Introduction to Deep Learning with NumPy and PyTorch 3/4

Part 3 - High Level API and Support Classes
"Bird's View"

1st Terascale Alliance Machine Learning School, DESY, Hamburg

Dirk Krücker
DESY - Hamburg, 23.10.2018

Content III

- Layers for networks
  - Linear layers
  - Parameters and weight initialization
  - Sequential model
  - Layers for activation function
- Data loader
  - Losses
    - classification and binary cross entropy
  - Saving and loading
- Base skeleton for learning
- A “physical” toy model
- Optimizer
  - stochastic decent
  - mini batch
  - Adam
  - Momentum

The High-Level API

Now you can forget all you have learnt before (for a while)

- PyTorch comes with predefined modules to build neural networks. You simply plug together predefined elements from
  - torch.nn
  - torch.optim
  - ...

PyTorch nn Module

The layers of network building

- The necessary functions that can work with autograd are defined in `torch.nn.functional`
  - They work on the given input alone
  - There is an `inplace` argument to avoid a copy
  - For building networks the similar named Modules within `torch.nn.Module` are used (see later)
- Base class for all neural network modules is `torch.nn.Module`
  - Subclasses of `torch.nn.Module` are **losses** and **network components**.
- Losses typically have an ***input*** as first and a ***target*** as second argument (true value), e.g `torch.nn.MSELoss()`
  - Target must not have requires_grad=True
  - default for `MSEloss` is BTW to normalize by **batch size**

```
import torch.nn.functional as F
x = torch.randn(2,5)
print x
x = F.relu(x,inplace=True)
print x
```

```
tensor([[ 1.8924,  0.0350,  0.3512, -0.9755,  0.7330],
        [-0.7416, -1.7051, -0.5408, -1.2087,  0.8866]])
tensor([[1.8924, 0.0350, 0.3512, 0.0000, 0.7330],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.8866]])
```

```
L = torch.nn.MSELoss()
x = torch.tensor([[ 3. ]])
y = torch.tensor([[ 0. ]])
print L(x, y)
x = torch.tensor([[ 3., 0., 0., 0. ]])
y = torch.tensor([[ 0., 0., 0., 0. ]])
print L(x, y)
```

```
tensor(9.)
tensor(2.2500)
```

PyTorch nn Module

The layers of network building

- **Batch size**
  Modules from `torch.nn` process for efficiency only batches of inputs stored in a tensor with an additional first dimension (batch size N) to index them, e.g. for a n-dim input layer the network expects and (N x n) tensor
- All **network components,** i.e sub-classes of `torch.nn.Module` own parameters (weights and biases) to be optimized during training. E.g. `torch.nn.Module.Linear(n_in,n_out,..)`
- **Parameters** are of the type `torch.nn.Parameter` which is a Tensor with requires_grad =True
  and which is automatically registered as parameter when it is used as within a module,
  e.g. listed by `torch.nn.Module.named_parameters()`
  or `state_dict()`

```
n_in  = 5 # input layer dim
n_out = 2 # output layer dim
N_batch=7 # batch size
Lin = torch.nn.Linear(n_in,n_out,bias=True)
X= torch.randn(N_batch,n_in)
X
```

```
tensor([[ 2.0887, -0.2774, -0.1268, -0.2070, -0.0333],
        [-0.1090,  0.1299,  1.1037,  0.5043,  1.3526],
        [-1.9945,  0.2344, -0.6238, -0.4122, -1.7143],
        [-2.1356,  1.5589, -0.7676,  1.6411,  0.5602],
        [ 0.2318, -0.4806, -1.2825,  1.4896,  0.5033],
        [ 0.6014, -0.5852, -0.5622, -0.7757,  1.0918],
        [-0.2252, -0.4936,  2.4095,  1.3029,  0.2937]])
```

```
Lin(X)
```

```
tensor([[ 0.5273,  0.5355],
        [ 0.0015, -0.5588],
        [ 0.8705,  0.8711],
        [ 1.0274,  1.1841],
        [-0.3085, -0.7045],
        [-0.0713, -0.4259],
        [-0.3867,  0.0860],
        [ 0.5716,  1.2475]], grad_fn=<ThAddmmBackward>)
```

```
for n, p in Lin.named_parameters(): print(n, p.size())
```

```
('weight', torch.Size([2, 5]))
('bias', torch.Size([2]))
```

PyTorch nn Module

The layers of network building

- **Weights** for network components are automatically randomly initialized on creation
- The `torch.nn.linear` module is by default initialized with uniform $\pm\frac{1}{\sqrt{n_{in}}}$

```
Lin.state_dict()
```

