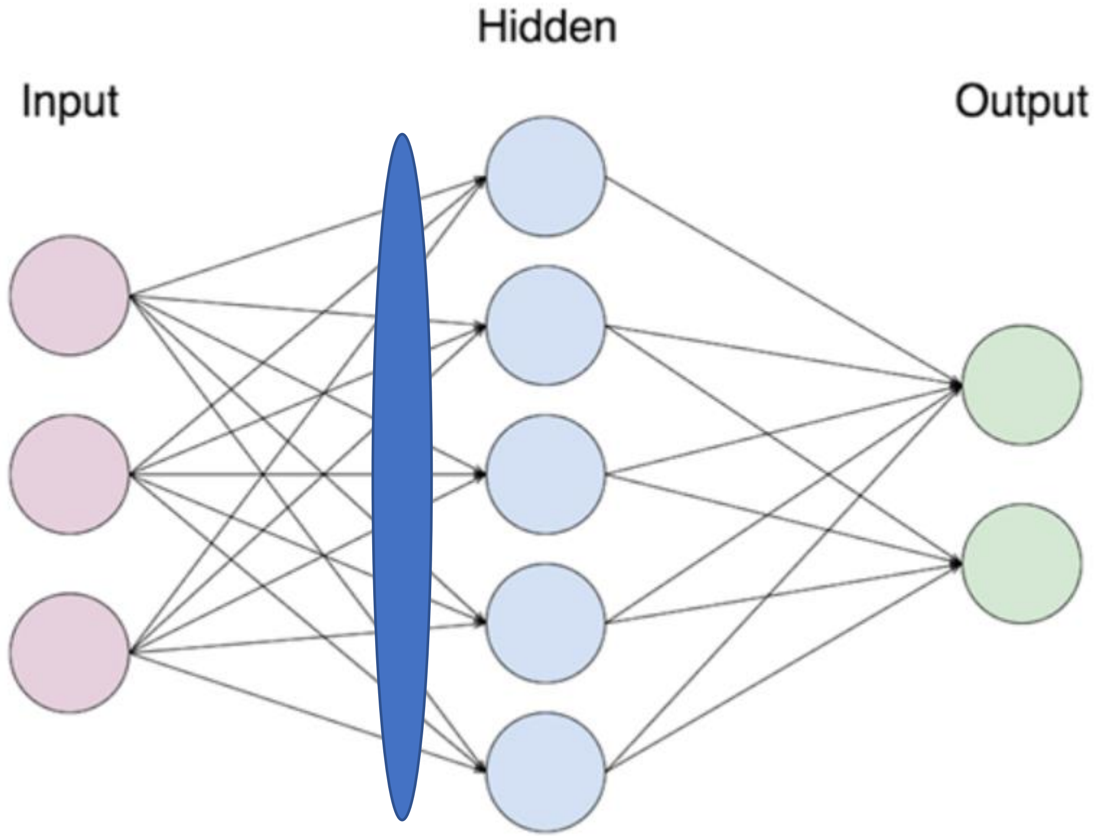
- Leaky ReLU
  - Similar to ReLU, but has non-zero values for negative x
  - Takes argument *negative_slope*, which determines the slope for x<0.
- For full list of activation functions, see: https://pytorch.org/docs/stable/nn.html



Leaky ReLU with negative slope = 0.1

Python Operators/Layers

- Activation Functions
- Normalization
- Dropout
- Loss Functions



Normalization

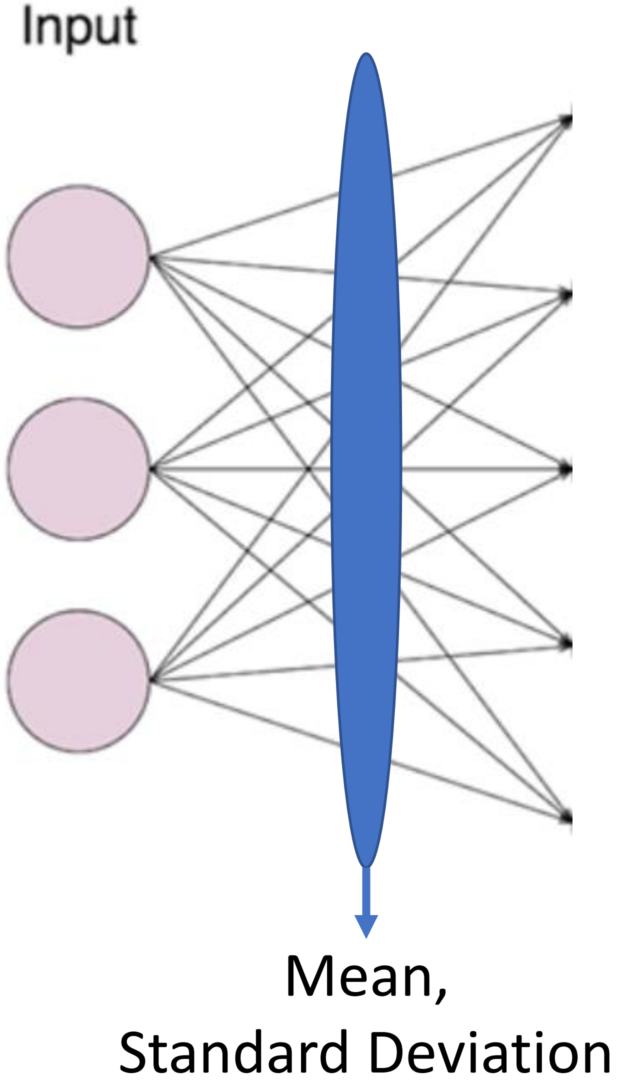
Image source: https://towardsdatascience.com/step-by-step-guide-to-building-your-own-neural-network-from-scratch-df64b1c5ab6e

Input Normalization: Batch Normalization

- Normalizes input into each layer for each training mini-batch
- Addresses issue of shifting input distributions over training
- Inputs:
  - num_features: Number of features in the input vector
  - eps: numerical stability parameter

Example:

```
b = torch.nn.BatchNorm1D(100)
input = torch.rand(50, 100)
output = b(input)
```



Input Normalization

Other normalization procedures include:

- Layer Norm: Transposes Batch Norm. Normalizes over all summed inputs to a layer
  - https://arxiv.org/abs/1607.06450
- Group Norm: Normalizes by grouped channels instead of batches
  - https://arxiv.org/abs/1803.08494

