Evaluating Uncertainty Quantification in Medical Image Segmentation: A Multi-Dataset, Multi-Algorithm Study

Nyaz Jalal, **Małgorzata Sliwińska**, Wadim Wojciechowski, Iwona Kucybała, Miłosz Rozynek, Kamil Krupa, Patrycja Matusik, Jarosław Jarczewski, **Zbisław Tabor**

Applied Sciences 2024, 14, 10020.

Maximum likelihood principle

Let's throw the coin ten times. We observe heads 7 times out of 10. What is the probability of throwing heads?

Maximum likelihood principle

$$P(p|k,N)=\binom{N}{k}p^k(1-p)^{N-k}$$

$$\frac{dP}{dp}=0$$

$$\frac{dP}{dp}=\binom{N}{k}kp^{k-1}(1-p)^{N-k}-\binom{N}{k}p^k(N-k)(1-p)^{N-k-1}=0$$

$$p=k/N$$

Maximum likelihood principle

$$f(y;\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

$$L(a,b) = f(y_1,\mu_{x_2},\sigma) \cdot f(y_2,\mu_{x_2},\sigma) \cdots = \prod_{i=1}^{n} f(y_i;\mu_i,\sigma) = \prod_{i=1}^{n} f(y_i;(a \cdot x_i + b),\sigma)$$

$$l(a,b) = -\log(L(a,b)) = -\log\left(\prod_{i=1}^{n} f(y_i;\mu_i,\sigma)\right) = -\prod_{i=1}^{n} \log(f(y_i;(a \cdot x_i + b),\sigma))$$

$$w = \operatorname{argmin}_w \left\{ \frac{1}{2\sigma^2} \sum_{i=1}^{n} (\mu_{x_i} - y_i)^2 \right\}$$

Maximum likelihood principle

Non-constant variance

$$w = \text{argmin}_w \left\{ \sum_{i=1}^{n} -\log \left(\frac{1}{\sqrt{2\pi\sigma(x_i, w)^2}} \right) + \frac{(\mu(x_i, w) - y_i)^2}{2\sigma(x_i, w)^2} \right\}$$

Maximum likelihood principle

$$P(Y = k|X = x, W = w) = \begin{cases} p_0(x, w) & \text{for } k = 0 \\ p_1(x, w) & \text{for } k = 1 \end{cases} \text{ with } \sum p_i = 1$$

$$w = \text{argmax}_w \left\{ \prod_{i=1}^{n} P(Y = y_i \;\; x_i, w) \right\}$$

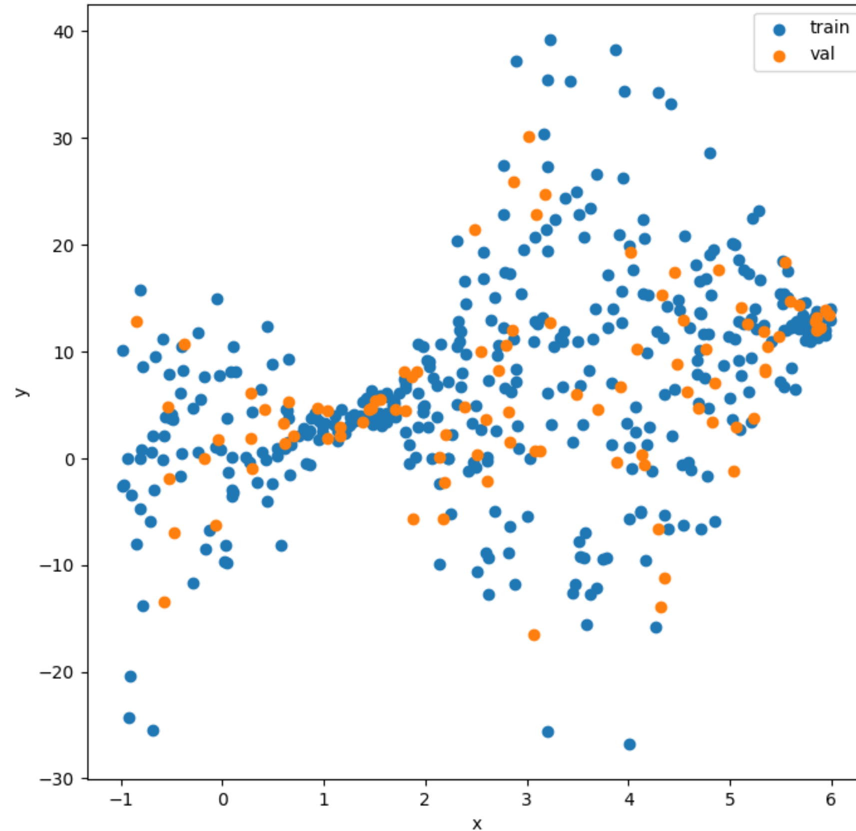
$$= \text{argmax}_w \left\{ \prod_{i \text{ with } y_i = 0}^{n} P(Y = 0 \;\; x_i, w) \quad \prod_{i \text{ with } y_i = 1}^{n} P(Y = 1 \;\; x_i, w) \right\}$$

