In [ ]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader
```

Autograd

Computational graph:

- **Input** tensors are the **leaves**
- **Output** tensor is the **root**
- Every **operation** is a **node**

When you call `.backward()` on the output tensor, autograd traverses the graph from the root to the leaves, calculating and storing the gradients of each **leaf** (input) tensor in its `.grad` attribute.

If you call `.retain_grad()` on a non-leaf tensor (an intermediate result), its gradient will also be computed and stored in its `.grad` attribute during the backward pass. If not, only the gradients of leaf tensors will be retained.

In [4]:
```
x = torch.ones(2,2, requires_grad=True)     # leaf node (input)
y = x + 2                                   # non-leaf (intermediate) result
z = y * y * 3                               # result = root node
out = z.mean()

y.retain_grad() # retain gradients for non-leaf (intermediate) nodes
out.backward()  # compute gradients, starting from root z and going backwards to leaves

# print(z.grad) # None, because z doesn't have a next node
print(y.grad)   # d(out)/d(y) with y \in R^{2x2} (only because we called y.retain_grad())
print(x.grad)   # d(out)/d(x) with x \in R^{2x2}
```

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

`x.grad` computes the gradient of `out` with respect to $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 \end{bmatrix}$:

$$\nabla_x \texttt{out} = \begin{bmatrix} \frac{\partial \texttt{out}}{\partial \mathbf{x}_1} & \frac{\partial \texttt{out}}{\partial \mathbf{x}_2} \end{bmatrix} = \begin{bmatrix} \frac{\partial \texttt{out}}{\partial x_{11}} & \frac{\partial \texttt{out}}{\partial x_{12}} \\ \frac{\partial \texttt{out}}{\partial x_{21}} & \frac{\partial \texttt{out}}{\partial x_{22}} \end{bmatrix}$$

Let's denote the tensor `out` with o:

$$o = \frac{1}{4}\sum_{i,j} z_{ij}, \qquad z_{ij} = 3y^2, \qquad y = x_{ij} + 2$$

Let's compute the derivative, given $x_{ij} = 1$:

$$\begin{aligned} \frac{\partial o}{\partial x_{ij}} &= \frac{\partial o}{\partial z_{ij}} \frac{\partial z_{ij}}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial x_{ij}} \\ &= \frac{1}{4} \cdot 6(x_{ij} + 2) \cdot 1 \\ &= \frac{3}{2}(x_{ij} + 2) \\ &= \frac{3}{2}(1 + 2) \\ &= \frac{9}{2} = 4.5 \end{aligned}$$

Let's treat the tensor as a flattened vector $\mathbf{x} \in \mathbb{R}^4$. We define the chain of functions:

1. $\mathbf{y} = \mathbf{x} + 2\mathbf{1}$
2. $\mathbf{z} = 3(\mathbf{y} \odot \mathbf{y})$
3. $o = \frac{1}{4}\sum z_i = \frac{1}{4}\mathbf{1}^\top \mathbf{z}$

Using the multivariate Chain Rule, the gradient is the product of Jacobians:

$$\frac{\partial o}{\partial \mathbf{x}} = \frac{\partial o}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

Let's calculate the three Jacobians step-by-step.

Step 1 (Mean): The derivative of a sum/mean is a row vector.

$$\frac{\partial o}{\partial \mathbf{z}} = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{bmatrix}$$

Step 2 (Element-wise Square): Since z_i only depends on y_i, the off-diagonal entries are 0. This results in a **Diagonal Jacobian Matrix**.

$$\frac{\partial \mathbf{z}}{\partial \mathbf{y}} = \begin{bmatrix} \frac{\partial z_1}{\partial y_1} & 0 & \dots & 0 \\ 0 & \frac{\partial z_2}{\partial y_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial z_4}{\partial y_4} \end{bmatrix} = \begin{bmatrix} 6y_1 & 0 & 0 & 0 \\ 0 & 6y_2 & 0 & 0 \\ 0 & 0 & 6y_3 & 0 \\ 0 & 0 & 0 & 6y_4 \end{bmatrix}$$

Step 3 (Linear Add): The derivative is the Identity matrix.

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{I}_4$$

Multiplying the row vector by the diagonal matrix:

$$\begin{aligned} \frac{\partial o}{\partial \mathbf{x}} &= \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{bmatrix} \begin{bmatrix} 6y_1 & 0 & 0 & 0 \\ 0 & 6y_2 & 0 & 0 \\ 0 & 0 & 6y_3 & 0 \\ 0 & 0 & 0 & 6y_4 \end{bmatrix} \mathbf{I}_4 \\ &= \begin{bmatrix} \frac{1}{4}(6y_1) & \frac{1}{4}(6y_2) & \frac{1}{4}(6y_3) & \frac{1}{4}(6y_4) \end{bmatrix} \end{aligned}$$

Given input $x_{ij} = 1 \implies y_{ij} = 3$:

$$= \begin{bmatrix} 4.5 & 4.5 & 4.5 & 4.5 \end{bmatrix}$$

In Autograd engines (like PyTorch), we never explicitly construct the Jacobian matrix $\frac{\partial \mathbf{z}}{\partial \mathbf{y}}$ for element-wise operations because it would be an $N \times N$ matrix with mostly zeros.

Mathematically, multiplying a vector $\mathbf{v}$ by a diagonal matrix $\text{diag}(\mathbf{u})$ is equivalent to the **Hadamard (element-wise) product**:

$$\mathbf{v}^\top \cdot \text{diag}(\mathbf{u}) \equiv (\mathbf{v} \odot \mathbf{u})^\top$$

This is why we can simplify the notation to $\frac{\partial o}{\partial \mathbf{z}} \odot \frac{\partial \mathbf{z}}{\partial \mathbf{y}}$, as done in simpler derivations.

In [8]:
```
my_tensor = torch.rand(2,3, requires_grad=True)