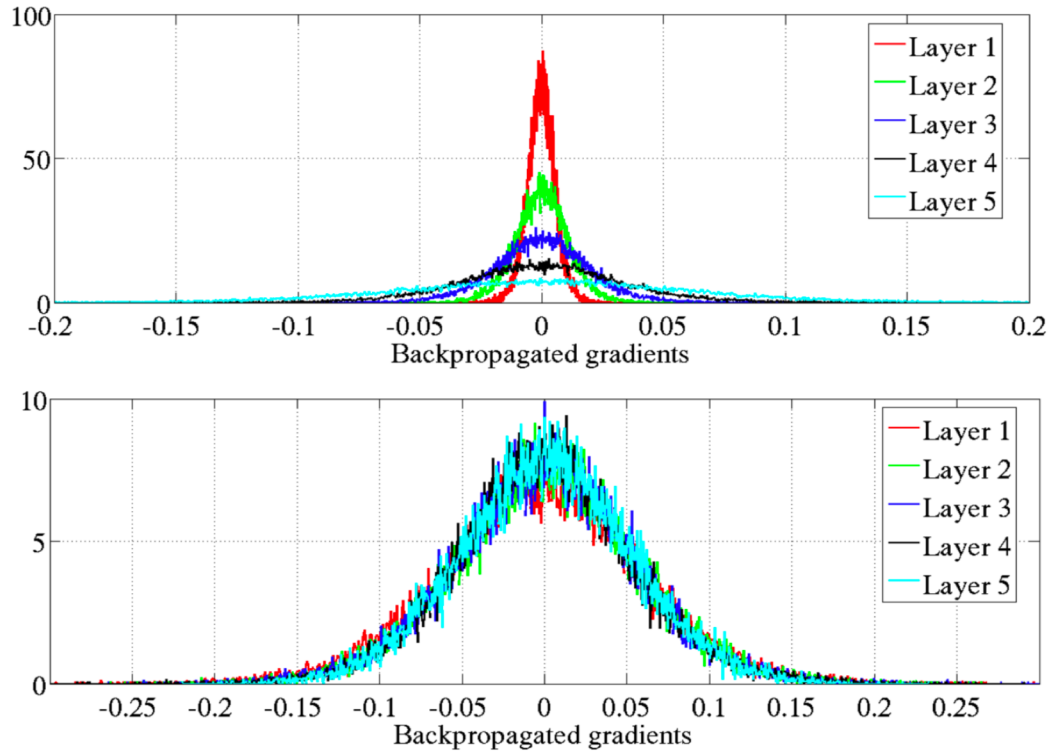
```
OrderedDict([('weight', tensor([[-0.1619, -0.2796,  0.2579, -0.2794,  0.0105],
                                [-0.3019, -0.0996, -0.1139, -0.0191,  0.1561]])),
             ('bias', tensor([-0.1794,  0.2967]))])
```

Xavier Initialization

A short digression

Why do we care about the size of the weights?

- Forward view
  - Large weights will increase the signal after passing each layer. If we use a activation function as *tanh,* it will go into saturation.
  - Small weights will result in a signal that mainly stays in the linear part of the *tanh* around 0. Then our network starts to loose its non-linearity.
- Backward view
  - Deep neural networks become difficult to train because of the *vanishing gradient problem.* If we follow the gradient through the different layers, it tends to become smaller. The effect strongly depends on the initial size, i.e. variance of the weights
- This had been studied by Xavier Glorot and Yoshua Bengio (2010) and later by He at al. (2015)
  The problem can be mitigated by choosing the proper variance when randomly initializing the weights.



(Glorot and Bengio, 2010)

- The optimal variance is different for forward and backward step but they propose to initialize with $\mathrm{Var}(w) = \frac{2}{N_{in}+N_{out}}$ for example by sampling uniformly from $\pm\sqrt{\frac{6}{N_{in}+N_{out}}}$ The details depend on the activation functions. ReLU had been studied by He et al.

PyTorch nn Module

The layers of network building

- **Weights** for network components are automatically randomly initialized on creation

```
Lin.state_dict()
```

```
OrderedDict([('weight', tensor([[-0.1619, -0.2796,  0.2579, -0.2794,  0.0105],
                                [-0.3019, -0.0996, -0.1139, -0.0191,  0.1561]])),
             ('bias', tensor([-0.1794,  0.2967]))])
```

- The `torch.nn.linear` module is by default initialized with uniform $\pm\frac{1}{\sqrt{n_{in}}}$
- Different kind of initialization are provided by `torch.nn.init`, e.g. `torch.nn.init.xavier_uniform_( ... )`

```
torch.nn.init.xavier_uniform_(Lin.weight)
```

```
Parameter containing:
tensor([[ 0.0043,  0.4104, -0.4036, -0.5536,  0.6443],
        [ 0.2108,  0.2076,  0.5097,  0.6935, -0.6726]], requires_grad=True)
```