Maximum likelihood principle

$$\text{crossentropy} = -\frac{1}{n}\Big(\sum_{j \text{ with } y_j=0} \log(p_0(x_j)) + \sum_{j \text{ with } y_j=1} \log(p_1(x_j))\Big)$$

$$\text{crossentropy} = -\frac{1}{n}\Big(\sum_{j=1}^{n}(y_i \cdot \log(p_1(x_i)) + (1-y_i)\cdot\log(1-p_1(x_i)))\Big)$$

Probabilistic neural networks



Base deterministic model

```python
lr = 0.001

torch.manual_seed(42)
# Now we can create a model and send it at once to the device
baseModel = nn.Sequential(nn.Linear(1, 1)).to(device)

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer = optim.SGD(baseModel.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

n_epochs = 1000

losses = []
for epoch in range(n_epochs):
    # Sets model to TRAIN mode
    baseModel.train()

    # Step 1 - Computes model's predicted output - forward pass
    yhat = baseModel(x_train_tensor)

    # Step 2 - Computes the loss
    loss = loss_fn(yhat, y_train_tensor)

    losses.append(loss.item())
    # Step 3 - Computes gradients for both "b" and "w" parameters
    loss.backward()

    # Step 4 - Updates parameters using gradients and
    # the learning rate
    optimizer.step()
    optimizer.zero_grad()
```

Probabilistic neural networks

```python
# Define the model
class MyModel1(nn.Module):
    def __init__(self):
        super(MyModel1, self).__init__()
        self.dense1 = nn.Linear(1, 1)
        self.hidden1 = nn.Linear(1, 30)
        self.hidden2 = nn.Linear(30, 20)
        self.hidden3 = nn.Linear(20, 20)
        self.out2 = nn.Linear(20, 1)

    def forward(self, x):
        out1 = self.dense1(x)
        hidden1 = torch.relu(self.hidden1(x))
        hidden2 = torch.relu(self.hidden2(hidden1))
        hidden3 = torch.relu(self.hidden3(hidden2))
        out2 = self.out2(hidden3)

        # Concatenate out1 and out2
        params = torch.cat([out1, out2], dim=-1)

        return params
```

$$w = \operatorname{argmin}_w \left\{ \sum_{i=1}^{n} -\log\left(\frac{1}{\sqrt{2\pi\sigma(x_i,w)^2}}\right) + \frac{(\mu(x_i,w)-y_i)^2}{2\sigma(x_i,w)^2} \right\}$$

```python
def heteroscedastic_loss(y_pred,y_true):
    # Extract mu and sigma from y_pred

    mu = y_pred[:, 0:1]  # First column is mu
    sigma = torch.exp(y_pred[:, 1:2])  # Second column is sigma, exponentiate it

    # Compute constant a
    a = 1 / (math.sqrt(2. * math.pi) * sigma)

    # Compute b
    b1 = torch.square(mu - y_true)
    b2 = 2 * torch.square(sigma)
    b = b1 / b2

    # Compute the loss
    loss = torch.sum(-torch.log(a) + b, dim=0)
    return loss
```