print(my_tensor)

# multiply by two and assign the result to a new variable
x = my_tensor.multiply(2)

print(x)

# sum the variables elements
my_sum = x.sum()

# for intermediate steps, we can call .retain_grad() to keep track of the gradients
x.retain_grad()

# perform a backward pass on the last variable
my_sum.backward()
print(f"Gradient of my_sum w.r.t. my_tensor: {my_tensor.grad}") # input always has gradients
print(f"Gradient of my_sum w.r.t. x: {x.grad}") # x is not a leaf node but we called x.retain_grad()
print(f"Gradient of my_sum w.r.t. itself: {my_sum.grad}") # result = root node, so it does not have any gradien
```

```
tensor([[0.3858, 0.2446, 0.4414],
        [0.7755, 0.0104, 0.1654]], requires_grad=True)
tensor([[0.7716, 0.4891, 0.8828],
        [1.5511, 0.0208, 0.3308]], grad_fn=<MulBackward0>)
Gradient of my_sum w.r.t. my_tensor: tensor([[2., 2., 2.],
        [2., 2., 2.]])
Gradient of my_sum w.r.t. x: tensor([[1., 1., 1.],
        [1., 1., 1.]])
Gradient of my_sum w.r.t. itself: None
```

```
/var/folders/dn/qw26xkk51853rtytqd9ns_5r0000gn/T/ipykernel_15226/3036280057.py:20: UserWarning: The .grad attrib
ute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute won't be populated during autog
rad.backward(). If you indeed want the .grad field to be populated for a non-leaf Tensor, use .retain_grad() on
the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access the leaf Tensor instead.
See github.com/pytorch/pytorch/pull/30531 for more informations. (Triggered internally at /Users/runner/work/pyt
orch/pytorch/pytorch/build/aten/src/ATen/core/TensorBody.h:494.)
  print(f"Gradient of my_sum w.r.t. itself: {my_sum.grad}") # result = root node, so it does not have any gradie
nts
```

> ! The shape of the gradient `.grad` attribute **matches the shape of the original tensor.** !

Defining Neural Nets

In [ ]:
```python
class Net(nn.Module):

    def __init__(self):

        super(Net, self).__init__()

        # -------------------
        # define layers here
        # -------------------


    def forward(self, x):

        # -------------------------
        # define forward pass here
        # -------------------------

        x = ...
        x = ...

        return x
        # or return softmax(x, dim=1) for classification

net = Net()
```

Parameters and gradients:

In [ ]:
```python
# Print parameters
list(net.named_parameters())
list(net.parameters())

# Access a specific parameter
param = net.Wi
print(param.shape)
print(param.requires_grad)
print(param.is_leaf) # not created by an operation (e.g. W)
print(param.data)
print(param.grad)