PyTorch nn Module

The layers of network building

- `torch.nn` provides all kind of layers which are beyond this lecture (see following days)
  - `torch.nn.Conv2d`
  - `torch.nn.MaxPool1d`
  - `torch.nn.Dropout` - see next slides
  - and many more
- `torch.nn` provides all kind of activation functions, for example
  - `torch.nn.ReLU()`
  - `torch.nn.Tanh()`
  - `torch,nn.Sigmoid()`
  - and many more
- They all can be combined into one model with `torch.nn.Sequential`

```
model = torch.nn.Sequential(
    torch.nn.Linear(10,20),
    torch.nn.ReLU(),
    torch.nn.Linear(20,1),
    torch.nn.Sigmoid()
)
```

```
print model
```

```
Sequential(
  (0): Linear(in_features=10, out_features=20, bias=True)
  (1): ReLU()
  (2): Linear(in_features=20, out_features=1, bias=True)
  (3): Sigmoid()
)
```

PyTorch nn Module

The layers of network building

- They all can be combined into one model with `torch.nn.Sequential`
- The layers in a model can be named
- The layer can be accessed by the name or an index
- and the parameters of a layers in a model are available this way

```
model = torch.nn.Sequential()
model.add_module('layer1', torch.nn.Linear(10,20))
model.add_module('relu', torch.nn.ReLU())
model.add_module('layer2', torch.nn.Linear(20,1))
model.add_module('sigmoidOut', torch.nn.Sigmoid())
```

```
print model
```

```
Sequential(
  (layer1): Linear(in_features=10, out_features=20, bias=True)
  (relu): ReLU()
  (layer2): Linear(in_features=20, out_features=1, bias=True)
  (sigmoidOut): Sigmoid()
)
```

```
model.layer2.bias
```

```
Parameter containing:
tensor([-0.0981], requires_grad=True)
```

PyTorch nn Module

The layers of network building

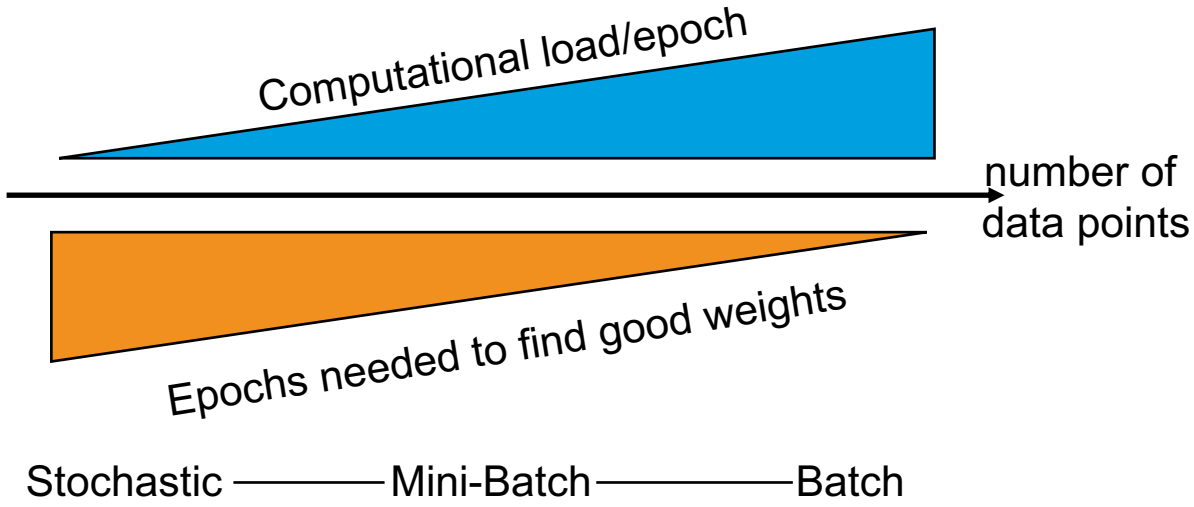
- PyTorch can save models
  - The parameters only
    - `torch.save(the_model.state_dict(), PATH)`
  - The model+parameters
    - `torch.save(the_model, PATH)`
- and load them back

```
torch.save(model.state_dict(), './sequential.para')

# existing model
model.load_state_dict(torch.load('./sequential.para'))

torch.save(model,'./sequential.model')

the_model = torch.load('./sequential.model')
```

PyTorch - Optimizer II

More on Gradient descent optimizer

- Also about using gradient descent for training is more to say than we did up to now
- We have already mentioned that PyTorch prefers batches of data for efficiency
- If we do gradient descent we have the choice between 2 alternatives
  - **Batch gradient descent**
    Take *all* data and calculate then the loss and update the weights
  - **Stochastic gradient descent (SGD)**
    Take just *one* data point and calculate the loss and update the weights
  - When all data points have been used we call it an **epoch**
  - **Batch size** typically means mini-batch



