Section 2 : Deep Learning with PyTorch

Notebook:	Secure and Private AI Scholarship Challenge		
Created:	01-Jun-19 11:09 PM	**Updated:**	05-Jun-19 8:34 PM
Author:	Taimur Zahid		
URL:	about:blank#blocked		

UDACITY
facebook Artificial Intelligence
SECURE AND PRIVATE AI
SCHOLARSHIP CHALLENGE FROM FACEBOOK

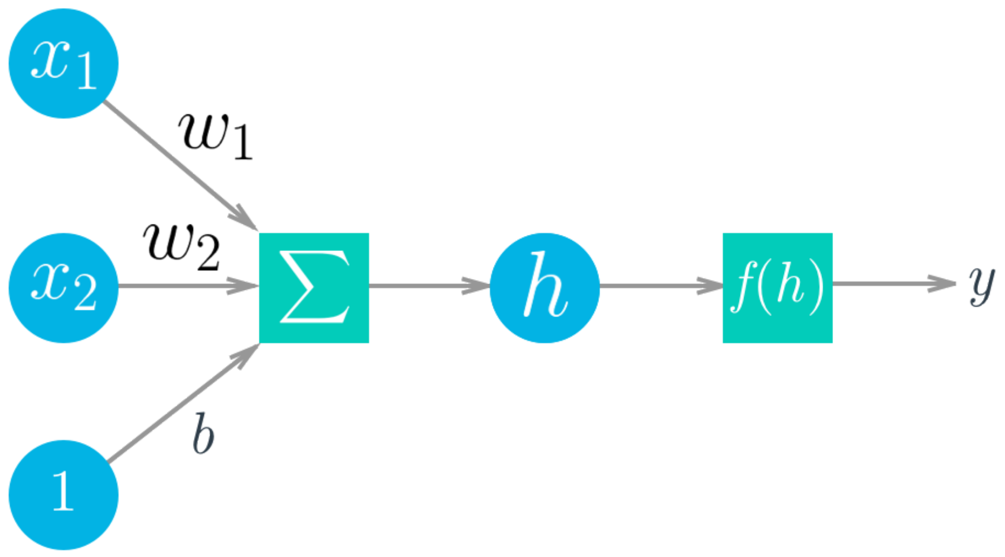
Resources: https://github.com/taimurzahid/Secure-and-Private-AI-Scholarship-Challenge
LinkedIn: https://linkedin.com/in/taimurzahid/

About PyTorch and this Lesson:

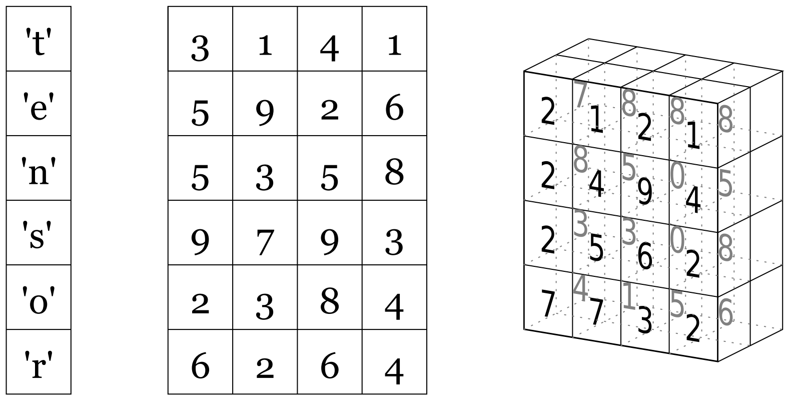
- We will cover Tensors - the main Data Structure of PyTorch.
- We will Learn about Autograd, which is a module used by PyTorch to calculate Gradients for training Neural Networks. It does all the work of **backpropagation**.
- We will build a network using PyTorch and run data **forward** through it.
- We will implement **Transfer Learning**.
- Notebooks: https://github.com/udacity/deep-learning-v2-pytorch/tree/master/intro-to-pytorch

Single Layer Neural Network:

- Neural Networks are built from individual units, called neurons. Each Unit has some weighted inputs which are summed together and then passed through an activation function to get the unit's output.



- Tensors are the base Data Structure that we use in PyTorch or other Neural Network Frameworks, like Tensorflow, etc. A tensor is a generalization of Vectors and Matrices. A tensor of a one dimensional array is a Vector. A two-dimensional Array tensor is called a Matrix.