Python Operators/Layers

- Activation Functions
- Normalization
- Dropout
- Loss Functions

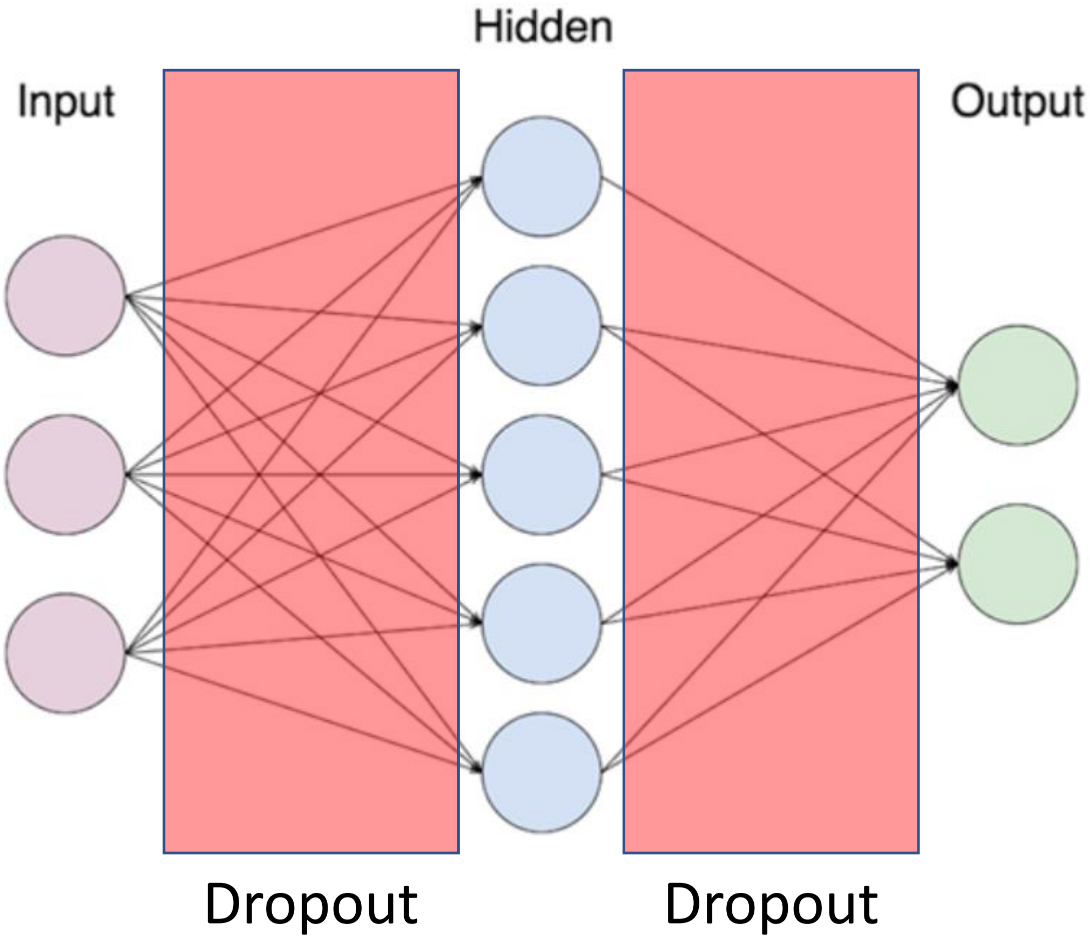
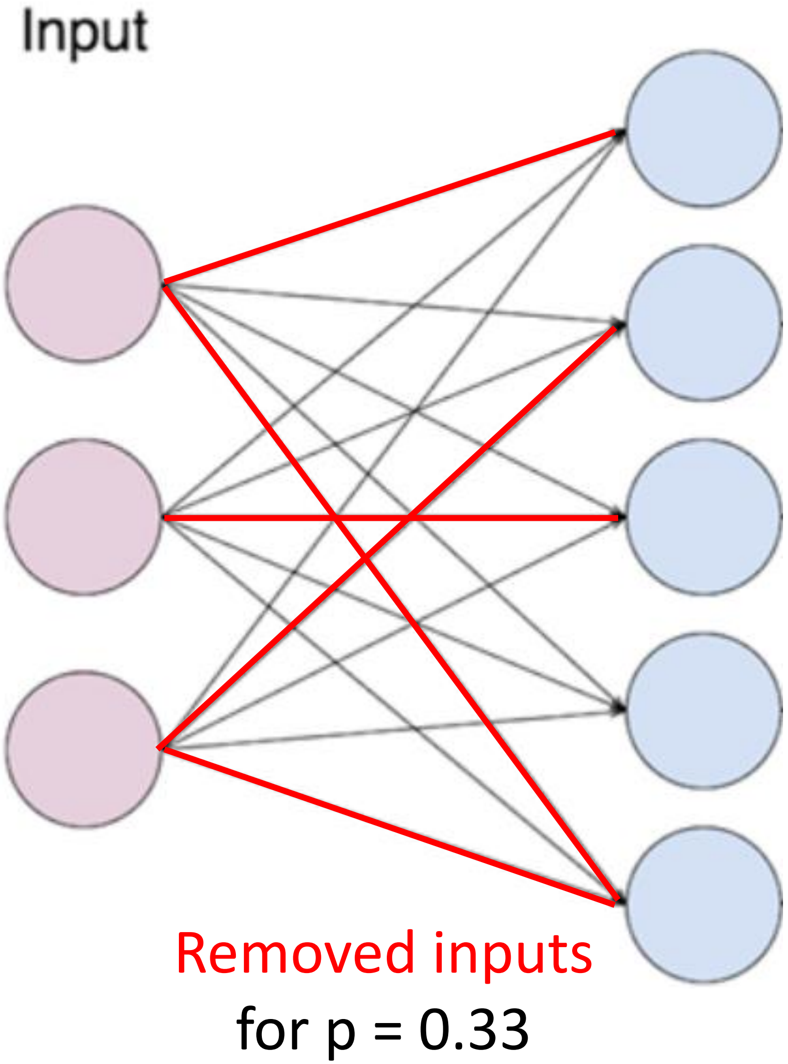


Image source: https://towardsdatascience.com/step-by-step-guide-to-building-your-own-neural-network-from-scratch-df64b1c5ab6e

Dropout

- Randomly zeroes some elements of input tensor with probability *p*
- Effective technique for regularization
- Outputs scaled by 1/1-p
- Treated as identity during evaluation