# Calling .forward()
output = net(input)

# Compute gradients
output.backward(torch.randn(num_input, num_output))
# backward() only requires an argument when the output is not a scalar

# Set gradients to zero
# This is important because by default, gradients are accumulated in PyTorch.
net.zero_grad()
```

Loss and optimizers:

In [ ]:
```python
# Compute loss
loss = nn.MSELoss()(output, target)
loss = nn.CrossEntropyLoss()(output, target)

# Define optimizer:
optimizer = optim.SGD(
net.parameters(),   # what parameters to optimize
lr=0.01,            # learning rate \alpha
momentum=0.9,       # momentum \beta
weight_decay=1e-4,  # L2 regularization
)
# 1. SGD + Momentum:    learning rate 0.01 - 0.1
# 2. ADAM:              learning rate 3e-4 - 1e-5
# 3. RMSPROP:           somewhere between SGD and ADAM
```

```
# Zero gradients
optimizer.zero_grad()

# Compute gradients
loss.backward()

# Update parameters
optimizer.step()
```

Ballpark estimates of hyperparameters

- **Number of hidden units and network structure:**

  One rarely goes below **512** units for feedforward networks (unless your are training on CPU...). In general trial and error.

- **Parameter initialization:** Parameter initialization is extremely important. Often used initializer are

  1. Kaiming He
  2. Xavier Glorot
  3. Uniform or Normal with small scale (0.1 - 0.01)
  4. Orthogonal (this usually works very well for RNNs)

  Bias is nearly always initialized to zero using `torch.nn.init.constant(tensor, val)`

- **Mini-batch size:**

  Usually people use 16-256. Bigger is not allways better. With smaller mini-batch size you get more updates and your model might converge faster. Also small batch sizes use less memory, which means you can train a model with more parameters.

- **Nonlinearity:** The most commonly used nonliearities are

  1. ReLU
  2. Leaky ReLU
  3. ELU
  4. Sigmoid (rarely, if ever, used in hidden layers anymore, squashes the output to the interval [0, 1] - appropriate if the targets are binary.
  5. Tanh is similar to the sigmoid, but squashes to [-1, 1]. Rarely used any more.
  6. Softmax normalizes the output to 1, usrful if you have a multi-class classification problem.

- **Optimizer and learning rate:**

  1. SGD + Momentum: learning rate 0.01 - 0.1
  2. ADAM: learning rate 3e-4 - 1e-5
  3. RMSPROP: somewhere between SGD and ADAM

Defining a network:

In [ ]:
```
def __init__(self, num_features, num_hidden, num_output):
    super(Net, self).__init__()

    # Define activation function
    self.activation = F.relu # nn.ReLU()
    # x = F.leaky_relu(z, negative_slope=0.01)
    # x = F.elu(x)
    # x = torch.sigmoid(x)
    # x = torch.tanh(x)

    # Using nn.Linear...

    self.fc1 = nn.Linear(num_features, num_hidden)
    self.fc2 = nn.Linear(num_hidden, num_output)

    # Or manually...

    # OPT1: Xavier-Glorot initialization: \sigma = sqrt(2 / (in + out))
    #   ! Remember: size = (out, in) = (num_rows, num_columns) in x @ W.T + b
    #   ! Here we go from num_features --> num_hidden
    self.W1 = nn.Parameter(init.xavier_normal_(torch.Tensor(num_hidden, num_features)))
    self.b1 = nn.Parameter(init.constant_(torch.Tensor(num_hidden), 0))

    # OPT2: Kaiming-He initialization: \sigma = sqrt(2 / in)
    #   ! Remember: size = (out, in) = (num_rows, num_columns) in x @ W.T + b
    #   ! Here we go from num_hidden --> num_output
    self.W2 = nn.Parameter(init.kaiming_normal_(torch.Tensor(num_output, num_hidden)))
    self.b2 = nn.Parameter(init.constant_(torch.Tensor(num_output), 0))

    # Dropout
    self.dropout1 = nn.Dropout(p=0.5) # no size dependence
    self.dropout2 = nn.Dropout(p=0.5) # no size dependence
```

```
    # BatchNorm
    self.batchnorm1 = nn.BatchNorm1d(num_hidden) # size of layer
    self.batchnorm2 = nn.BatchNorm1d(num_output) #   to which we apply batchnorm
```

Using the layers in `forward()`:

In [ ]:
```
def forward(self, x):