- Medium batch size can be processed efficiently (BLAS, cuBLAs)
- Stochastic gradient descent introduces some noisiness which can be an advantage to avoid local minima
- SGD converges faster - frequent updates
- The exact size is a hyperparameter that can be tuned - typical value 32-256 (2^n to fit into memory)

PyTorch - Optimizer II

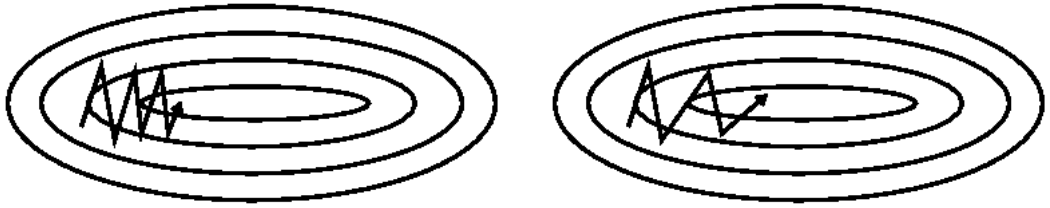
More on Gradient descent optimizer

$$\Delta^i := \eta \frac{dL}{d\mathbf{W}}$$

$$\mathbf{W}^{i+1} = \mathbf{W}^i - \Delta^i$$

η learning rate

- There are more refinements available for the optimizer
  - **Momentum:** Gradient descent is often is very slow near the optimum -> Remember the last step
    - `torch.optim.SGD(params,lr,momentum)`
  - Different kind of adaptive behavior.
  - Popular choice
    - **ADAM** (DP Kingma 2014)
      `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)`
      - Defaults will do in most cases



Gradient descent in a 'canyon', i.e. gradient different in the 2 dimensions without and with momentum

$$\Delta^i := \gamma \Delta^{i-1} + \eta \frac{dL}{d\mathbf{W}}$$

PyTorch - Optimizer II

More on Gradient descent optimizer

- There are more refinements available for the optimizer
  - **Momentum:** Gradient descent is often is very slow near the optimum -> Remember the last step
    - `torch.optim.SGD(params,lr,momentum)`
  - Different kind of adaptive behavior.
  - Popular choice
    - **ADAM** (DP Kingma 2014)
      `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)`
      - Defaults will do in most cases
      - amsgrad see J. Sashank et al. 2018

ADAM

$$m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla L$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)(\nabla L)^2$$
$$\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta_1^t}$$
$$\hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_2^t}$$
$$w_{t+1} = w_t - \eta \frac{\hat{m}_{t+1}}{\sqrt{\hat{m}_{t+1}} + \epsilon}$$

PyTorch - Optimizer II

More on Gradient descent optimizer

- `torch.optim` is the package that provides the different optimizer
- Simplest way to use them by `model.parameters()`
- or you can give explicit list `[var1,var2]`
- You can also specify the parameters of the optimizer, learning rate, momentum etc. differently for each weight or bias:
  Per-parameter options
  - This is a list [] of python dictionaries {}
  - The *key* is a name of a *named parameter,* the values are variables or lists of variables
  - In addition you can give argument values that are used as default

```
var1=torch.tensor(1.,requires_grad=True)
var2=torch.tensor(1.,requires_grad=True)

optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
optimizer = torch.optim.Adam([var1, var2], lr = 0.0001)

optimizer = torch.optim.SGD([
                {'params': model.layer1.parameters()},
                {'params': model.layer2.parameters(), 'lr': 1e-2}
            ], lr=1e-3, momentum=0.9)
```

If you want to use a GPU do the `model.cuda()` first and then add the parameters to the optimizer. Otherwise the optimizer access the wrong memory, the variables version for cpu.

PyTorch - Optimizer II

PyTorch learning - base skeleton

- All optimizer from `torch.optim`
  - All optimizers implement a `step()` method, that updates the parameters
  - The function can be called once the gradients are computed using e.g. `backward()`
- The learning loop:
  - Get the next batch here called input
  - Apply the model to the batch
  - Calculate the loss
  - Set all gradients to zero: `optimizer.zero_grad()`
    - This is a method of the optimizer. The optimizer knows the parameters
    - Explicitly erasing the gradients before calling backward()
  - Do the backward step to calculate the gradients
  - Update the parameters for this iteration: `optimizer.step()`

```
for input, target in dataset:
    output = model(input)
    loss = loss_bce(output, target)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Note the simplicity!

PyTorch - Data Loader

PyTorch learning

```
import torch.utils.data
dataset      = torch.utils.data.TensorDataset(x_train, y_train)
data_loader  = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle = True)

n_=len(data_loader)
n_epochs
for epoch in range(n_epochs):
    for x,y in data_loader:
        x = x.to(device)
        y = y.to(device)