't'
'e'
'n'
's'
'o'
'r'

tensor of dimensions [6]
(vector of dimension 6)

3	1	4	1
5	9	2	6
5	3	5	8
9	7	9	3
2	3	8	4
6	2	6	4

tensor of dimensions [6,4]
(matrix 6 by 4)

tensor of dimensions [4,4,2]

- Any higher dimensional tensors are simply called Tensors. An example of a three-dimensional tensor is the RGB Image, where each color channel is a single dimension. For every pixel, there is a value in each of the color channel.
- Creating a Sigmoid Activation Function:

```
import torch
```

```
def activation(x):
    return 1/(1 + torch.exp(-x))
```

$$y = f(w_1 x_1 + w_2 x_2 + b)$$
$$y = f\left(\sum_i w_i x_i + b\right)$$

-

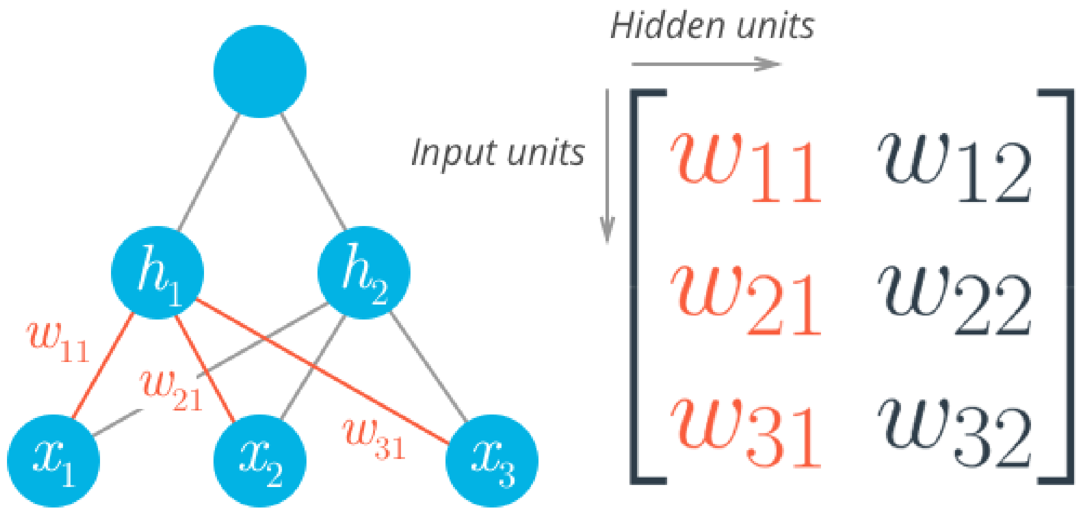
```
y = activation(torch.sum(features * weights) + bias)
# OR
y = activation((features * weights).sum() + bias)
```

- A more efficient method is to use **Matrix Multiplication** as it can be accelerated using modern libraries, like CUDA, and high-performance computing GPUs. **torch.mm()** is more simple and more strict about the tensors that you pass in. Whereas, **torch.matmul()** supports broadcasting. This means that the output can be non-deterministic if the input tensors' shapes mismatch. It is recommended to use torch.mm() since it behaves as expected and throws an error instead of producing weird results.
- In order to use torch.mm() the number of columns in the first matrix needs to be equal to the number of rows in the second matrix. This can be done in the following ways.
- **weights.reshape(a,b)** which will return a new tensor with a copy of the tensor with size (**a , b**).
- **weights.resize_(a,b)** returns the same tensor with a different shape. Keep in mind that if the new tensor's shape results in fewer elements than the original tensor, some elements will be removed from the tensor. Similarly, if the new shape results in more elements than the original tensor, the new elements will be uninitialized in memory. The underscore implies in-place operation. These operations change the tensor directly without making a copy.

