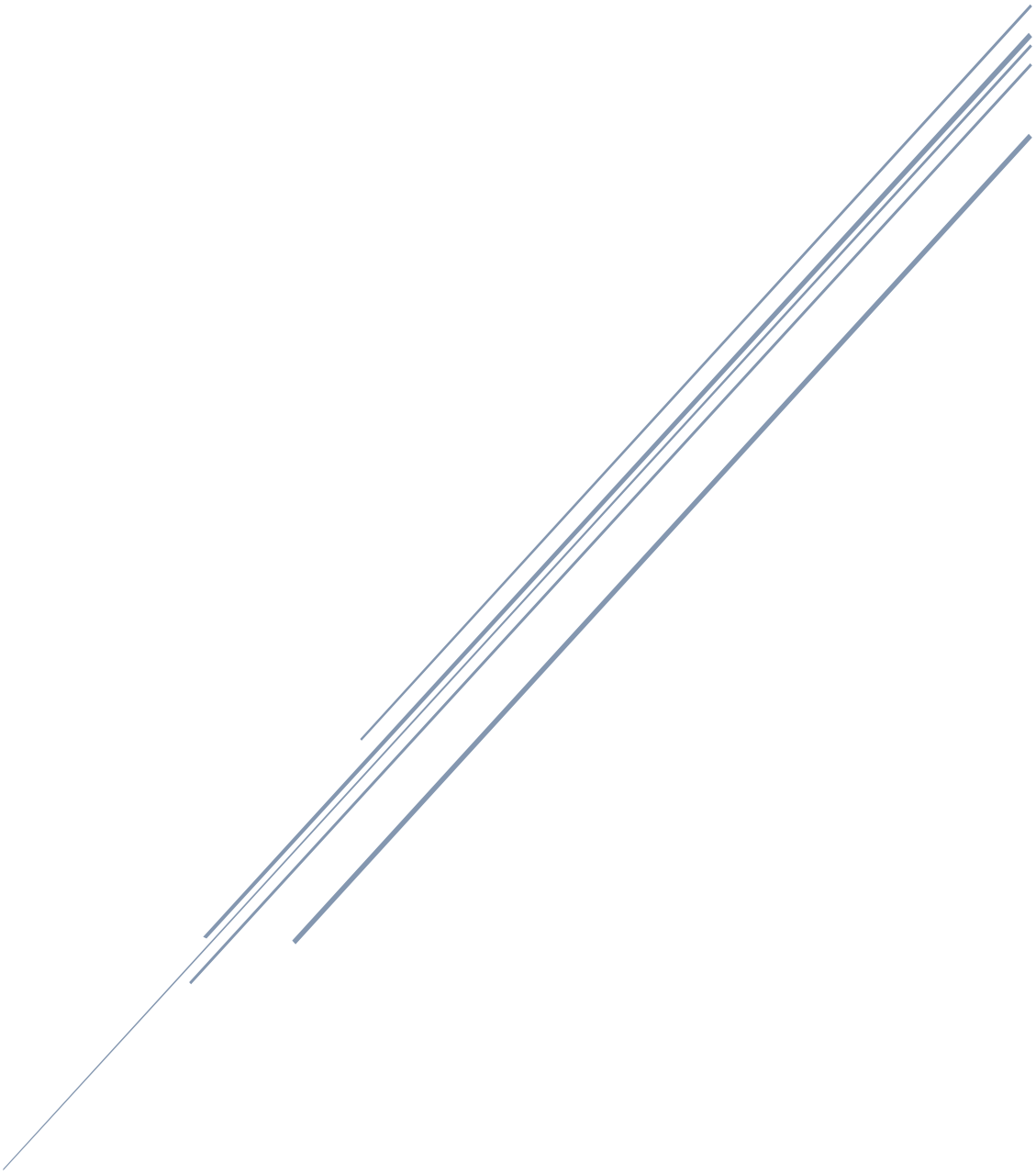
DEEP NETWORKS

Module 5



v-cardona

Deep Learning with Python and PyTorch

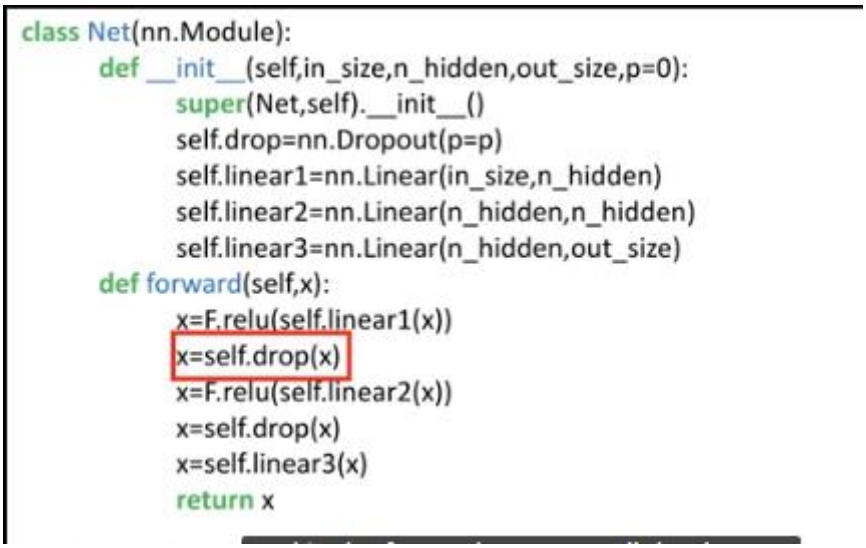
Contenido

Dropout

Regularization technique for make a model without taking into account the noise samples. In each iteration, we randomly kill a neuron of each hidden layer. If it is a high probability we can kill more neurons, if it is low probability will kill one or none. We calculate the forward step as usual, and we produce a prediction.

The model without dropout performed better on the training data, but performed worse on the test data. And the model with dropout performed better on the validation data.

In PyTorch, we put a parameter of probability, and create a dropout object. Then, in the forward step, we call dropout method on the output after every linear layer and activation.



```
class Net(nn.Module):
    def __init__(self,in_size,n_hidden,out_size,p=0):
        super(Net,self).__init__()
        self.drop=nn.Dropout(p=p)
        self.linear1=nn.Linear(in_size,n_hidden)