    # ---- HIDDEN LAYER 1 ----

    # Apply fully connected layer...
    x = self.fc1(x)                     # ...using nn.Linear
    x = F.linear(x, self.W1, self.b1)    # ... or manually x @ W.T + b

    # Apply batchnorm BEFORE activation function
    x = self.batchnorm1(x)

    # Apply nonlinearity
    x = self.activation(x)

    # Apply dropout AFTER activation function
    x = self.dropout1(x)

    # ---- HIDDEN LAYER 2 ----

    x = self.fc2(x)                     # ...using nn.Linear
    x = F.linear(x, self.W2, self.b2)    # ... or manually x @ W.T + b

    # Apply batchnorm BEFORE activation function
    x = self.batchnorm2(x)

    # Apply activation function
    x = self.activation(x)

    # Apply dropout AFTER activation function
    x = self.dropout2(x)

    # ----- OUTPUT LAYER -----

    x = self.fc3(x)                     # ...using nn.Linear
    x = F.linear(x, self.W3, self.b3)    # ... or manually x @ W.T + b

    # NO SOFTMAX HERE
    #   IF USING CrossEntropyLoss
    return x
```

Using the layers in `forward()`:

In [ ]:
```
net = Net(num_features, num_hidden, num_output)
```

L1 regularization requires manually adding the sum of absolute values of parameters to the loss function:

In [ ]:
```
# L1 regularization
l1_lambda = 1e-5
l1_norm = sum(p.abs().sum() for p in net.parameters())

# MANUALLY add l1_lambda * l1_norm to your loss function during training.
criterion = nn.CrossEntropyLoss()
loss = criterion(output, target) + l1_lambda * l1_norm
```

CrossEntropyLoss expects raw **logits**, not probabilities. It applies SoftMax **internally** before computing the cross-entropy loss. That's why we didn't apply softmax in the last layer of the network

Training loop:

In [ ]:
```
# we could have done this ourselves,
# but we should be aware of sklearn and its tools
from sklearn.metrics import accuracy_score

# setting hyperparameters and gettings epoch sizes
batch_size = 100
num_epochs = 200
num_samples_train = x_train.shape[0]
num_batches_train = num_samples_train // batch_size
num_samples_valid = x_valid.shape[0]
num_batches_valid = num_samples_valid // batch_size

# setting up lists for handling loss/accuracy
train_acc, train_loss = [], []
valid_acc, valid_loss = [], []
test_acc, test_loss = [], []
cur_loss = 0
losses = []
```

```
get_slice = lambda i, size: range(i * size, (i + 1) * size)

for epoch in range(num_epochs):
    # Forward -> Backprob -> Update params

    ########
    ## Train
    ########

    cur_loss = 0 # sum all batch losses

    # set network to training mode
    net.train()

    # loop over all mini-batches
    for i in range(num_batches_train):

        # reset gradients
        optimizer.zero_grad()

        slce = get_slice(i, batch_size)
        output = net(x_train[slce])

        # compute gradients given loss
        target_batch = targets_train[slce]

        batch_loss = criterion(output, target_batch) # criterion = nn.CrossEntropyLoss()

        #------------
        ## If using L1 regularization, add the L1 penalty to the loss
        # l1_lambda = 1e-5
        # l1_norm = sum(p.abs().sum() for p in net.parameters())
        # batch_loss += l1_lambda * l1_norm
        #------------

        batch_loss.backward() # backpropagate gradients
        optimizer.step() # update parameters after this batch of data

        cur_loss += batch_loss

    # append average loss per batch for the epoch
    losses.append(cur_loss / batch_size)

    # set network to evaluation mode
    net.eval()

    ######################
    ### Evaluate training
    ######################

    train_preds, train_targs = [], []

    # loop over all mini-batches
    for i in range(num_batches_train):

        slce = get_slice(i, batch_size)

        # get outputs of final network (after training with all mini-batches, but within the epoch)
        output = net(x_train[slce])
        preds = torch.max(output, 1)[1] # most likely class indices
        train_targs += list(targets_train[slce].numpy())
        train_preds += list(preds.data.numpy())

    ########################
    ### Evaluate validation
    ########################

    val_preds, val_targs = [], []

    for i in range(num_batches_valid):

        slce = get_slice(i, batch_size)