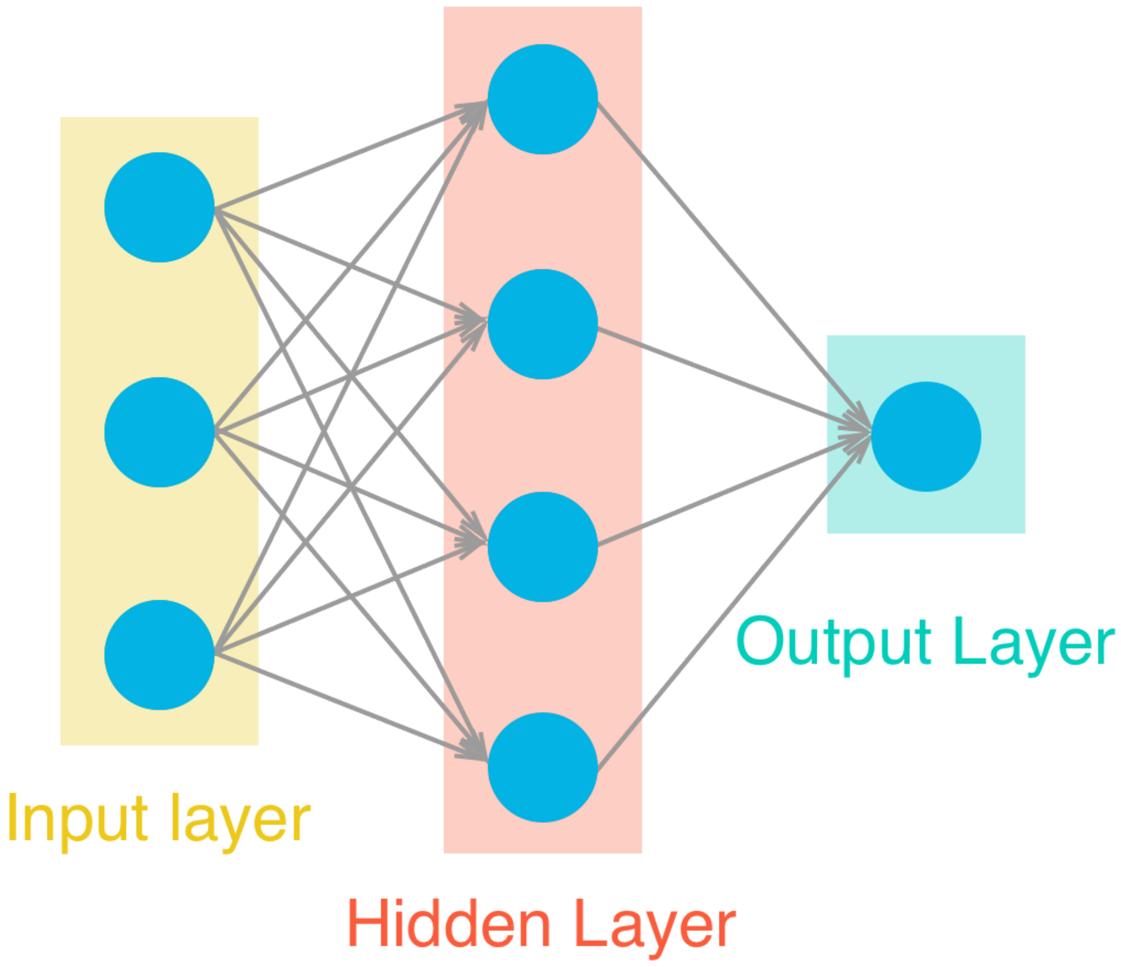
- **weights.view(a,b)** will return a new tensor with the same data as weights and with size **(a,b)**. It doesn't mess with the data in memory. It will also return an error if the total number of elements are affected after reshaping.
- **tensor.shape** will output the shape of the tensor. Extremely useful for Debugging.
- **Networks using Matrix Multiplication:**

```
y = activation( torch.mm(features, weights.view(5,1)) + bias )
```

Multilayer Neural Networks:


Hidden units
Input units
h_1
h_2
w_{11}
w_{21}
w_{31}
x_1
x_2
x_3
$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$

- The output of one layer of neurons becomes the input for the next layer.
- For multiple input and output neurons, we now need to express the weights as matrix.



- The first layer is the input layer. The middle layer is the hidden layer and the last layer is the output layer.
- The output for this network can be given by:

$$y = f_2\left(f_1\left(\vec{x}\,\mathbf{W}_1\right)\mathbf{W}_2\right)$$

```
h = activation(torch.mm(features, W1) + B1)
output = activation(torch.mm(h, W2) + B2)
```

Numpy to Torch and Vice Versa:

- This makes it easy for you to convert a Tensor into a Numpy array and perform Preprocessing steps and then convert it back to a Tensor.

```
b = torch.from_numpy(a) # creates a tensor from numpy array

a = b.numpy # converts it back to a tensor
```

- Note that the memory is shared between the two variables, a and b. Changing one in-place will affect the other.

Neural Networks in PyTorch:

- MNIST is a bunch of grayscale handwritten digits. The label of the image is the number the image contains.

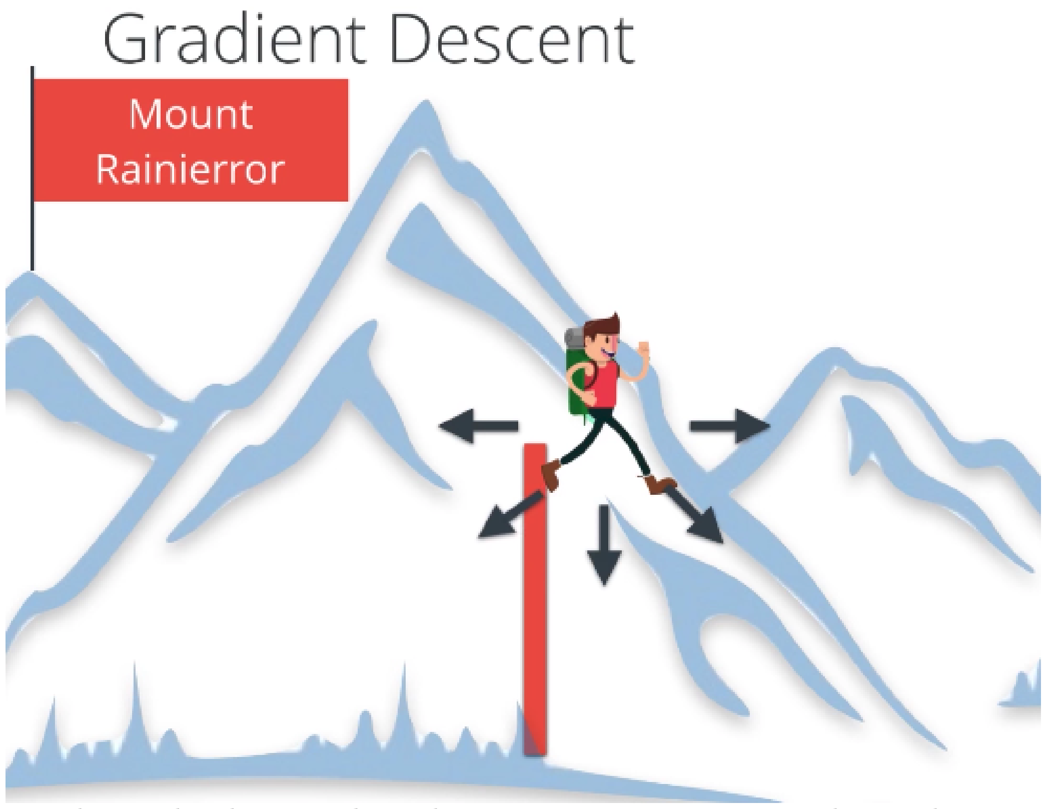
- A batch size of 64 implies that every time we'll iterate it over the loop, we'll get a set of 64 images and labels out.

```
# MNIST dataset
source: https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/02-
intermediate/convolutional_neural_network/main.py#L35-L56

batch_size = 64

train_dataset = torchvision.datasets.MNIST(
    root='../../data/',
    train=True,
    transform=transforms.ToTensor(),
    download=True)

test_dataset = torchvision.datasets.MNIST(
    root='../../data/',
    train=False,
    transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    shuffle=False)
```