Probabilistic neural networks

```python
# Sets learning rate - this is "eta" ~ the "n"-like Greek letter
lr = 0.001

torch.manual_seed(42)
# Now we can create a model and send it at once to the device
#model1 = nn.Sequential(nn.Linear(1, 5),nn.ReLU(),nn.Linear(5, 2)).to(device)
model1 = MyModel1().to(device)

# Defines a SGD optimizer to update the parameters
# (now retrieved directly from the model)
optimizer = optim.AdamW(model1.parameters(), lr=lr)


# Defines number of epochs
n_epochs = 10

losses = []
for epoch in range(n_epochs):
    # Sets model to TRAIN mode
    model1.train()

    # Step 1 - Computes model's predicted output - forward pass
    yhat = model1(x_train_tensor)

    # Step 2 - Computes the loss
    loss = heteroscedastic_loss(yhat, y_train_tensor)

    losses.append(loss.item())
    # Step 3 - Computes gradients for both "b" and "w" parameters
    loss.backward()

    # Step 4 - Updates parameters using gradients and
    # the learning rate
    optimizer.step()
    optimizer.zero_grad()
```

Probabilistic neural networks

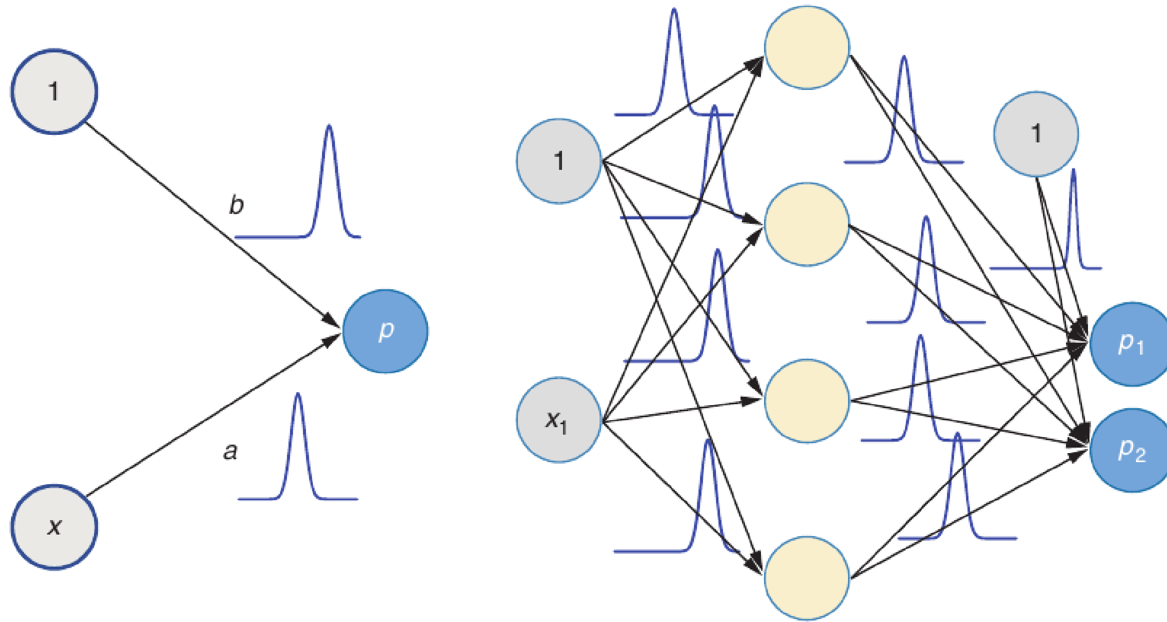
```python
from torch.distributions import Normal

# Define the model
class MyProbabilisticModel(nn.Module):
    def __init__(self):
        super(MyProbabilisticModel, self).__init__()
        self.dense1 = nn.Linear(1, 1)
        self.hidden1 = nn.Linear(1, 30)
        self.hidden2 = nn.Linear(30, 20)
        self.hidden3 = nn.Linear(20, 20)
        self.out2 = nn.Linear(20, 1)

    def forward(self, x):
        out1 = self.dense1(x)
        hidden1 = torch.relu(self.hidden1(x))
        hidden2 = torch.relu(self.hidden2(hidden1))
        hidden3 = torch.relu(self.hidden3(hidden2))
        out2 = self.out2(hidden3)

        # Concatenate out1 and out2
        params = torch.cat([out1, out2], dim=-1)

        # Use the custom distribution
        loc = params[:, 0:1]
        scale = 1e-3 + torch.nn.functional.softplus(0.05*params[:, 1:2])

        return Normal(loc, scale)


# Define the negative log-likelihood loss function
def NLL(y, distr):
    return -distr.log_prob(y).mean()
```

```python
lr = 0.001
nepochs = 10
# Initialize the model and optimizer
model2 = MyProbabilisticModel()
optimizer = optim.Adam(model2.parameters(), lr=lr)

losses = []
for epoch in range(nepochs):
  model2.train()
  optimizer.zero_grad()
  dist = model2(x_train_tensor)
  loss = NLL(y_train_tensor, dist)
  loss.backward()
  optimizer.step()
  losses.append(loss.item())
```