        # get outputs of final network (after training with all mini-batches, but within the epoch)
        output = net(x_valid[slce])
        preds = torch.max(output, 1)[1] # most likely class indices
        val_targs += list(targets_valid[slce].numpy())
        val_preds += list(preds.data.numpy())


    # compute accuracy
    train_acc_cur = accuracy_score(train_targs, train_preds)
    valid_acc_cur = accuracy_score(val_targs, val_preds)

    train_acc.append(train_acc_cur)
```

```
    valid_acc.append(valid_acc_cur)

    if epoch % 10 == 0:
        print("Epoch %2i : Train Loss %f , Train acc %f, Valid acc %f" % (
                epoch+1, losses[-1], train_acc_cur, valid_acc_cur))

epoch = np.arange(len(train_acc))
plt.figure()
plt.plot(epoch, train_acc, 'r', epoch, valid_acc, 'b')
plt.legend(['Train Accucary','Validation Accuracy'])
plt.xlabel('Updates'), plt.ylabel('Acc')
```

Convolutional layers:

In [ ]:

```
def __init__(self, in_channels, out_channels, kernel_size, stride, padding):

    super(Model, self).__init__()

    # ---
    # CONVOLUTIONAL layer
    #   to change number of channels
    # ---

    # Keep spatial dimensions (uneven k!)
    stride = 1
    padding = (kernel_size - 1) // 2

    # Convert   x \in R^{batch_size x in_channels x H x W}
    # to        x \in R^{batch_size x out_channels x H' x W'}
    self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)

    # ---
    # POOLING layer
    #   to reduce spatial dimensions H and W,
    #   introduce translational invariance,
    #   reduce computation by factor 4
    # ---

    # Convert   x \in R^{batch_size x out_channels x H' x W'}
    # to        x \in R^{batch_size x out_channels x H'/2 x W'/2}
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
    # Halve H and W

    # ---
    # FULLY CONNECTED layer
    #   for the output (classification/regression)
    # ---

    # Convert  x \in R^{batch_size x (out_channels * H'' * W'')}
    # to       x \in R^{batch_size x num_output}
    flattened_size = out_channels * H_final * W_final
    self.fc = nn.Linear(flattened_size, num_output)

    # Or a deeper (classification) network with more FC layers
    self.fc = nn.Sequential(
        nn.Flatten(),
        nn.Linear(flattened_size, hidden_size),
        activation_fn,
        nn.Linear(hidden_size, self.num_classes),
        # No softmax here, as CrossEntropyLoss does that internally
    )

def forward(self, x):

    # x \in R^{batch_size x in_channels x H x W}
    identity = x

    # Apply convolutional layer
    out = self.conv(x)          # R^{batch_size x out_channels x H' x W'}
    out = self.bn(out)          # apply batchnorm (optional)

    # Change number of channels in identity
    if identity.shape[1] != out.shape[1]:
        identity = self.conv_identity(identity) # 1x1 conv to change channels

    # residual connection
    out = out + identity

    # Apply activation function
    out = F.relu(out)

    # Apply pooling layer
    out = self.pool(out)          # R^{batch_size x out_channels x H'/2 x W'/2}

    # Apply fully connected layer
```

```
    # (flattening is done inside this layer)
    out = self.fc(out)            # R^{batch_size x num_output}

    return out
```

In order to keep H and W the same after a convolution with kernel size k and stride $s = 1$, use padding $p = \frac{k-1}{2}$ for odd k:

```
# Assuming s=1, k=odd
conv_padding = (kernel_size - 1) // 2  # for odd kernel_size
```

For even k, it is not possible to keep the same size with symmetric padding.

In case of stride $s > 1$, the output size is given by:

$$H_{out} = \left\lfloor \frac{H_{in} + 2p - k}{s} \right\rfloor + 1, \quad W_{out} = \left\lfloor \frac{W_{in} + 2p - k}{s} \right\rfloor + 1$$

In this case of $s > 1$, we cannot keep the output size the same as the input size. Stride implies downsampling, i.e. skipping pixels. If $s = 2$, we keep roughly half the pixels in each dimension:

$$H_{out} \approx \frac{H_{in}}{s}, \quad W_{out} \approx \frac{W_{in}}{s}$$

Recurrent Neural Networks

He intialization normally sets

$$\mathrm{Var}(w) = \frac{2}{\mathrm{num\_in}}$$

However, in RNN's, since we have both input and hidden contributions, and they have unit variance, their sum has variance 2. Therefore, to keep the overall variance of the pre-activation at 1, we set:

$$\mathrm{Var}(w) = \frac{1}{\mathrm{num\_in}}$$

In [ ]:
```
# RNN in PyTorch
class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(RNNModel, self).__init__()
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.rnn(x)  # out: tensor of shape (batch_size, seq_length, hidden_size)
        out = self.fc(out[:, -1, :])  # get the last time step
        return out
```

LSTM

In PyTorch:

In [ ]:
```
class MyRecurrentNet(nn.Module):