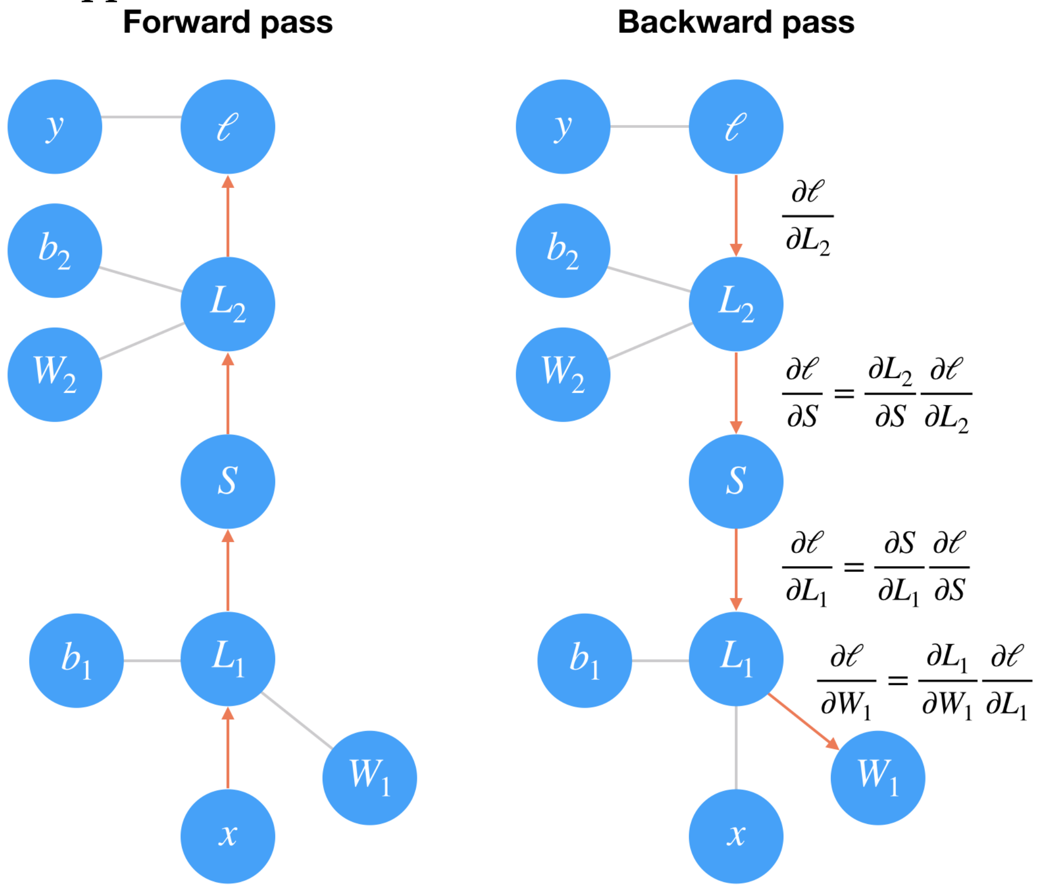
- A neural network in PyTorch with 784 input units, 2 hidden layers with 128 and 64 units, and 10 output units.

```
class Network(nn.Module):
    def __init__(self):
        super().__init__()
        # Defining the layers, 128, 64, 10 units each
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        # Output layer, 10 units - one for each digit
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        ''' Forward pass through the network, returns the output logits '''

        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.softmax(x, dim=1)

        return x

model = Network()
model
```

- Training a Neural Network requires us to give the input and output label to the network to train a model that would approximate a function to learn how the input relates to the output label and tries to generalize on it.
- We would evaluate the performance of this function using a loss function which is a measure of the prediction error. We want to measure how far away our network prediction is from the correct label.
- The loss depends on the output of the network. And the output of the network depends on the weights, i.e network parameters.
- Our goal is to adjust the weights such that the loss is minimized.
- We use Gradient Descent to adjust the weights. The gradient is the slope of the loss function with respect to our parameters. Since the gradient points towards the fastest change, we can take the negative of the gradient to go downhill where the loss is minimum.