...
```

One element in the learning loop needs further discussion: the Data Loader

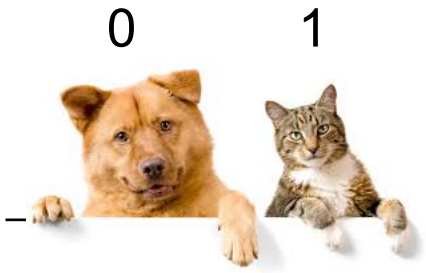
- For (mini) batches we need a framework that serves the trainings data data and targets
- For one epoch all trainings data must be used once (typically randomly shuffled)
- Several utilities in `torch.utils.data`: dataset and data loader

- `torch.utils.data.Dataset`, simpelst case: `torch.utils.data.TensorDataset` for torch tensors
- `torch.utils.data.DataLoader`
- When your data is not a multiple of batch_size the last epoch is smaller

PyTorch nn Module

More on loss functions - Classification

- For classification we need other loss functions than `MSELoss`, especially
  - `torch.nn.CrossEntropyLoss`
  - `torch.nn.BCELoss` binary cross entropy
  - `torch.nn.BCEWithLogitsLoss`
  - **Binary Cross Entropy**



Binary classification

signal or background

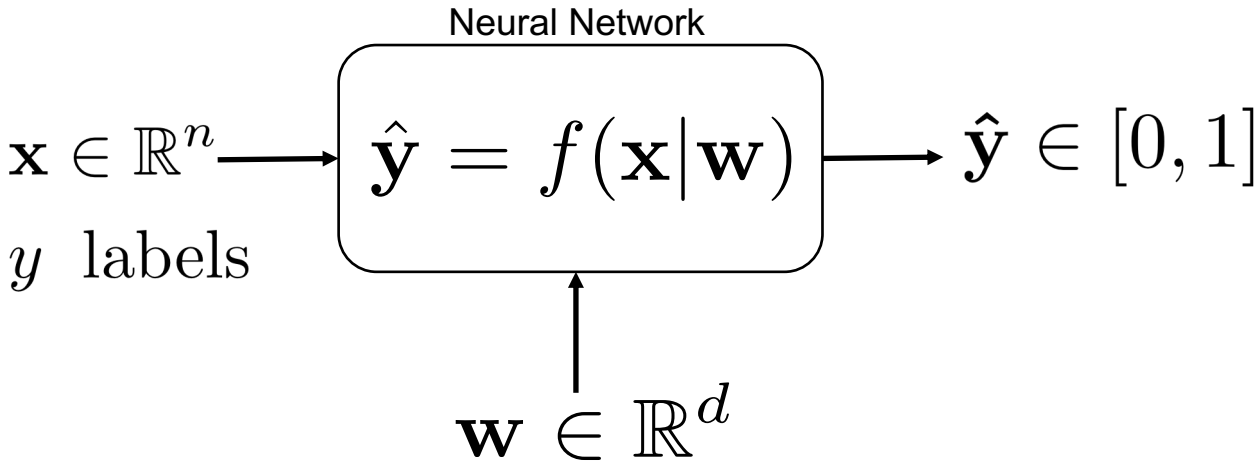
Binary Cross Entropy

Binary what?

- We want to do **binary classification**:
  - We have **labeled** trainings data:
    Background vs signal
  - Some input variables x_i label $y_i \in \{0,1\}$
    Note: y is **discrete** now
  - We want to get a **probability** $\hat{y} \in [0,1]$
    that we have a signal
  - signal: $P(1|\mathbf{x}_i, \mathbf{w}) = \hat{y}_i$

    background: $P(0|\mathbf{x}_i, \mathbf{w}) = 1 - \hat{y}_i$

  - That's a binomial model for which we can get the optimal weights $\mathbf{w}$ by a **Maximum Likelihood fit**
  - The likelihood for one batch of n events with $n=n_s+n_b$ and the $-lnL$



Binary Cross Entropy

Binary what?

- We want to do **binary classification**:
  - We have **labeled** trainings data:
    Background vs signal
  - Some input variables x_i label $y_i \in \{0,1\}$
    Note: y is **discrete** now
  - We want to get a **probability** $\hat{y} \in [0,1]$ that we have a signal
  - signal: $P(1|\mathbf{x}_i, \mathbf{w}) = \hat{y}_i$

    background: $P(0|\mathbf{x}_i, \mathbf{w}) = 1 - \hat{y}_i$

  - That's a binomial model for which we can get the optimal weights $\mathbf{w}$ by a **Maximum Likelihood fit**
  - The likelihood for one batch of n events with $\boldsymbol{n=n_s+n_b}$ and the $-lnL$

$$L(\mathbf{w}|batch) = \prod_{i=1}^{n_s} \hat{y}_i \prod_{j=1}^{n_b} (1-\hat{y}_j)$$

$$-\ln L(\mathbf{w}|batch) = -\sum_{i=1}^{n_s} \ln \hat{y}_i - \sum_{j=1}^{n_b} \ln(1-\hat{y}_j)$$

$$\overset{\text{using the label}}{=} -\sum_{i=1}^{n} \left[y_i \ln \hat{y}_i + (1-y_i) \ln(1-\hat{y}_j) \right]$$

This expression is called **cross entropy** up to 1/n.
Minimizing this expression provides the best optimal weights

Binary Cross Entropy

Why is it called cross entropy?

- Reminder: Entropy is the averaged <$ln(p)$>
  - Shannon Entropy
- Cross entropy is averaged over a different distribution q
- Gibb's inequality

$$\text{Entropy} \equiv \int ln(p(x))\, p(x)\, dx$$

$$\text{Cross Entropy} \equiv \int ln(p(x))\, q(x)\, dx$$

$$\int ln(p)\, p\, dx \leq \int ln(p)\, q\, dx$$

- Cross entropy is a measure how similar two distributions are
- The minimum is $p(x)=q(x)$. Minimizing makes

$$-\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$

similar

PyTorch nn Module

More on loss functions - Classification

- For classification we need other loss functions, especially
  - `torch.nn.BCELoss` **binary** cross entropy
    - typically combined with a sigmoid
  - `torch.nn.BCEWithLogitsLoss`
    - loss and sigmoid in one class for better numerical stability
  - `torch.nn.NLLLoss`
    (negative log likelihood loss)
    - **multi-class** classification - {1,..,C} labels
    - Similar logic, replace the binomial model by a multinomial model
    - Softmax (smooth version of max/multidim version of logistic function)
  - `torch.nn.LogSoftMax`