Python Operators/Layers

- Activation Functions
- Normalization
- Dropout
- Loss Functions

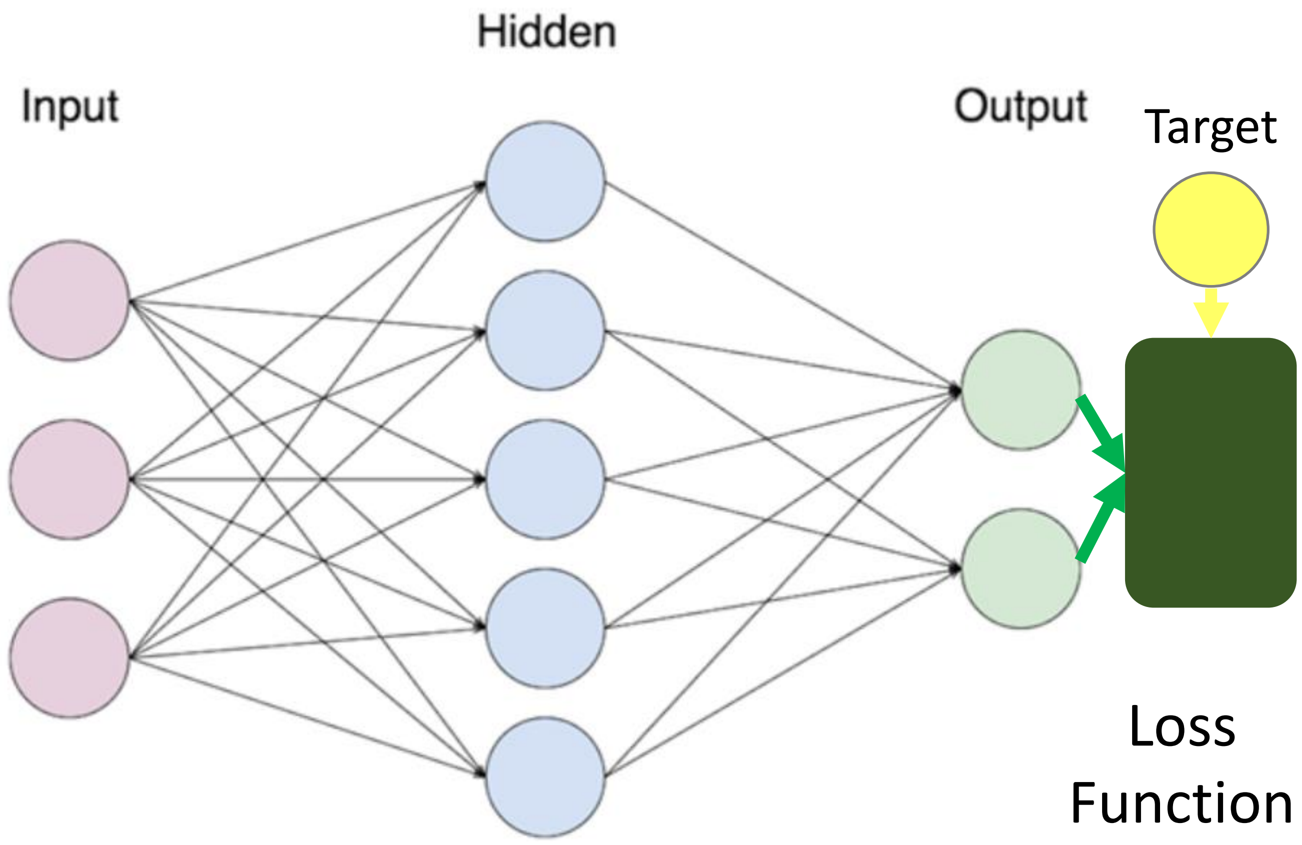
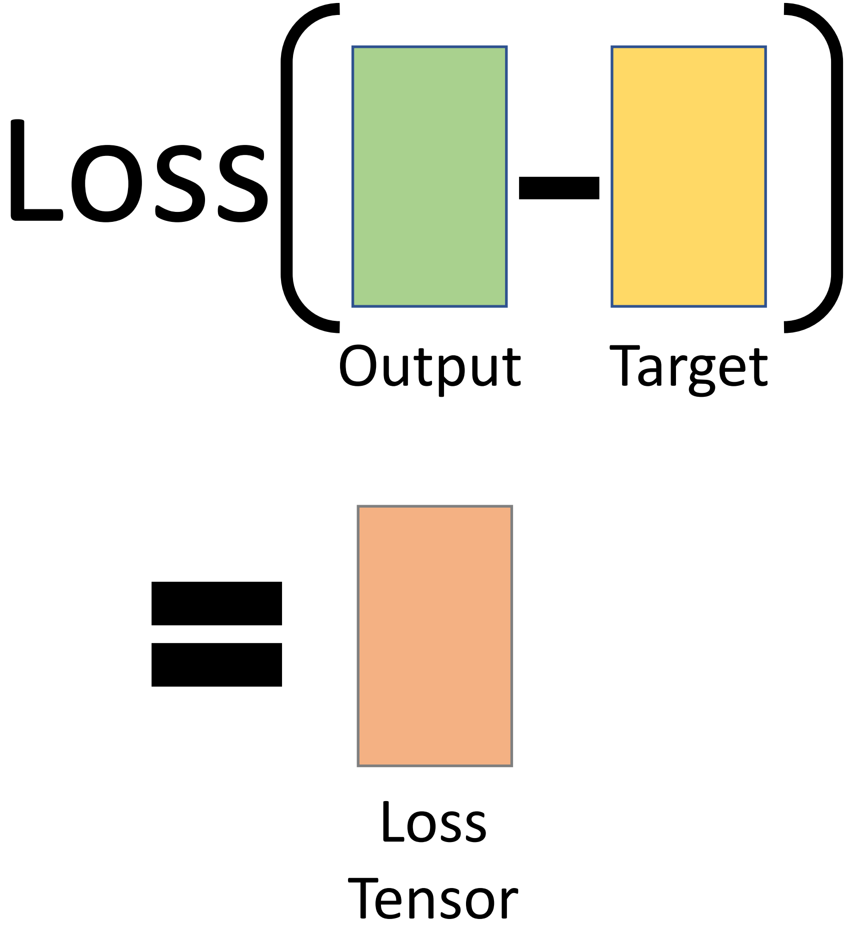


Image source: https://towardsdatascience.com/step-by-step-guide-to-building-your-own-neural-network-from-scratch-df64b1c5ab6e