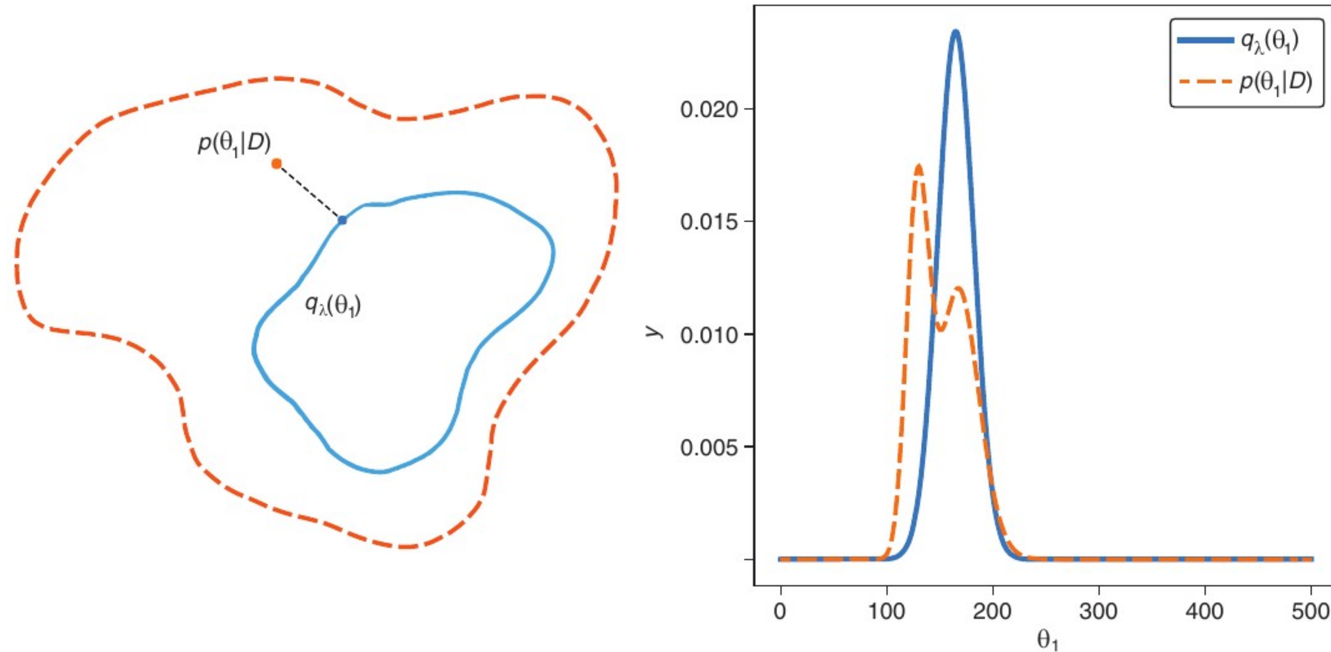
Probabilistic neural networks

```
model2.eval()

N_SAMPLES = 300
rng = np.arange(-5,10,0.5)
means = []
stds = []
for x in rng:
  x_test = torch.as_tensor([x]).float().to(device).view(-1,1)
  with torch.no_grad():
    preds = model2(x_test).sample((N_SAMPLES,)).squeeze().detach().cpu().numpy()
    means.append(np.mean(preds))
    stds.append(np.std(preds))
```