```
m = nn.LogSoftmax()
loss = nn.NLLLoss()
# input is of size N x C = 3 x 5
input = torch.randn(3, 5, requires_grad=True)
# each element in target has to have 0 <= value < C
target = torch.tensor([1, 0, 4])
output = loss(m(input), target)
output.backward()
```

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

Please work through the following notebook:

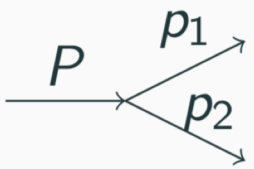
createToy.ipynb

toy_classification.ipynb

The toy model

The toy model

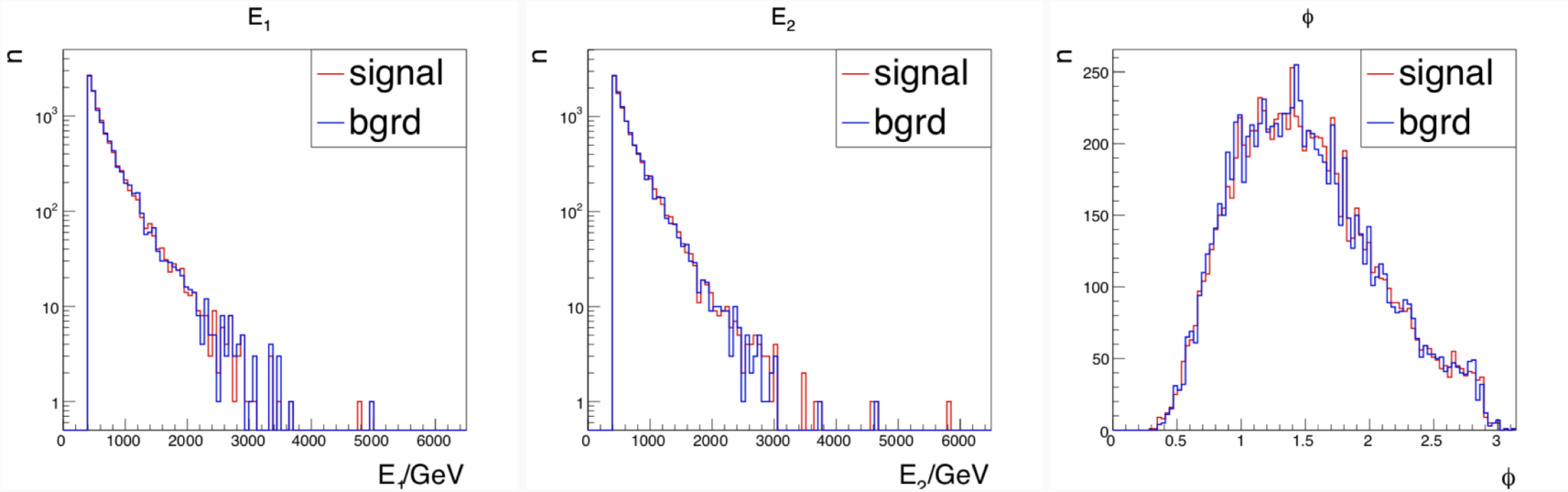
We need a toy example:



- Assume we have 1 particle decaying into 2 particles