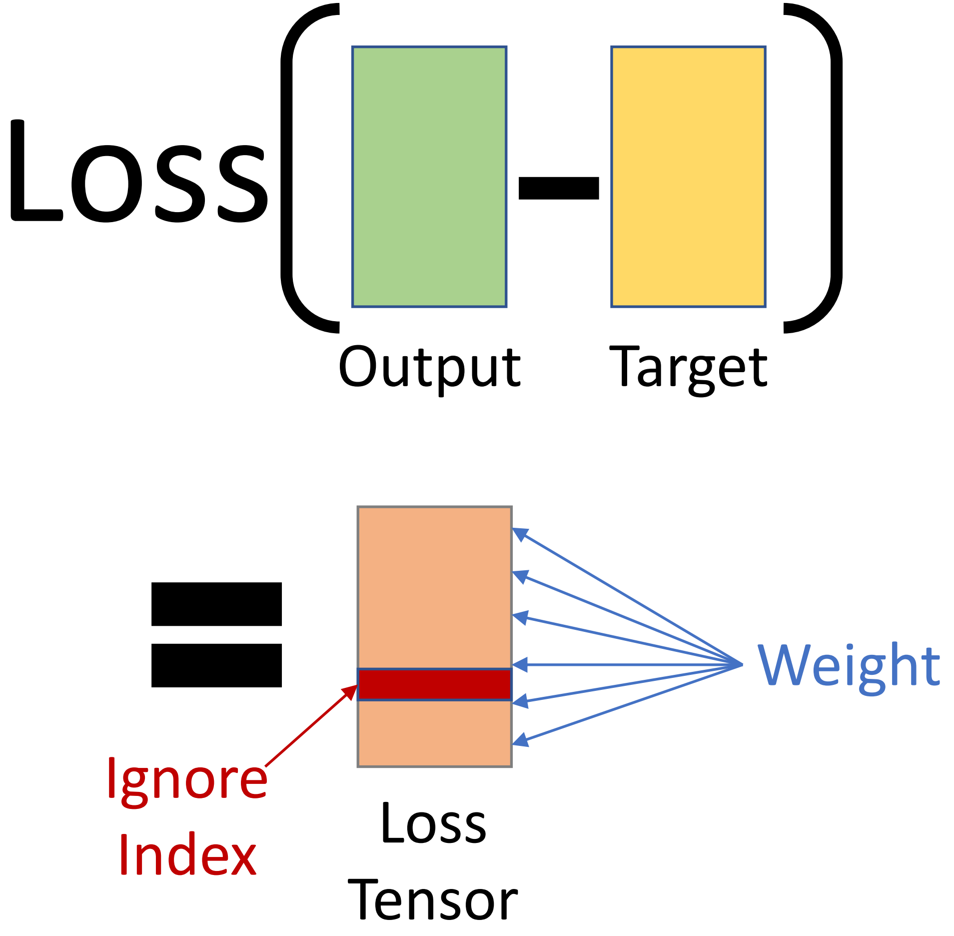
Loss Functions

- Loss function parameters:
  - Reduction: how the output will be reduced in dimension:
    - None: Gives entire Loss Tensor with no reduction over batches
    - Sum: Takes sum of the loss tensor across batches, returning a single number
    - Mean: Same as sum, but divides by the number of batches to get the mean



Loss Functions – Cross Entropy Loss

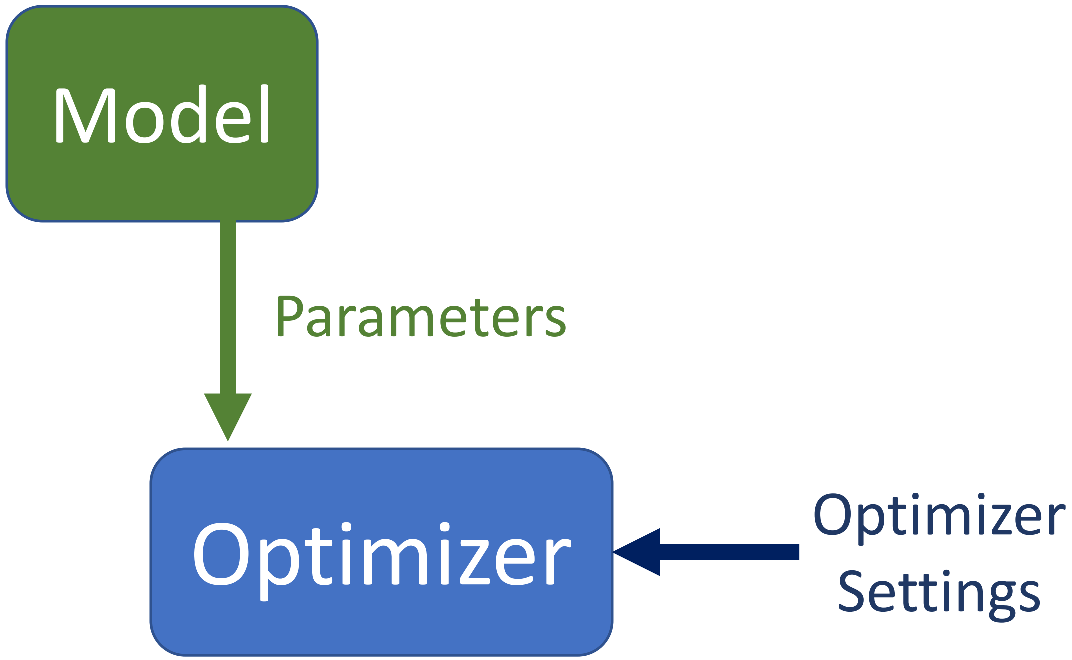
- Cross Entropy Parameters
  - Weight
    - 1D tensor assigning weights to each class, which is helpful if you have an unbalanced training set
  - ignore_index
    - Specifies a target value that is ignored and does not contribute to the input gradient



Designing Training Procedures

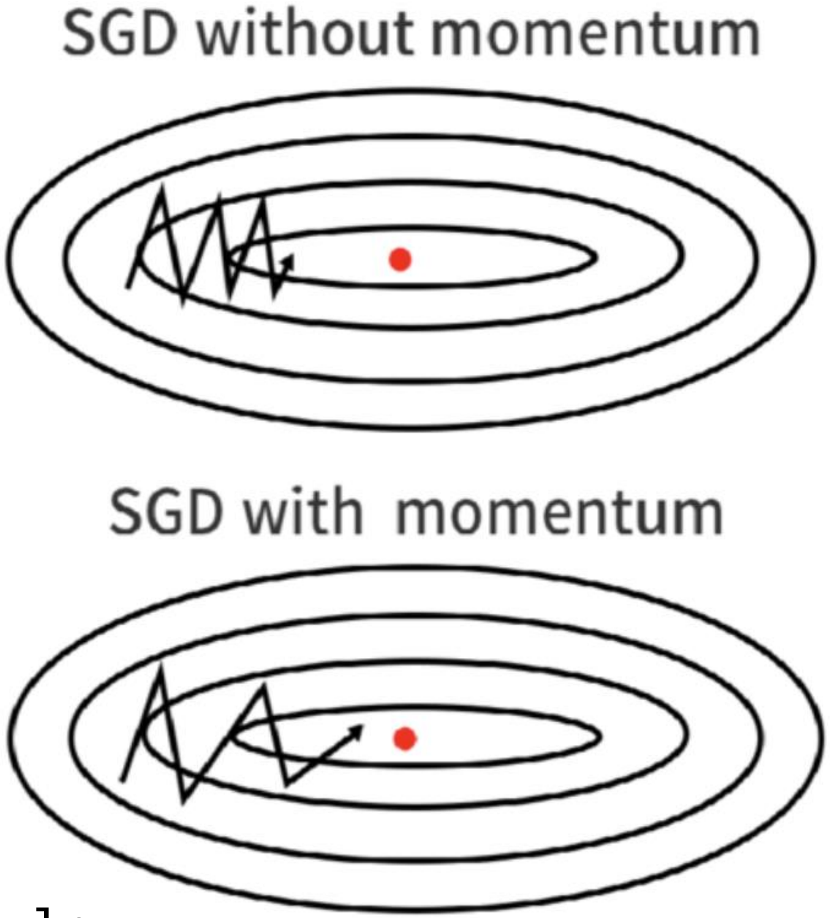
Optimizer Initialization

- Parameters:
  - Should be iterable containing parameters to optimize
  - E.g., model.parameters() or [var1, var2]
  - Parameters must be defined BEFORE the optimizer
- Optimizer Settings
  - Learning rate, weight decay, etc.