Bayesian probabilistic neural networks



Bayesian probabilistic neural networks



The principle idea of variational inference (VI). The larger region on the left depicts the space of all possible distributions, and the dot in the upper left represents the posterior $p(\theta_1|D)$, corresponding to the dotted density in the right panel. In the left panel, the inner region depicts the space of possible variational distributions. The optimized variational distribution, illustrated by the point in the inner loop, corresponds to the solid density displayed in the right panel, which has the smallest distance to the posterior as shown by the dotted line.

Bayesian probabilistic neural networks

Kullback – Leibler divergence

$$\mathrm{KL}[q_\lambda(\theta)||p(\theta|D)] = \int q_\lambda(\theta)\log\frac{q_\lambda(\theta)}{p(\theta|D)}d\theta$$

$$\lambda^* = \mathrm{argmin}\{\mathrm{KL}[q_\lambda(\theta)||p(\theta)] - E_{\theta\sim q_\lambda}[\log(p(D|\theta)]\}$$

Bayesian probabilistic neural networks

```
class BayesianDense(nn.Module):
    def __init__(self, in_features, out_features, activation_fn=F.relu, prior_std=1.0):
        super(BayesianDense, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.activation_fn = activation_fn
        self.prior_std = prior_std
        self.prior = Normal(0, self.prior_std)

        # Parameters
        self.weight_mu = nn.Parameter(torch.randn(out_features, in_features),requires_grad=True)
        self.weight_rho = nn.Parameter(torch.randn(out_features, in_features),requires_grad=True)
        self.bias_mu = nn.Parameter(torch.randn(out_features),requires_grad=True)
        self.bias_rho = nn.Parameter(torch.randn(out_features),requires_grad=True)

    def forward(self, x):
        weight_eps = torch.randn_like(self.weight_mu)
        weight_std = 1e-3 + torch.nn.functional.softplus(0.05*self.weight_rho) #torch.exp(self.weight_rho)
        weight = self.weight_mu + weight_std * weight_eps

        bias_eps = torch.randn_like(self.bias_mu)
        bias_std = 1e-3 + torch.nn.functional.softplus(0.05*self.bias_rho)  #torch.log1p(torch.exp(self.bias_rho))
        bias = self.bias_mu + bias_std * bias_eps

        output = torch.matmul(x, weight.t()) + bias

        p = Normal(self.weight_mu, weight_std)
        q = Normal(torch.zeros_like(self.weight_mu), torch.ones_like(self.weight_mu))
        kl_divergence_1 = torch.sum(kl_divergence(p, q))

        p = Normal(self.bias_mu, bias_std)
        q = Normal(torch.zeros_like(self.bias_mu), torch.ones_like(self.bias_mu))
        kl_divergence_2 = torch.sum(kl_divergence(p, q))

        kl_div = kl_divergence_1 + kl_divergence_2

        return self.activation_fn(output),kl_div
```