- With Multi-layered architecture, we use an algorithm called backpropagation to do this. Backpropagation is an application of the chain rule from calculus.

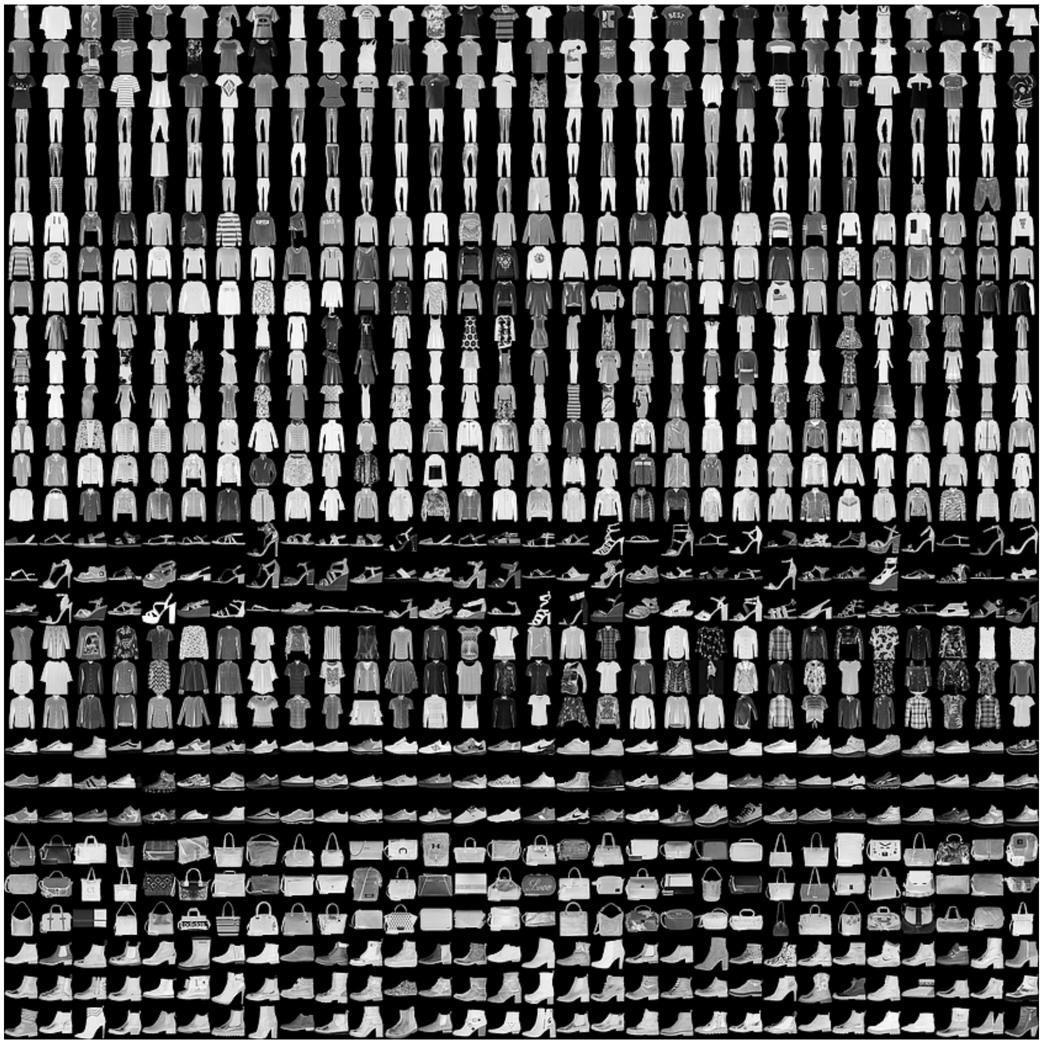

Forward pass
Backward pass
y
ℓ
b_2
L_2
W_2
S
b_1
L_1
W_1
x
$\frac{\partial \ell}{\partial L_2}$
$\frac{\partial \ell}{\partial S} = \frac{\partial L_2}{\partial S}\frac{\partial \ell}{\partial L_2}$
$\frac{\partial \ell}{\partial L_1} = \frac{\partial S}{\partial L_1}\frac{\partial \ell}{\partial S}$
$\frac{\partial \ell}{\partial W_1} = \frac{\partial L_1}{\partial W_1}\frac{\partial \ell}{\partial L_1}$