Optimizers

- Stochastic Gradient Descent (`torch.optim.SGD`)
  - params: Model parameters
  - lr: Learning rate (required)
  - momentum: momentum factor (default: 0)
  - weight_decay: (default: 0)



Example: `torch.optim.SGD(model.parameters(), lr = 0.001, momentum = 0.2, weight_decay = 0.1)`

Optimizers

- Adam (`torch.optim.Adam`)
    - params: Model parameters
    - lr: Learning rate (default: 0.001)
    - betas: coefficients (tuple) used for computing running averages of gradient and its square (default: (0.9, 0.999))
    - eps: term added to denominator to improve numerical stability (default: 1e-8)
    - weight_decay: (default: 0)

Example:

```
torch.optim.Adam(model.parameters(),
lr = 0.01, betas = (0.95, 0.998),
eps = 1e-7)
```

Other Common Optimizers

- AdaDelta (`torch.optim.Adadelta`)
  - Precursor to Adam which uses first-order estimates to adapt learning rate
- Adamax (`torch.optim.Adamax`)
  - Variant on Adam based on infinity norm
- RMSProp (`torch.optim.RMSprop`)
  - Take the square root of the gradient average before adding epsilon to normalization of LR

Introduction to Convolutional Neural Nets (CNNs)

Convolutional Layers

- Convolutional Layers use a moving frame (kernel) to process input
- Helpful for learning local features
- Parameters of the kernel are learned during training

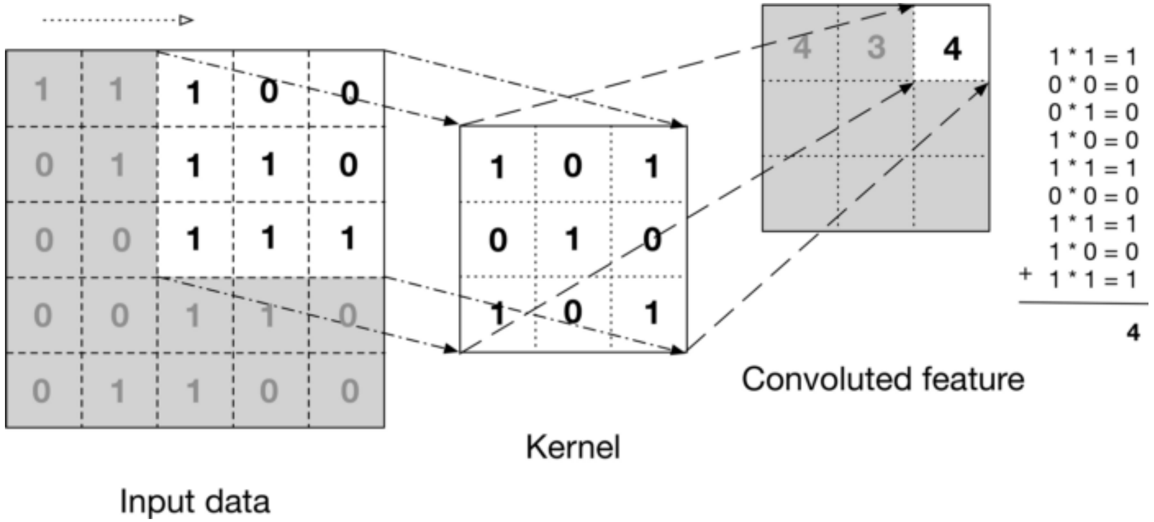


Image source: https://medium.com/machine-learning-for-li/different-convolutional-layers-43dc146f4d0e

Convolutional Layers: Arguments

- in_channels: number of channels in the input image (e.g., RGB)
- out_channels: Number of output channels
- kernel size: Tuple (or int) indicating the dimensions of the convolving kernel

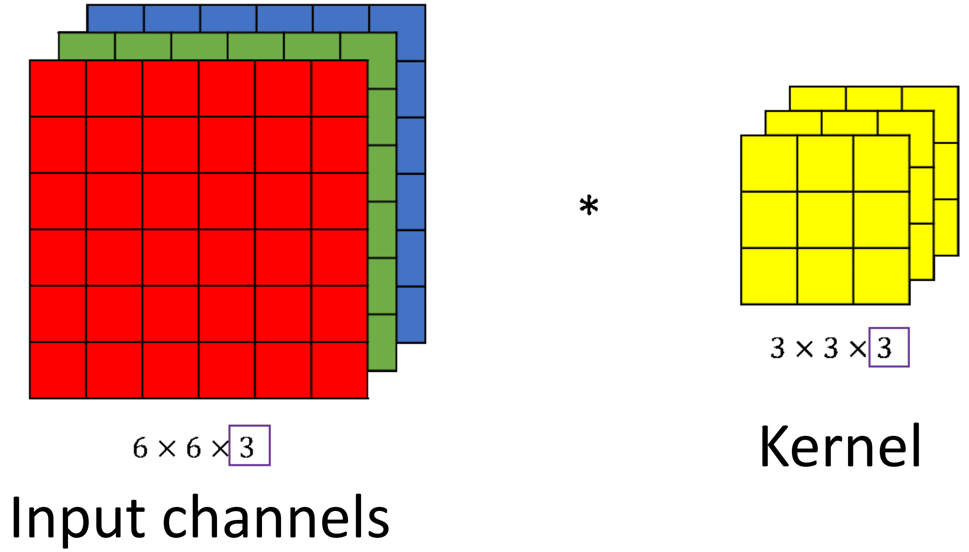
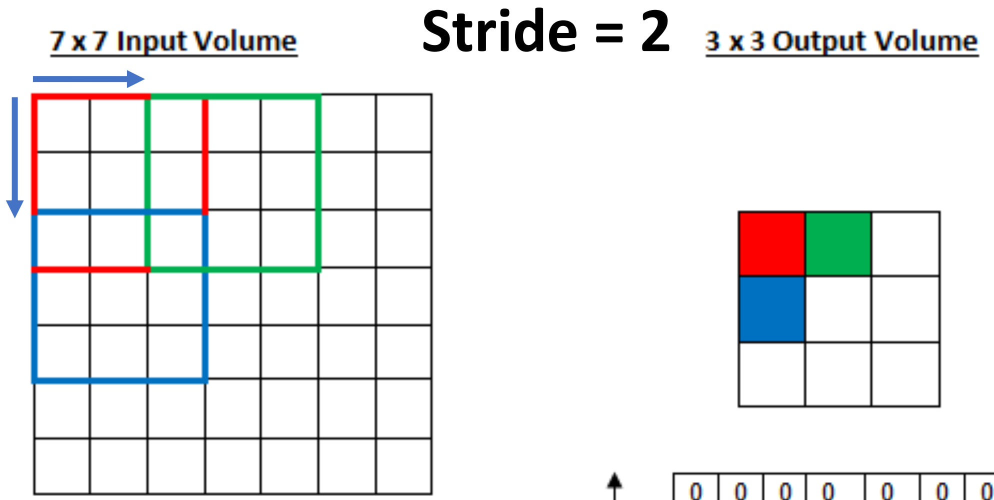


Image source: http://datahacker.rs/convolution-rgb-image/

Convolutional Layers: Arguments

- stride: step size between each convolution – how far the kernel moves in each direction between convolutions (default: 1)

- padding- zero-padding added to each side of the input (default: 0)