Bayesian probabilistic neural networks

```python
# Define the model
class MyBayesianModel(nn.Module):
    def __init__(self):
        super(MyBayesianModel, self).__init__()
        self.dense1 = BayesianDense(1, 1)
        self.hidden1 = BayesianDense(1, 30)
        self.hidden2 = BayesianDense(30, 20)
        self.hidden3 = BayesianDense(20, 20)
        self.out2 = BayesianDense(20, 1)

    def forward(self, x):
        out1,kl1 = self.dense1(x)
        hidden1,kl2 = self.hidden1(x)
        hidden2,kl3 = self.hidden2(hidden1)
        hidden3,kl4 = self.hidden3(hidden2)
        out2,kl5 = self.out2(hidden3)

        # Concatenate out1 and out2
        params = torch.cat([out1, out2], dim=-1)

        # Use the custom distribution
        loc = params[:, 0:1]
        scale = 1e-3 + torch.nn.functional.softplus(0.05*params[:, 1:2])

        return Normal(loc, scale), kl1 + kl2 + kl3 + kl4 + kl5

# Define the negative log-likelihood loss function
def NLL(y, distr):
    return -distr.log_prob(y).mean()
```

Bayesian probabilistic neural networks

```
        return Normal(loc, scale)

# Define the negative log-likelihood loss function
def NLL(y, distr):
    return -distr.log_prob(y).mean()

model = MyModel()
```

```
dnn_to_bnn(model, const_bnn_prior_parameters)
```

```
lr = 0.001
nepochs = 10000

optimizer = optim.Adam(model.parameters(), lr=lr)

batch_size = x_train_tensor.shape[0]

losses = []
for epoch in range(nepochs):

  model.train()
  optimizer.zero_grad()

  dist = model(x_train_tensor)
  kl = get_kl_loss(model)
  nll_loss = NLL(y_train_tensor, dist)
  loss = nll_loss + kl / batch_size

  loss.backward()
  optimizer.step()

  losses.append(loss.item())
```