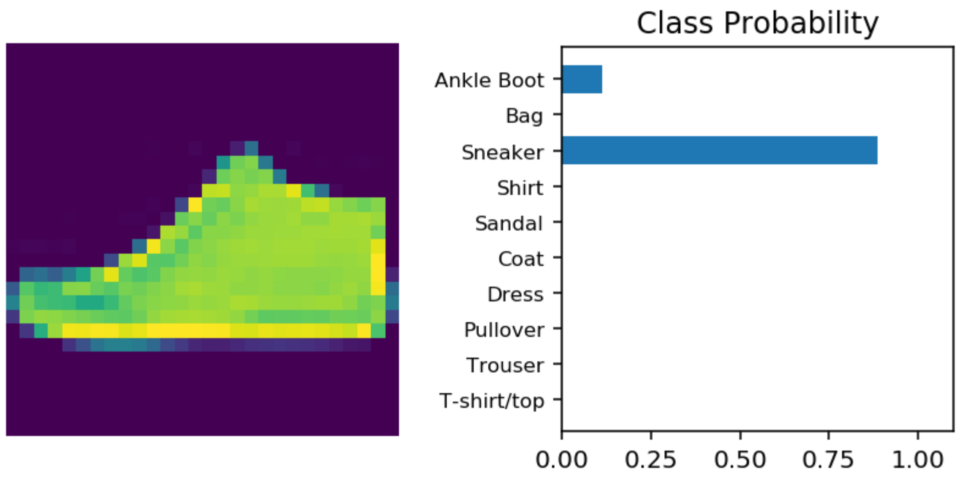
- In the forward pass through the network, our data and operations go from bottom to top here. We pass the input x through a linear transformation $L1$ with weights $W1$ and biases $b1$. The output then goes through the sigmoid operation S and another linear transformation $L2$. Finally we calculate the loss ℓ . We use the loss as a measure of how bad the network's predictions are. The goal then is to adjust the weights and biases to minimize the loss.

- To train the weights with gradient descent, we propagate the gradient of the loss backwards through the network. Each operation has some gradient between the inputs and outputs. As we send the gradients backwards, we multiply the incoming gradient with the gradient for the operation.
- In short, we'll make a forward pass through the network and calculate the loss. Then we go back to the network and calculate the gradient for the weights and update the weights. We repeat this until we get a minimized loss. We use alpha to prevent too large updates in the weights.
- Cross-Entropy Loss is typically used in classification problems.
- Optimizer is used to update the weights.
- A training pass starts by first making a forward pass through the network, then use the network output to calculate the loss, perform a backward pass to calculate gradients, and finally take a step with the optimizer to update the weights. This is performed in a loop to train a network.
- You need to prevent the gradients from summing up by clearing them at the start of every loop. Otherwise, you will be getting gradients from the previous training step in the current training step and will affect the training process.

Classifying Fashion-MNIST:

- You have a 28 x 28 grayscale images of clothes into one of 10 classes.

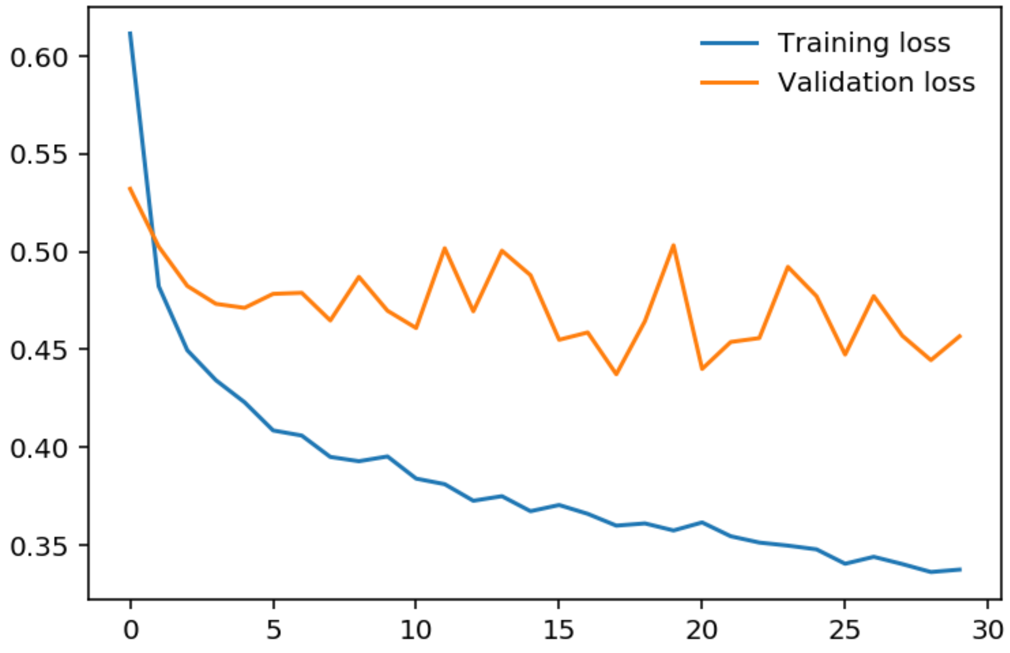




- x.shape[0] will give the batch size.
- Adam Optimizer is similar to Stochastic Gradient Descent, but it uses momentum which speeds up the fitting process. It also adjusts the learning rate for each of the individual parameters in your model.
- Inference means that we make predictions using a trained model.
- Overfitting refers to the model's inability to generalize to the training data and thus performing too well on the

training data while performing poorly on new data. In overfitting, the networks picks up correlations and patterns that are in the training set but aren't in the general dataset.