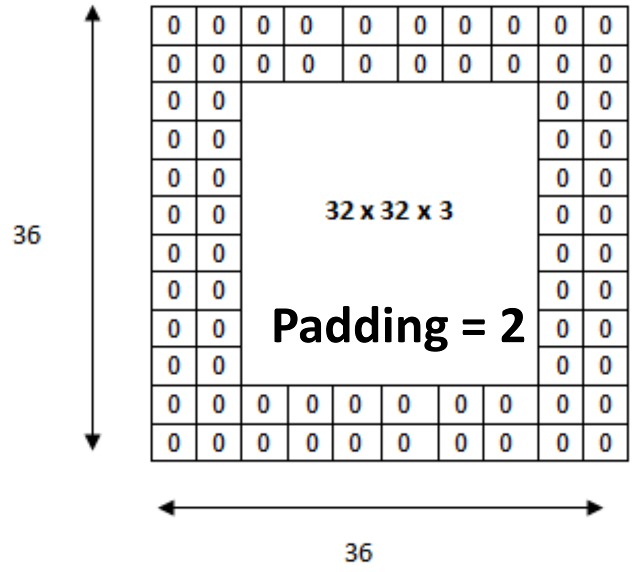


Image source: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

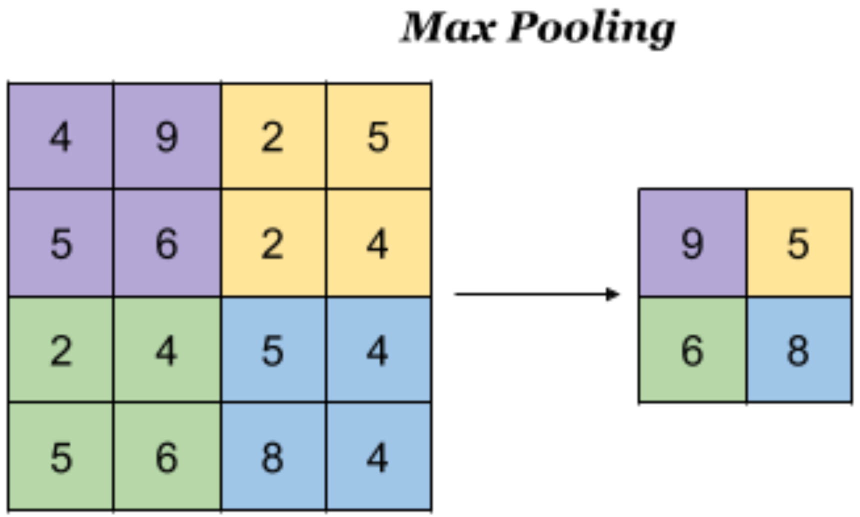
Pooling

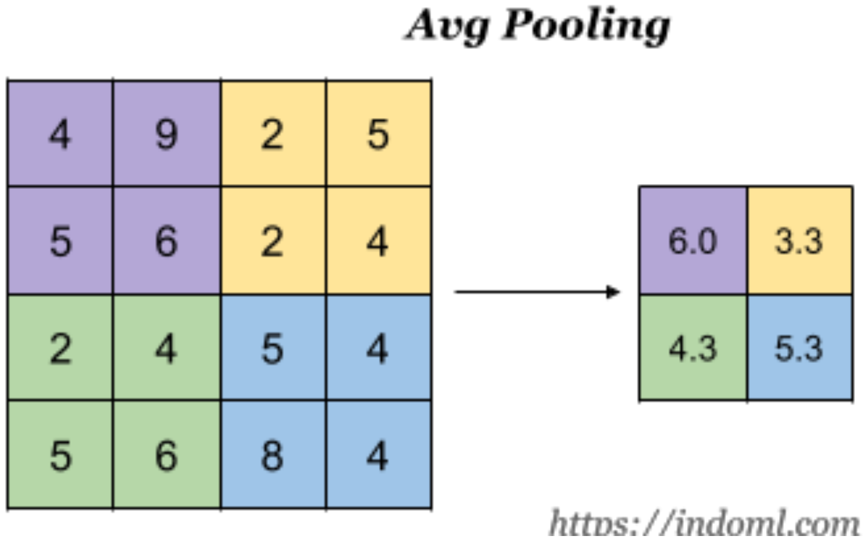
- Operates like Convolutional Layers, but perform simple math operations
- Moving frame calculates one of:
  - Max
  - Mean
  - Power-average (power defined by argument norm_type)

Example:

```
max_pool = nn.MaxPool2D(kernel_size = 2,
stride= 2)
```

```
avg_pool = nn.AvgPool2D(kernel_size = 2,
stride= 2)
```





Example: CNN Implementation

Following Tutorial here: https://medium.com/swlh/pytorch-real-step-by-step-implementation-of-cnn-on-mnist-304b7140605a

Data Preparation

- Use train_test_split to create a validation set from your training data (should be in array/tensor form) (20%)

```
from sklearn.model_selection import train_test_split
X_train, X_cv, y_train, y_cv = train_test_split(features, labels,
                                                test_size = 0.2,
                                                random_state = 1212)

X_train = np.array(X_train).reshape(33600, 784) #(33600, 784)
X_cv = np.array(X_cv).reshape(8400, 784) #(8400, 784)
```

Data Preparation

- Reshape data so that it has correct dimensions for CNN.
- MNIST images are grayscale, so they have only one channel
- Images should be 28x28

```
#Formatting on training set
train_x = X_train.reshape(33600, 1, 28, 28)
train_x  = torch.from_numpy(train_x).float()
# converting the target into torch format
y_train = torch.from_numpy(np.array(y_train))
# shape of training data
train_x.shape, y_train.shape

#Formatting on testing set
X_cv = X_cv.reshape(8400, 1, 28, 28)
X_cv  = torch.from_numpy(np.array(X_cv)).float()
# converting the target into torch format
y_cv = torch.from_numpy(np.array(y_cv))
X_cv.shape, y_cv.shape
```

Data Preparation

- Set batch size and create DataLoader objects for easier training

```
# batch_size, epoch and iteration
batch_size = 100


# Pytorch train and test sets
train = torch.utils.data.TensorDataset(train_x,y_train)
test = torch.utils.data.TensorDataset(X_cv,y_cv)

# data loader
train_loader = torch.utils.data.DataLoader(train, batch_size = batch_size,
                                           shuffle = False)
test_loader = torch.utils.data.DataLoader(test, batch_size = batch_size,
                                          shuffle = False)
```

Model Definition: Initialization

- We will use a standard architecture consisting of:
  - Two 2d convolutional layers w/ filter size (3x3). Each layer has 16 and 32 output channels, respectively
  - Two 2d MaxPool layers with filter size (2x2)
  - ReLU activations
  - An FC layer of 800 nodes

```python
# Create CNN Model
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()

        # Convolution 1
        self.cnn1 = nn.Conv2d(in_channels=1, out_channels=16,
                              kernel_size=3, stride=1, padding=0)
        self.relu1 = nn.ReLU()

        # Max pool 1
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        # Convolution 2
        self.cnn2 = nn.Conv2d(in_channels=16, out_channels=32,
                              kernel_size=3, stride=1, padding=0)
        self.relu2 = nn.ReLU()

        # Max pool 2
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)

        # Fully connected 1
        self.fc1 = nn.Linear(32 * 5 * 5, 10)
```

Model Definition: Forward()