    # A simple LSTM for predicting the next word in a sequence

    def __init__(self):

        super(MyRecurrentNet, self).__init__()

        # vocab_size = number of unique words in vocabulary

        # Recurrent layer
        self.lstm = nn.LSTM(input_size=vocab_size,
                            hidden_size=50,         # number of features in hidden state
                            num_layers=1,
                            batch_first=True)       # input & output tensors are provided as
                                                    # (batch, seq, feature)

        # Output layer
        self.l_out = nn.Linear(in_features=50,      # must match hidden_size above
                               out_features=vocab_size,# must match number of classes
                               bias=False)          # no bias needed here as we have a bias in the LSTM layer

    def forward(self, x):

        # LSTM.forward() returns output x and LAST hidden state (h, c)

        x, (h, c) = self.lstm(x)
```

```
        # x = output features from the last layer of the LSTM, for each timestep
        #       -> per batch, this is a sequence of (h_0, h_1, ..., h_{seq_length-1})
        #       -> shape (batch_size, seq_length, hidden_size)
        #
        # h = hidden state for last timestep t=seq_length
        #
        # c = cell state for last timestep t=seq_length

        # Flatten output for feed-forward layer
        x = x.view(-1, self.lstm.hidden_size)

        # view takes 2 arguments:
        #   -> num_rows, num_columns
        #   -> -1 means "infer this dimension based on the other one"

        # Now, x has shape (batch_size*seq_length, hidden_size)

        # Output layer
        x = self.l_out(x)

        # l_out returns a 2D tensor of shape (batch_size*seq_length, vocab_size)
        # and contains the logits for each class (word in vocabulary)

        return x

net = MyRecurrentNet()
```

Transformer

In [ ]:

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -math.inf)
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn

class MultiHeadedAttention(nn.Module):
    """A simple Multi-head attention layer."""
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        self.linears = nn.ModuleList([nn.Linear(d_model, d_model) for _ in range(4)])
        self.attn = None # store the attention maps
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        nbatches = query.size(0)
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)

        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2) for l, x in zip(self.lin

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(query, key, value, mask=mask, dropout=self.dropout)

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

Variational Autoencoder

In [ ]:

```
class VariationalAutoencoder(nn.Module):
    """A Variational Autoencoder with
    * a Bernoulli observation model `p_\theta(x | z) = B(x | g_\theta(z))`
    * a Gaussian prior `p(z) = N(z | 0, I)`
    * a Gaussian posterior `q_\phi(z|x) = N(z | \mu(x), \sigma(x))` (not \sigma * I!)
    """

    def __init__(self, input_shape:torch.Size, latent_features:int) -> None:
        super(VariationalAutoencoder, self).__init__()

        self.input_shape = input_shape
        self.latent_features = latent_features
```

```
        self.observation_features = np.prod(input_shape)


        # Inference Network
        # Encode the observation `x` into the parameters of the posterior distribution
        # `q_\phi(z|x) = N(z | \mu(x), \sigma(x)), \mu(x),\log\sigma(x) = h_\phi(x)`
        self.encoder = nn.Sequential(
            nn.Linear(in_features=self.observation_features, out_features=256),
            nn.ReLU(),
            nn.Linear(in_features=256, out_features=128),
            nn.ReLU(),
            # A Gaussian is fully characterised by its mean \mu and variance \sigma**2
            nn.Linear(in_features=128, out_features=2*latent_features)
            #! note the 2*latent_features !!
            # we predict \mu \in R^d and \log(\sigma) \in R^d (we don't assume \sigma * I)
        )