- In order to test for overfitting, we measure the performance on data not in the training set, called validation set. For overfitting the training loss is far less than the validation loss.



- To avoid overfitting, we'll use regularization techniques such as Dropout. Dropout forces the network to share information between the weights, and so this increase this ability to generalize to new data. We don't use dropout.

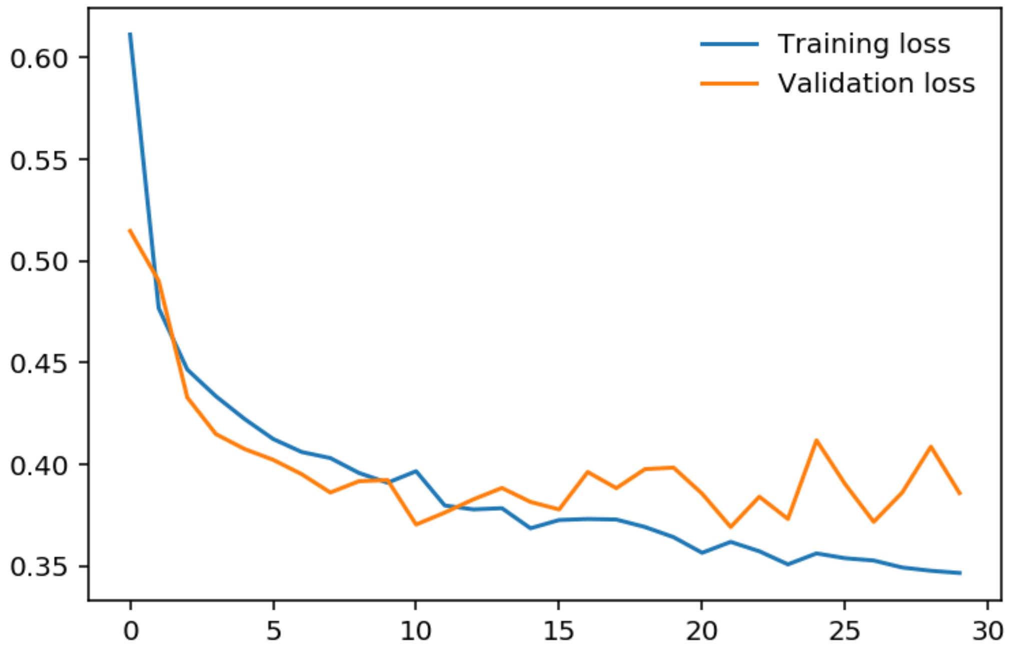
```
# dropout module with 20% drop probability
self.dropout = nn.Dropout(p=0.2)

# in the forward function
x = self.dropout( F.relu( self.fc1(x) ) )
```

- We turn off dropout when we're doing Validation, Testing, and Inference. To do that we turn the model into evaluation mode.

```
with torch.no_grad():
    # set model to evaluation mode
    model.eval()

    # validation pass here
    for images, labels in testloader:
        ...
```

```
# set model back to train mode
model.train()
```

- an ideal case, the training loss and the validation loss both should be less.



- Saving a model allows us to train a network once and use it later for making predictions. The parameters for PyTorch networks are stored in a model's state_dict. The state_dict contains the weight and bias matrices for each of our layers.

```
# saving the state dict to a checkpoint file.
torch.save(model.state_dict(), 'checkpoint.pth')

# loading the saved checkpoint file back to a state dict
state_dict = torch.load('checkpoint.pth')
```

- Model loading as shown above is pretty straightforward. If we load the state dictionary into a model with a different architecture, we're gonna get an error for a size mismatch.
- In order to avoid this size mismatch error, we have to rebuild the model exactly as it was when it was trained. So, we include the information about the architecture of the model within the checkpoint itself.

```
# saving the checkpoint file
checkpoint = {  'input_size' = 784,
                'output_size' = 10,
                'hidden_layers' = [each.out_features for each in
model.hidden_layers],
                'state_dict' : model.state_dict()}

torch.save(checkpoint, 'checkpoint.pth'}

# leading the checkpoint file
# this function needs to be built custom for every model
def load_checkpoint(filepath):
    checkpoint = torch.load(filepath)
    model = fc_model.Network(checkpoint['input_size'],
                             checkpoint['output_size'],
                             checkpoint['hidden_layers'])
    model.load_state_dict(checkpoint['state_dict'])

    return model

model = load_checkpoint('checkpoint.pth')
```

- To load in image data with ImageFolder, we need to define some transforms. Transforms include random crop, resize, center crop, to tensor, etc. You need to combine them into a pipeline with Compose() operation to run then in a sequence. You can use Transforms to Augment your data, i.e introduce Randomness in your data.

```
transforms = transforms.Compose([ transforms.Resize(255),
                                  transforms.CenterCrop(224),
                                  transforms.RandomResizedCrop(100),
                                  transforms.RandomHorizontalFlip(),
                                  transforms.RandomRotation(30),
                                  transforms.ToTensor(),
                                  transforms.Normalize([0.5, 0.5, 0.5],
                                                       [0.5, 0.5, 0.5])])
```

- Once the ImageFolder is loaded, you'll have to pass it to a DataLoader. It takes a dataset and returns batches of images and corresponding labels.

```
dataset = datasets.ImageFolder(data_directory, transform = transforms)

dataloader = torch.utils.data.DataLoader(dataset, batch_size = 32, shuffle =
True)

# looping through it, get a batch on each loop
for images, labels in dataloader:
    pass

# get one batch only
images, labels = next(iter(dataloader))
```

- Convolutional Layers exploit patterns and regularities in images.