  - E.g. a 750 GeV particle decays into 2 massless particles that we observe in our favourite detector
- The heavy particle does have some varying z-momentum in the lab-frame
- There are many other background particles
- We measure (lab system) the energies and the relative angle: E_1, E_2 and ϕ
  - Some cuts to select high energetic particles, e.g.: $E > 400$ GeV cut

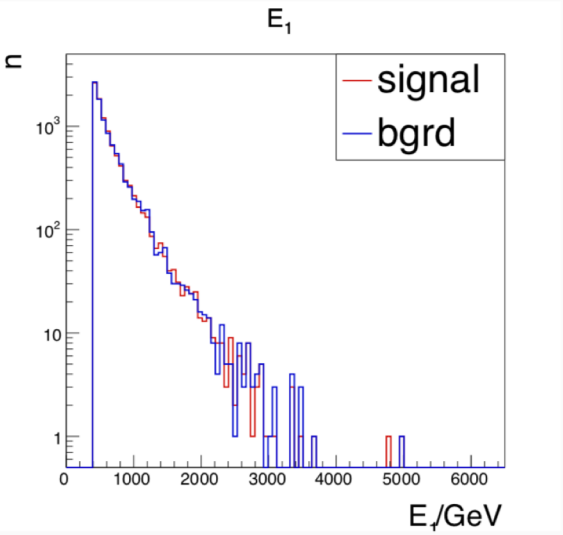
The toy model

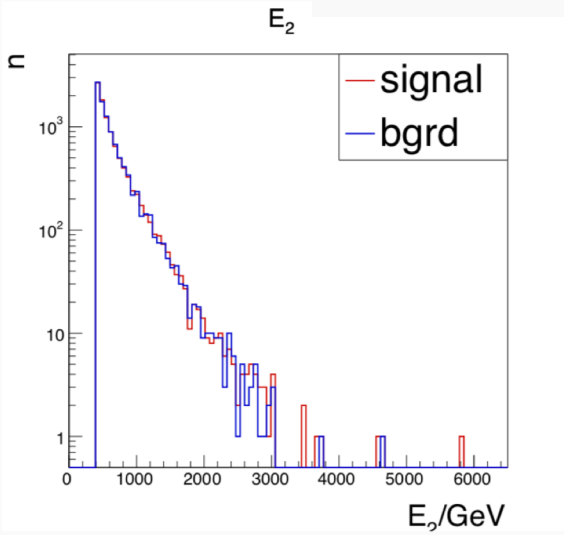
Invariant mass implied correlations

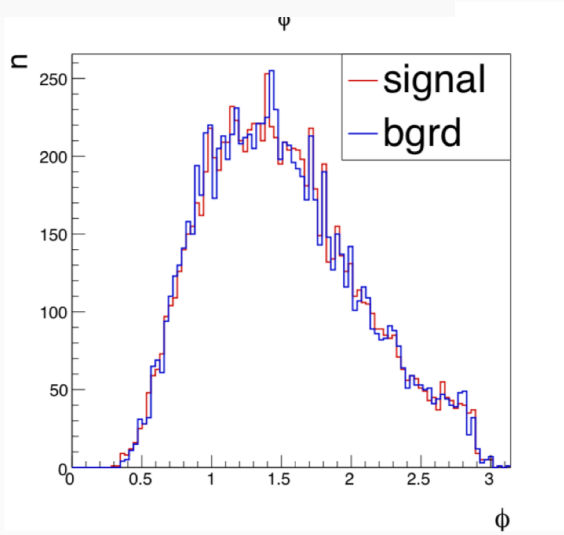


- The signal and background distribution for E_1, E_2 and ϕ are identical in this example