        self.linear2=nn.Linear(n_hidden,n_hidden)
        self.linear3=nn.Linear(n_hidden,out_size)
    def forward(self,x):
        x=F.relu(self.linear1(x))
        x=self.drop(x)
        x=F.relu(self.linear2(x))
        x=self.drop(x)
        x=self.linear3(x)
        return x
```

In PyTorch you have to set your model to train when you use dropout and you have to set your model to evaluate for the prediction step, or forward step.

Initialization

The total loss of the cost function for neural network has a problem of local minimums.

Randomly

A value between -1 and 1. It has a problem when the variance is too large but also when it is too small. the problem is if we have an arbitrary large number of inputs to a neuron, that we will suffer for vanishing gradient problem.

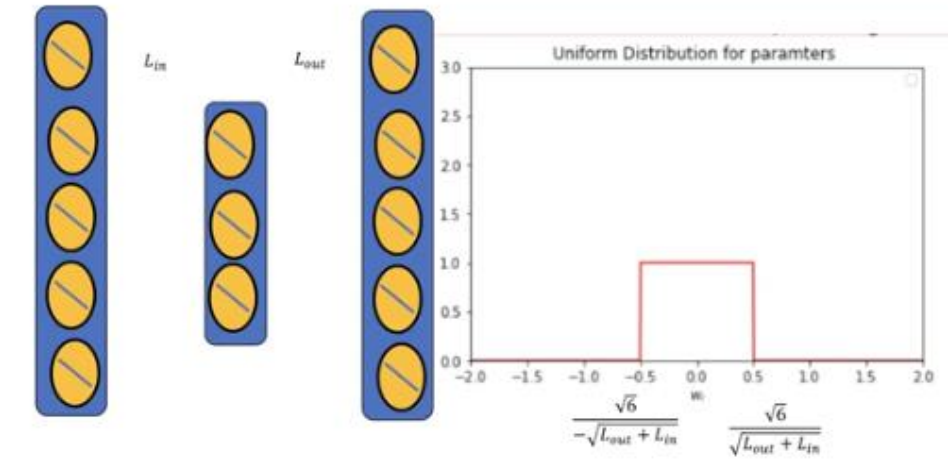
To fix this, one way is to scale the range of the uniform distribution proportional to the number of inputs to the neuron.

Xavier

The neuron, they vary the variance of the neuron, based on the number of inputs and number of outputs.