- As the input gets processed at each step, the dimension of the images changes
- The convolutional layers increase the number of channels used to represent the data
- Output is logits

```python
    def forward(self, x):
        # Input x dimensions:   #nx1x28x28
        # Set 1
        out = self.cnn1(x)      #nx16x26x26
        out = self.relu1(out)
        out = self.maxpool1(out)#nx16x13x13

        # Set 2
        out = self.cnn2(out)    #nx32x11x11
        out = self.relu2(out)
        out = self.maxpool2(out)#nx32x5x5

        #Flatten
        out = out.view(out.size(0), -1) #nx800

        #Dense
        out = self.fc1(out)     #nx10

        return out
```

Define Hyperparameters, Loss, Optimizers

- Define training iterations, learning rate
- Classification- use CrossEntropyLoss
- For optimizer, we use SGD for this example

```
#Definition of hyperparameters
n_iters = 2500
num_epochs = n_iters / (len(train_x) / batch_size)
num_epochs = int(num_epochs)

# Cross Entropy Loss
error = nn.CrossEntropyLoss()

# SGD Optimizer
model = CNNModel()
learning_rate = 0.001
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Model Training

- Define the quantities you want to track before the training loop
- tensor.view() allows you to reshape your input so that it is in the form your network needs

```
# CNN model training
count = 0
loss_list = []
iteration_list = []
accuracy_list = []
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        train = Variable(images.view(100,1,28,28))
        labels = Variable(labels)
        # Clear gradients
        optimizer.zero_grad()
        # Forward propagation
        outputs = model(train)
        # Calculate softmax and cross entropy loss
        loss = error(outputs, labels)
        # Calculating gradients
        loss.backward()
        # Update parameters
        optimizer.step()

        count += 1
```

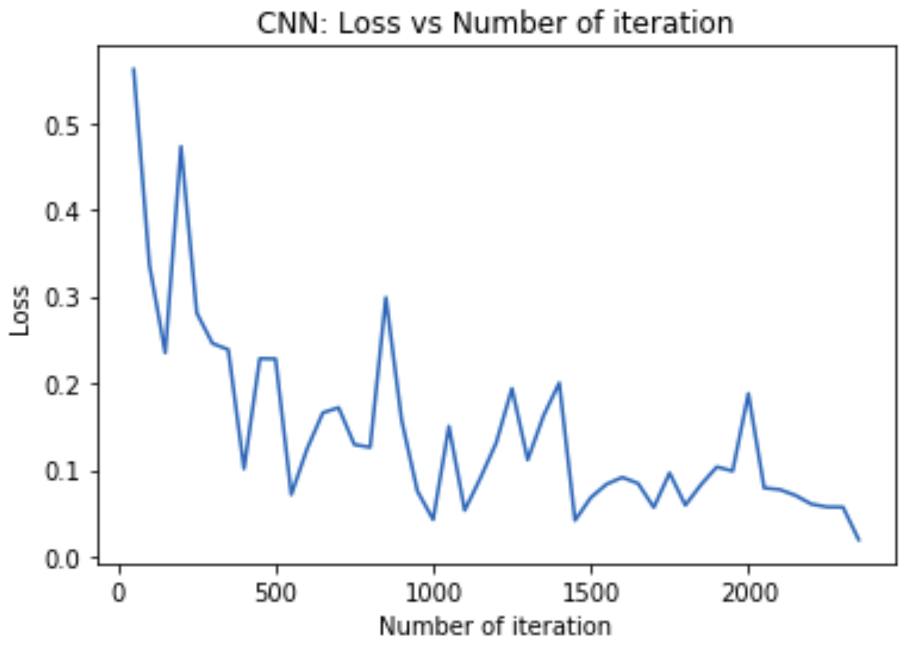
Model Training: Tracking progress

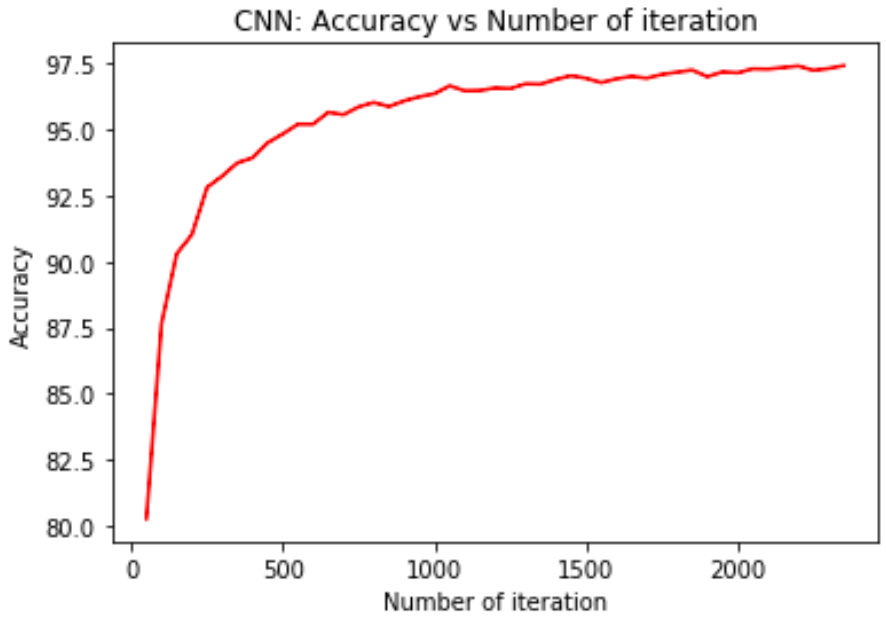
- Within training loop, track your accuracy on the validation/test set.
- Test and print at pre-defined intervals
- Track relevant information in lists defined above (loss, accuracy, iteration)

```
if count % 50 == 0:
    # Calculate Accuracy
    correct = 0
    total = 0
    # Iterate through test dataset
    for images, labels in test_loader:

        test = Variable(images.view(100,1,28,28))
        # Forward propagation
        outputs = model(test)
        # Get predictions from the maximum value
        predicted = torch.max(outputs.data, 1)[1]

        # Total number of labels
        total += len(labels)
        correct += (predicted == labels).sum()

    accuracy = 100 * correct / float(total)