- The "background" had been created by resampling from the E_1,E_2 and ϕ distribution but with mixing particles from different events $\Longrightarrow$ correlation destroyed!

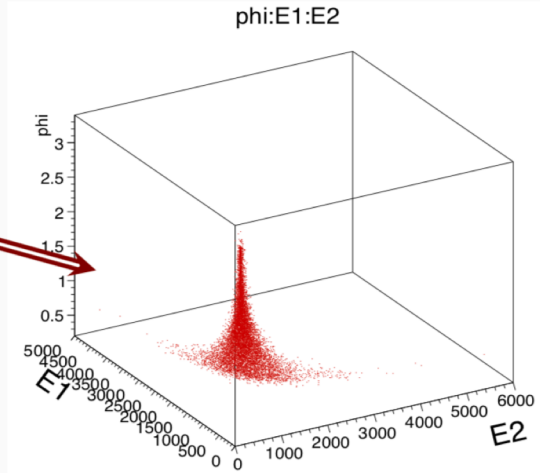
The toy model

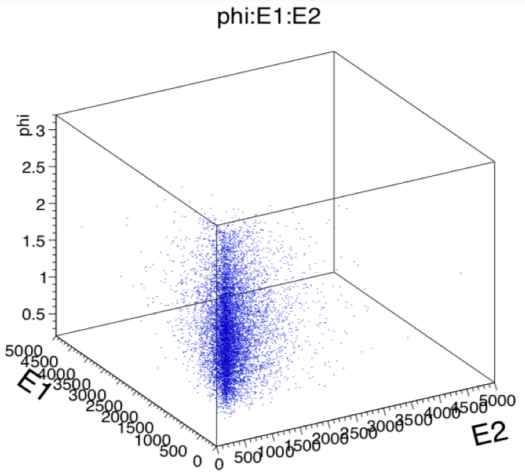




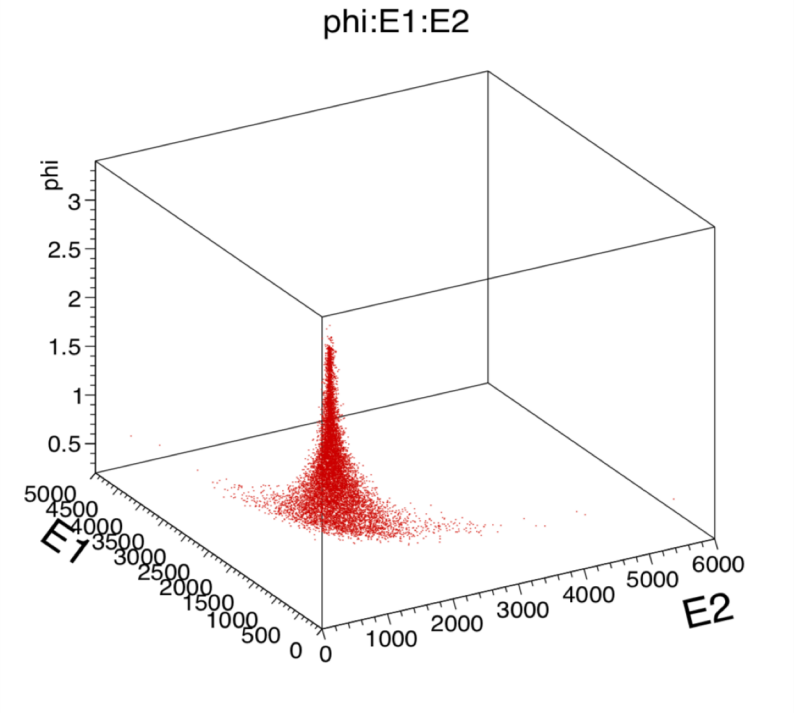


Identical marginal distribution but visible correlation in signal 3d-plot

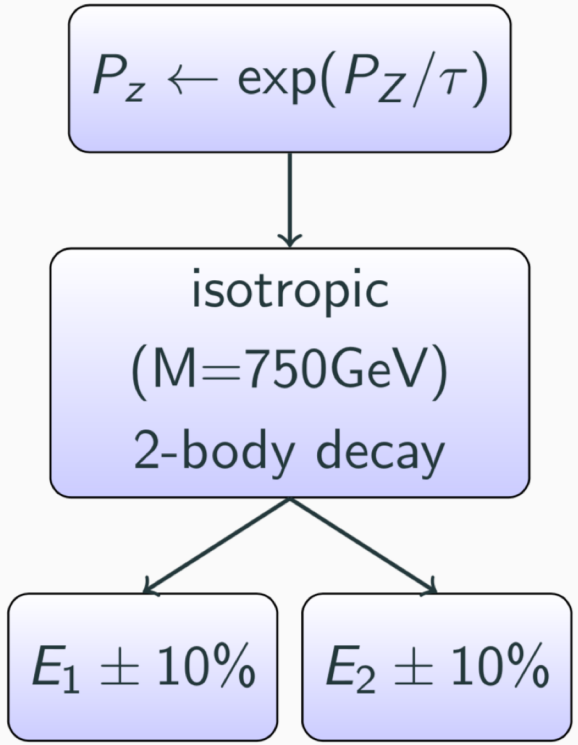




The toy model



The used minimalistic toy "Monte Carlo":



similar smearing for momentum

Thank you

Contact

DESY. Deutsches
Elektronen-Synchrotron

www.desy.de

Dirk Krücker
CMS
dirk.kruecker@desy.de