        # Generative Model
        # Decode the latent sample `z` into the parameters of the observation model
        # `p_\theta(x | z) = \prod_i B(x_i | g_\theta(x))`
        self.decoder = nn.Sequential(
            nn.Linear(in_features=latent_features, out_features=128),
            nn.ReLU(),
            nn.Linear(in_features=128, out_features=256),
            nn.ReLU(),
            nn.Linear(in_features=256, out_features=self.observation_features)
        )

        # define the parameters of the prior, chosen as p(z) = N(0, I)
        self.register_buffer('prior_params', torch.zeros(torch.Size([1, 2*latent_features])))

    def posterior(self, x:Tensor) -> Distribution:
        """return the distribution `q(x|x) = N(z | \mu(x), \sigma(x))`"""

        # compute the parameters of the posterior
        h_x = self.encoder(x)
        mu, log_sigma =  h_x.chunk(2, dim=-1)
        # we learn log(sigma) because sigma must be positive

        # return a distribution `q(x|x) = N(z | \mu(x), \sigma(x))`
        return ReparameterizedDiagonalGaussian(mu, log_sigma)

    def prior(self, batch_size:int=1)-> Distribution:
        """return the distribution `p(z)`"""
        prior_params = self.prior_params.expand(batch_size, *self.prior_params.shape[-1:])
        mu, log_sigma = prior_params.chunk(2, dim=-1)

        # return the distribution `p(z)`
        return ReparameterizedDiagonalGaussian(mu, log_sigma)

    def observation_model(self, z:Tensor) -> Distribution:
        """return the distribution `p(x|z)`"""
        px_logits = self.decoder(z)
        px_logits = px_logits.view(-1, *self.input_shape) # reshape the output
        return Bernoulli(logits=px_logits, validate_args=False)


    def forward(self, x) -> Dict[str, Any]:
        """compute the posterior q(z|x) (encoder), sample z~q(z|x) and return the distribution p(x|z) (decoder)

        # flatten the input
        x = x.view(x.size(0), -1)

        # define the posterior q(z|x) / encode x into q(z|x)
        qz = self.posterior(x)

        # define the prior p(z)
        pz = self.prior(batch_size=x.size(0))

        # sample the posterior using the reparameterization trick: z ~ q(z | x)
        z = qz.rsample()

        # define the observation model p(x|z) = B(x | g(z))
        px = self.observation_model(z)

        return {'px': px, 'pz': pz, 'qz': qz, 'z': z}


    def sample_from_prior(self, batch_size:int=100):
        """sample z~p(z) and return p(x|z)"""

        # degine the prior p(z)
        pz = self.prior(batch_size=batch_size)

        # sample the prior
        z = pz.rsample()
```

```
        # define the observation model p(x|z) = B(x | g(z))
        px = self.observation_model(z)

        return {'px': px, 'pz': pz, 'z': z}

def reduce(x:Tensor) -> Tensor:
    """for each datapoint: sum over all dimensions"""
    return x.view(x.size(0), -1).sum(dim=1)

class VariationalInference(nn.Module):
    def __init__(self, beta:float=1.):
        super().__init__()
        self.beta = beta

    def forward(self, model:nn.Module, x:Tensor) -> Tuple[Tensor, Dict]:

        # forward pass through the model
        outputs = model(x)

        # unpack outputs
        px, pz, qz, z = [outputs[k] for k in ["px", "pz", "qz", "z"]]

        # evaluate log probabilities
        log_px = reduce(px.log_prob(x))
        log_pz = reduce(pz.log_prob(z))
        log_qz = reduce(qz.log_prob(z))

        # compute the ELBO with and without the beta parameter:
        # `L^\beta = E_q [ log p(x|z) ] - \beta * D_KL(q(z|x) | p(z))`
        # where `D_KL(q(z|x) | p(z)) = log q(z|x) - log p(z)`
        kl = log_qz - log_pz
        elbo = log_px - kl
        beta_elbo = log_px - self.beta * kl

        # loss
        loss = -beta_elbo.mean()

        # prepare the output
        with torch.no_grad():
            diagnostics = {'elbo': elbo, 'log_px':log_px, 'kl': kl}

        return loss, diagnostics, outputs

vi = VariationalInference(beta=1.0)
loss, diagnostics, outputs = vi(vae, images)
```