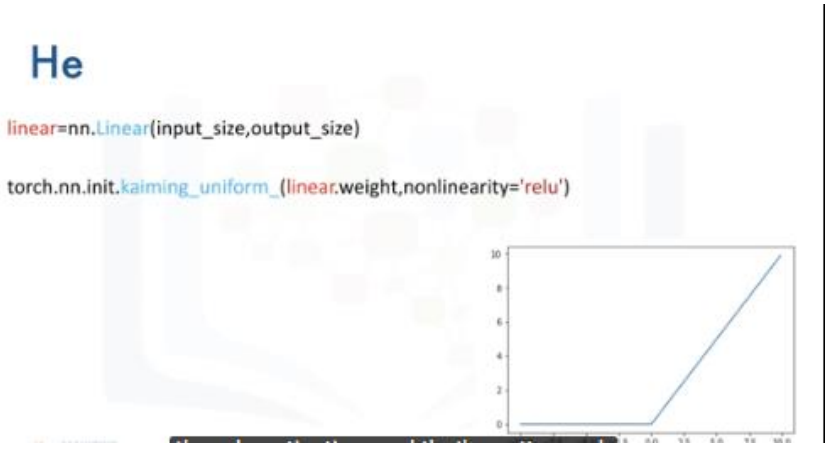
- *torch.nn.init.xavier_uniform_(linear.weight)*

```
class Net_Xavier(nn.Module):

    # Constructor
    def __init__(self, Layers):
        super(Net_Xavier, self).__init__()
        self.hidden = nn.ModuleList()

        for input_size, output_size in zip(Layers, Layers[1:]):
            linear = nn.Linear(input_size, output_size)
            torch.nn.init.xavier_uniform_(linear.weight)
            self.hidden.append(linear)

    # Prediction
    def forward(self, x):
        L = len(self.hidden)
        for (l, linear_transform) in zip(range(L), self.hidden):
            if l < L - 1:
                x = torch.tanh(linear_transform(x))
            else:
                x = linear_transform(x)
        return x
```



He

You initialize your linear object, and then from the module nn.init use the following function and your input is the data attributes weight from your linear object. And for this case, you have to set the type of activation function; we're going to do the relu activation, and that's pretty much it.



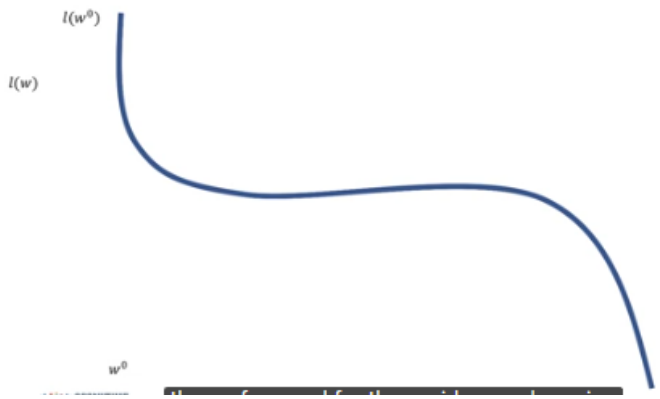
```python
class Net_He(nn.Module):

    # Constructor
    def __init__(self, Layers):
        super(Net_He, self).__init__()
        self.hidden = nn.ModuleList()

        for input_size, output_size in zip(Layers, Layers[1:]):
            linear = nn.Linear(input_size, output_size)
            torch.nn.init.kaiming_uniform_(linear.weight, nonlinearity='relu')
            self.hidden.append(linear)

    # Prediction
    def forward(self, x):
        L = len(self.hidden)
        for (l, linear_transform) in zip(range(L), self.hidden):
            if l < L - 1:
                x = F.relu(linear_transform(x))
            else:
                x = linear_transform(x)
        return x
```

With Relu we should use He initialization.