- Pre-trained networks such as the ImageNet can be used as a feature detector for the data it hasn't been trained on.
- Transfer learning refers to what was previously learn by the model to be transferred to your dataset.

```
# download the pre-trained model and load it into our model
model = models.densenet121(pretrained = True)
```

- We want to keep the feature part of the pre-trained model static and we want to train the classifier part. In order to keep the feature part of the pre-trained model static, we need to freeze the feature parameters. This means that when we run the tensors through the model, its not going to calculate the gradients. It will also speed up training.

```
# freeze our feature parameters
for param in model.parameters():
    param.requires_grad = False
```

- Next task is to create our own classifier. This classifier needs to be trained on our data. We then set up our model.

```
from collections import OrderedDict
classifier = nn.Sequential(OrderedDict([
             ('fc1', nn.Linear(1024, 500)),
             ('relu', nn.ReLU()),
             ('fc2', nn.Linear(500, 2)),
             ('output', nn.Logsoftmax(dim=1))
]))

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

model.fc = classifier
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.fc.parameters(), lr = 0.003)
model.to(device)
```

- GPU to CPU and Vice Versa

```
model.cuda() # move the model to GPU
images.cuda() # move tensors to GPU
model.cpu() # move model to CPU
images.cpu() # move tensors to CPU
```

- Training the model using Transfer Learning.

```
epochs = 1
steps = 0
running_loss = 0
print_every = 5

for epoch in range(epochs):
    for images, labels in trainloader:
        steps += 1

        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()

        logps = model(images)
        loss = criterion(logps, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        # Validation loop
        if step %% print_every == 0:
            model.eval() #evaluation mode
            test_loss = 0
            accuracy = 0

            for images, labels in testloader:

                images, labels = images.to(device), labels.to(device)

                logps = model(images)
                loss = criterion(logps, labels)
                test_loss += loss.item()

                # calculating our accuracy
                ps = torch.exp(logps)
                top_ps, top_class = ps.topk(1, dim = 1)
                equality = top_class == labels.view(*top_class.shape)
                accuracy += torch.mean(accuracy.type(torch.FloatTensor))

               print(f"Epoch {epoch+1}/{epochs}.. "
                     f"Train loss: {running_loss/print_every:.3f}.. "
                     f"Test loss: {test_loss/len(testloader):.3f}.. "
                     f"Test accuracy: {accuracy/len(testloader):.3f}")
               running_loss = 0
               model.train()
```

Activation Functions in Deep Learning:

- **Sigmoid Activation Function** squeezes the input values between zero and one. It is really useful for providing a probability.

```
def sigmoid(x):
    return 1/(1 + torch.exp(-x))
```

- **ReLU Activation Function**

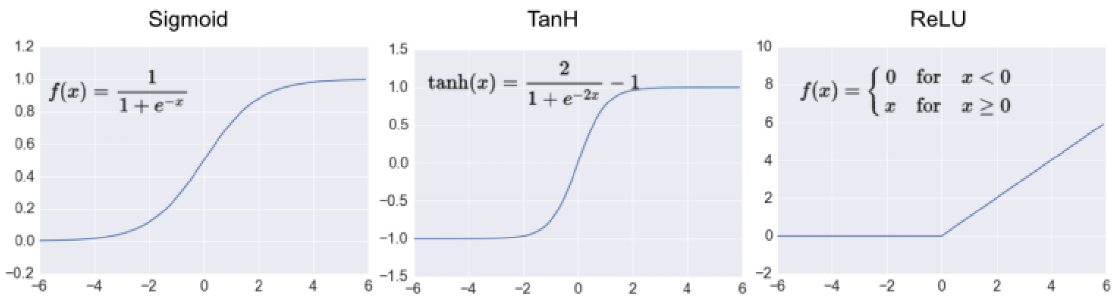
```
def relu(x):
    return max(0, x)
```

- **Softmax Activation Function** calculates a probability distribution that assigns the highest probability to the most likely class for a certain input.

```
def softmax(x):
    return torch.exp(x)/torch.sum(torch.exp(x), dim=1).view(-1, 1)
```

- **Hyperbolic Tangent Activation Function**

```
def tanh(x):
    return (2/(1 + torch.exp(-2x))) -1
```


Sigmoid
$f(x) = \frac{1}{1+e^{-x}}$
TanH
$\tanh(x) = \frac{2}{1+e^{-2x}} - 1$
ReLU
$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$