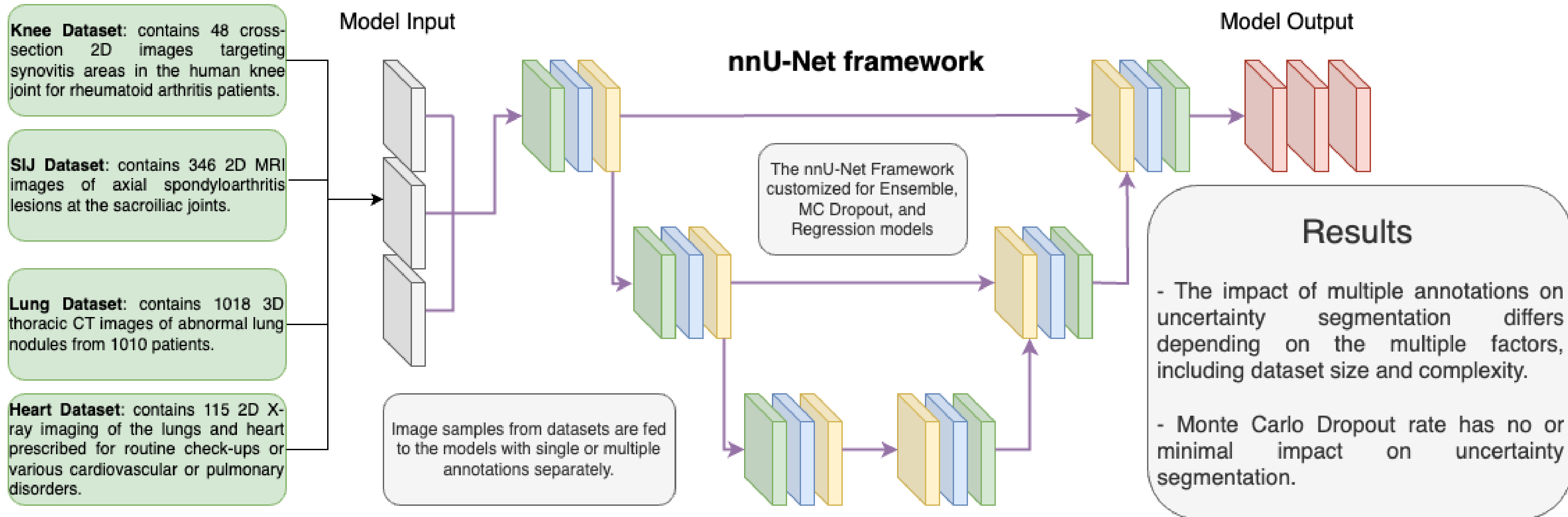
Evaluating Uncertainty

Table 1. Summary table for uncertainty approaches

Study	Feature 1*	Feature 2**	Feature 3***	Feature 4****	Feature 5 *****
Gudhe et al. (2023)	NO	YES	NO	NO	NO
Singh et al. (2022)	NO	NO	NO	YES	NO
Largent et al. (2022)	NO	NO	NO	YES	YES
Bórquez et al. (2023)	NO	NO	NO	YES	NO
Abdar et al. (2021)	YES	NO	NO	YES	YES
Hiasa et al. (2019)	NO	YES	NO	NO	NO
Weldebirhan Arega et al. (2021)	NO	NO	NO	NO	NO
Antico et al. (2020)	YES	NO	NO	YES	NO
Alonso-Caneiro et al. (2021)	NO	NO	NO	NO	NO
Wickstrøm et al. (2020)	NO	NO	NO	NO	NO
Song et al. (2022)	NO	NO	NO	NO	NO
Do et al. (2020)	NO	NO	NO	NO	NO
Roy et al. (2019)	NO	YES	NO	NO	NO
Ng et al. (2023)	NO	NO	NO	YES	YES
Camarasa et al. (2020)	NO	NO	NO	YES	NO
Hann et al. (2021)	NO	NO	NO	NO	NO
Zhao et al. (2022)	YES	NO	NO	NO	YES
Baumgartner et al. (2019)	YES	YES	YES	NO	NO
Kohl et al. (2018)	NO	NO	YES	NO	YES
Our Work	YES	YES	YES	YES	YES

*Multi Medical Image Types **Multi Anatomic Locations ***Multi Ground Truth
**** Dropout Rate Influence ***** Model Comparison

Evaluating Uncertainty



Evaluating Uncertainty

Key contributions:

- The influence of multiple annotations on segmentation models based on public image segmentation datasets.
- The influence of dropout rates on the MC models.
- The quantification of uncertainty on different body parts and image types.
- The performance evaluation of three well-known DL architectures in an unbiased environment.
- The efficacy evaluation of various well-known measurement metrics based on our models and datasets.

Evaluating Uncertainty

Table 2. Summary of the features of the datasets.

Properties	KNEE	SIJ	LUNG	HEART
Total Num.	47	345	1338	114
Training Num.	31	267	1029	65
Testing Num.	16	78	309	49
Image Size	448 by 448	Vary	Vary	512 by 512
Image Type	MRI	MRI	CT	X-ray
Num. Objects*	1	1	1	2
Segmented objects	Synovitis	Bone Oedema	Lung Nodules	Lung and Heart
Num. Radiologists	3	3	3 or 4	3
Body Part	Knee	Sacroiliac Joint	Chest	Chest
Image Dimension	2D	2D	3D	2D