Gradient descent with momentum

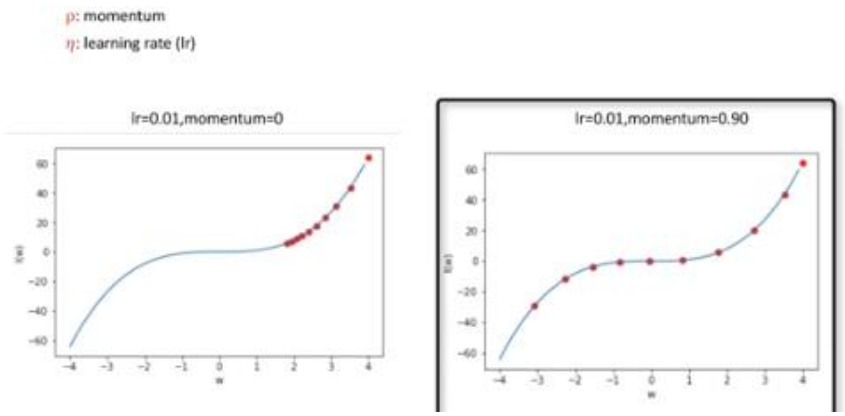
Saddle points

Flat point in a surface



We have into account the velocity that we have, and to upgrade the parameter we add the velocity also. We can pass the saddle point with momentum.

A momentum close to 1, we basically add the two parameters. If it is lower than 1 we obtain a smaller number. If momentum is very small, we can be stuck in the same point

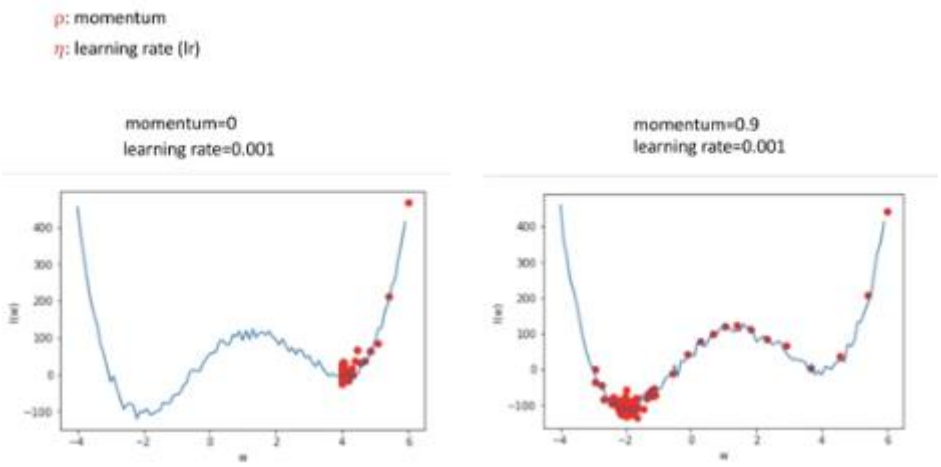


Local minimums

Is the smallest point in a small neighbourhood and the global minimum is the smallest point for the whole function.

If momentum is too small will stuck in a local minimum, when a good momentum value will reach the global minimum. But if it is too large, we can miss any local minimum

Gradient noise

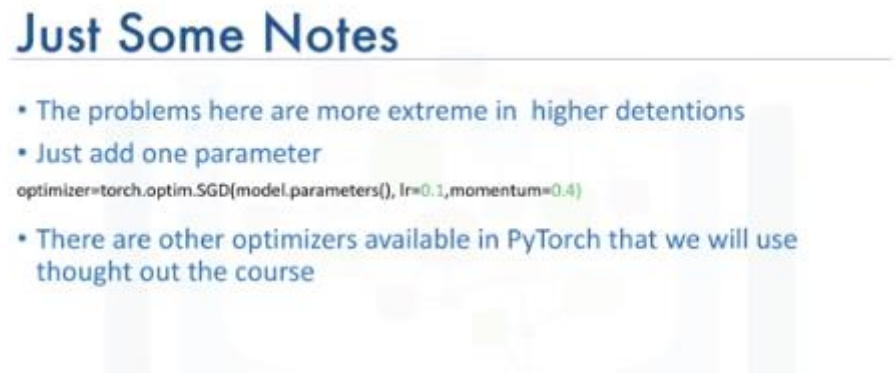


+-Noise value

High condition number

This has to do with the Eigen values of the Hessian matrix of our cost function. If the cost function has a condition number of one it is basically round, and the gradient will point to every K the minimum and we're from gradient descent will reach a minimum very fast. If our cost function has, a high condition number will take more time to converge.