    # store loss and iteration
    loss_list.append(loss.data)
    iteration_list.append(count)
    accuracy_list.append(accuracy)
if count % 500 == 0:
    # Print Loss
    print('Iteration: {}  Loss: {}  Accuracy: {} %'.format(count, loss.data, accuracy))
```

Plot your tracked quantities

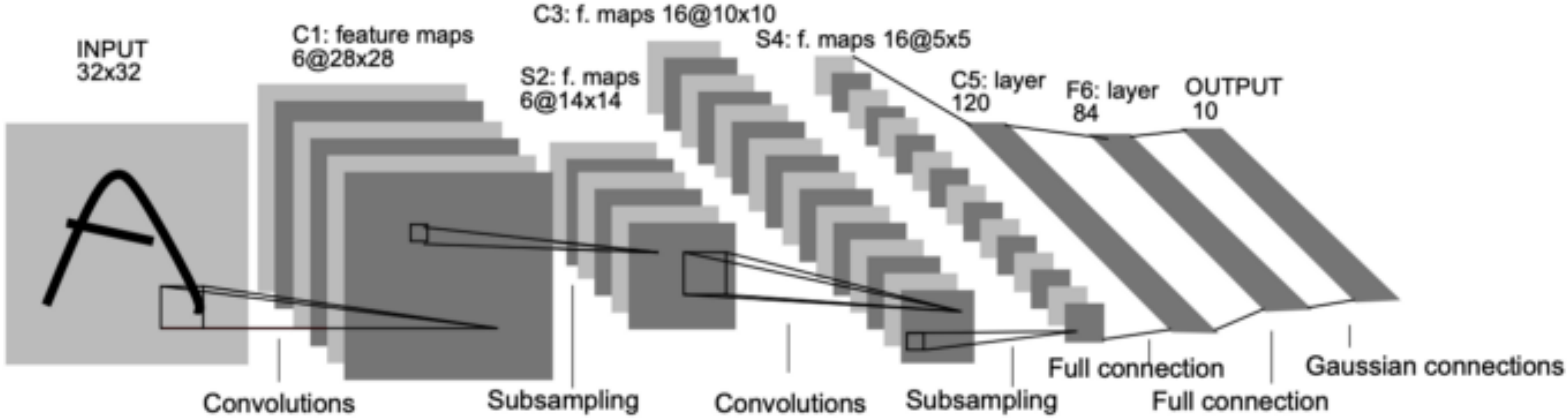
```
# visualization loss
plt.plot(iteration_list,loss_list)
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("CNN: Loss vs Number of iteration")
plt.show()

# visualization accuracy
plt.plot(iteration_list,accuracy_list,color = "red")
plt.xlabel("Number of iteration")
plt.ylabel("Accuracy")
plt.title("CNN: Accuracy vs Number of iteration")
plt.show()
```





Exercise: MNIST Classification using LeNet-5

LeNet-5 Model

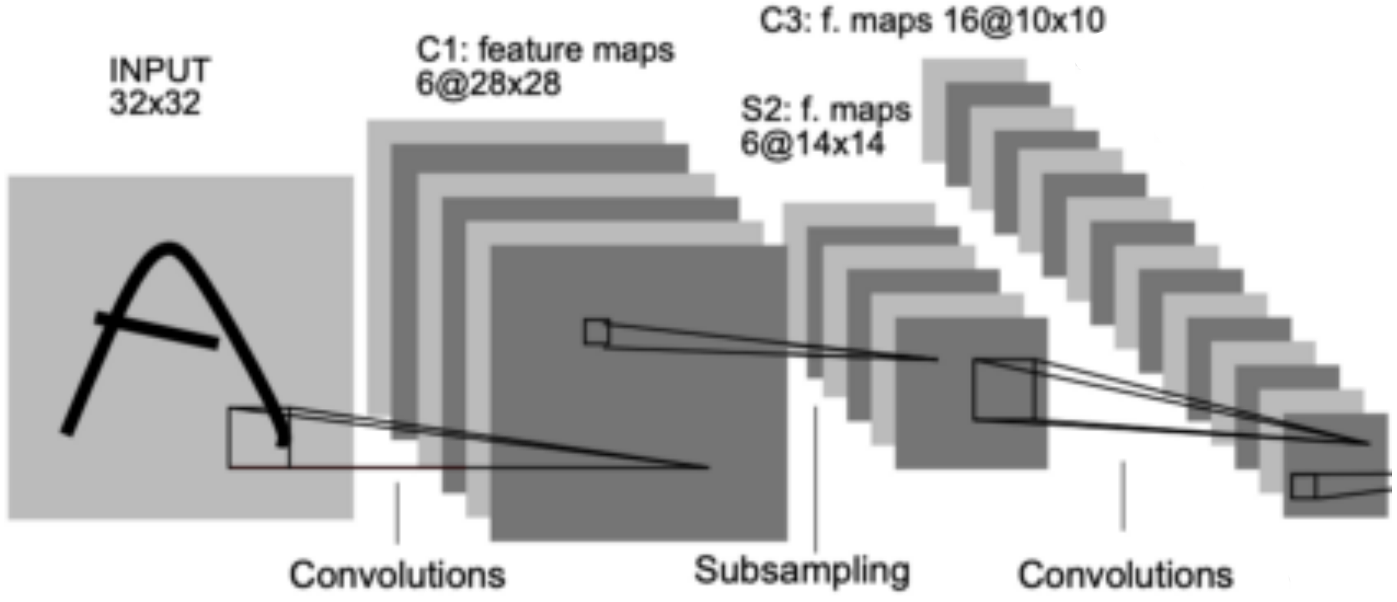


- Well-known network with seven layers, three of which are convolutional.

Image Source:
http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

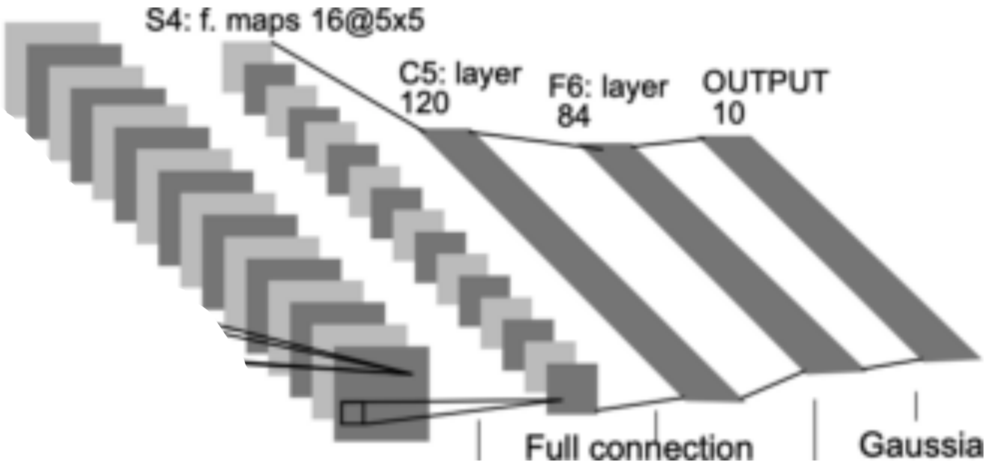
LeNet-5 Model Layers

- Layer 1: Convolutional Layer with 6 filters (output channels), kernel size of 5x5, and padding of 2
- Layer 2: Average pooling (2x2 kernel)
- Layer 3: Convolutional layer. 16 filters, 5x5 kernel size, no padding



LeNet Model Layers

- Layer 4: Average pooling (2x2)
- Layer 5: 120 filters of size 5x5. Output is 1x1x120
- Layer 6: Fully connected layer. Input dimensions: 120, Output dimensions: 84
- Layer 7: Fully-connected layer. Input dimensions: 84, Output dimensions: 120



Assignment Details

- Implement LeNet in PyTorch
- tanh activation
- Use Adam Optimizer
- Should be able to achieve greater than 95% accuracy

```
class LeNet5(nn.Module):

    def __init__(self, n_classes):
        super(LeNet5, self).__init__()

        #define LeNet5

    def forward(self, x):
        #Define forward pass
        return logits    #can also return probabilities
                         #by performing softmax
```