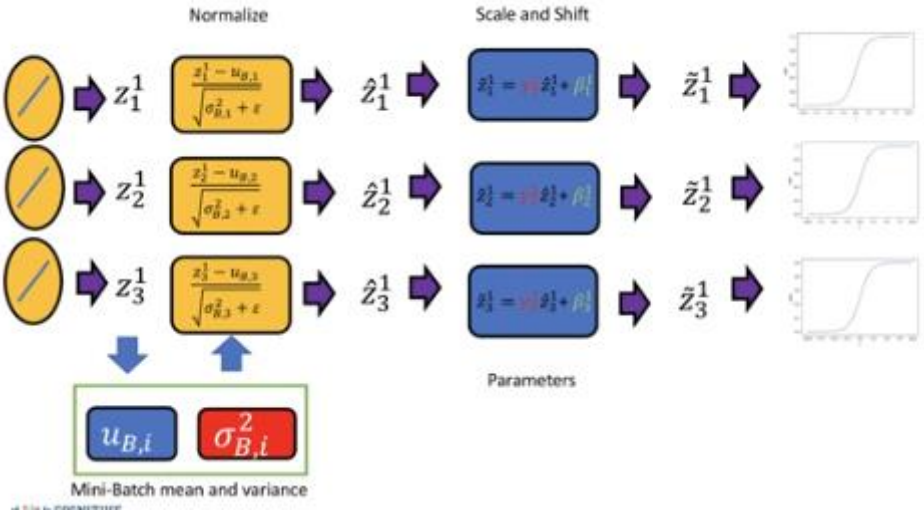
The greater the momentum, the faster it takes our algorithm to converge, but a lower can gives you a better loss value, but not always. We will select the momentum value using validation data.



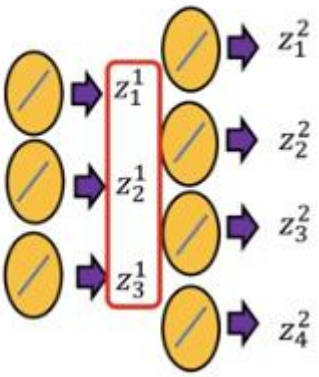
- *optimizer_momentum=optim.SGD(model_momentum.parameters(),lr=0.01,momentum=0.2)*

Batch normalization

We apply batch normalization only before passing the activation functions.




```
class NetBatchNorm(nn.Module):
  def __init__(self,in_size,n_hidden1,n_hidden2,out_size):

    super(NetBatchNorm,self).__init__()

    self.linear1=nn.Linear(in_size,n_hidden1)
    self.linear2=nn.Linear(n_hidden1,n_hidden2)
    self.linear3=nn.Linear(n_hidden2,out_size)

    self.bn1 = nn.BatchNorm1d(n_hidden1)
    self.bn2 = nn.BatchNorm1d(n_hidden2)

def forward(self,x):
    x=F.sigmoid(self.bn1( self.linear1(x)))
    x=F.sigmoid(self.bn2( self.linear2(x)))
    x=self.linear3(x)
    return x
```



Has to set the model to train. If you look at our training loss you can see it converges a lot faster and similar results for the test data as well.

More

- Reducing Internal Covariate Shift
- Remove Dropout
- Increase learning rate.
- Bias is not necessary

```
class NetBatchNorm(nn.Module):

    # Constructor
    def __init__(self, in_size, n_hidden1, n_hidden2, out_size):
        super(NetBatchNorm, self).__init__()
        self.linear1 = nn.Linear(in_size, n_hidden1)
        self.linear2 = nn.Linear(n_hidden1, n_hidden2)
        self.linear3 = nn.Linear(n_hidden2, out_size)
        self.bn1 = nn.BatchNorm1d(n_hidden1)
        self.bn2 = nn.BatchNorm1d(n_hidden2)

    # Prediction
    def forward(self, x):
        x = self.bn1(torch.sigmoid(self.linear1(x)))
        x = self.bn2(torch.sigmoid(self.linear2(x)))
        x = self.linear3(x)
        return x

    # Activations, to analyze results
    def activation(self, x):
        out = []
        z1 = self.bn1(self.linear1(x))
        out.append(z1.detach().numpy().reshape(-1))
        a1 = torch.sigmoid(z1)
        out.append(a1.detach().numpy().reshape(-1).reshape(-1))
        z2 = self.bn2(self.linear2(a1))
        out.append(z2.detach().numpy().reshape(-1))
        a2 = torch.sigmoid(z2)
        out.append(a2.detach().numpy().reshape(-1))
        return out
```