*Beside background

Evaluating Uncertainty

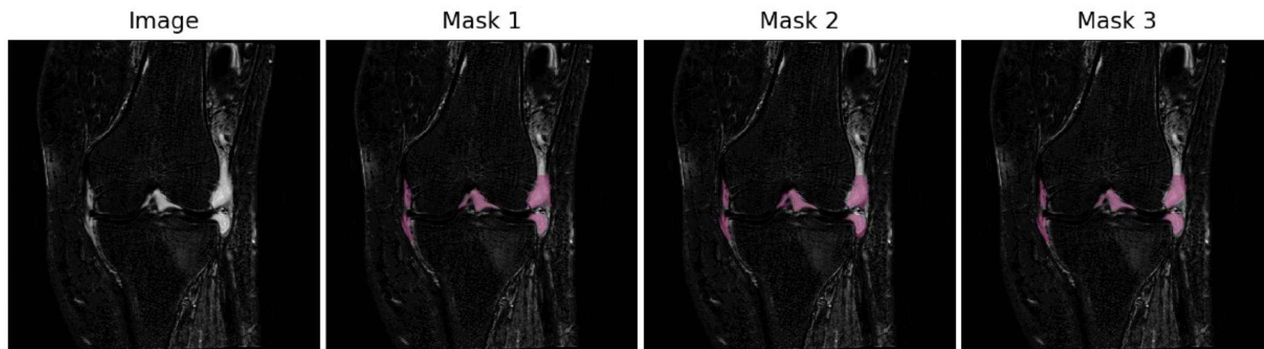


Fig. 1. The original image and three annotations by different radiologists from the KNEE dataset.

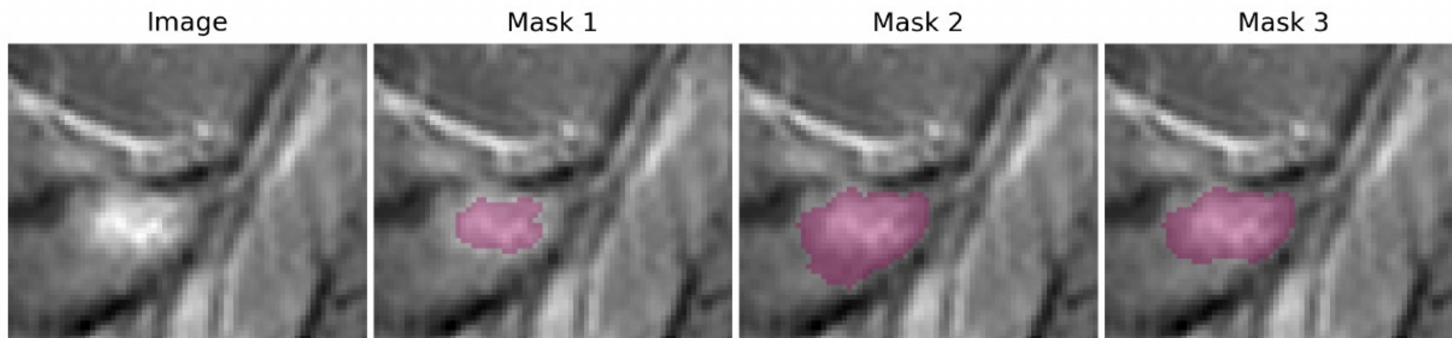


Fig. 2. The original image and three annotations by different radiologists from the SIJ dataset.

Evaluating Uncertainty

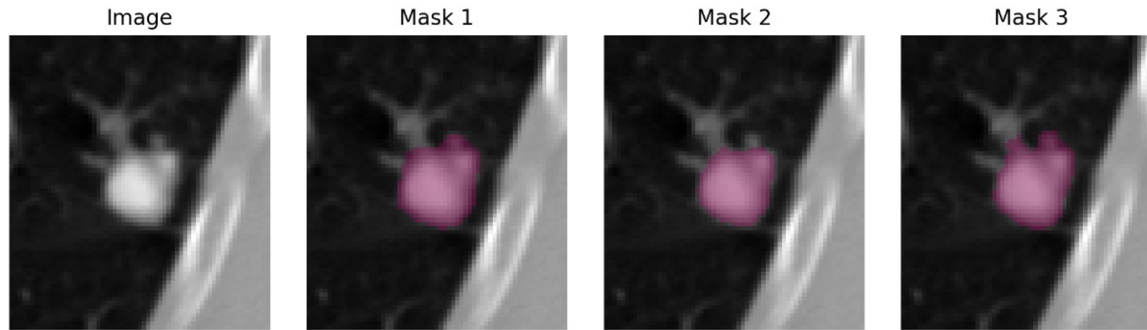


Fig. 3. The original image and three annotations by different radiologists from the LUNG dataset.

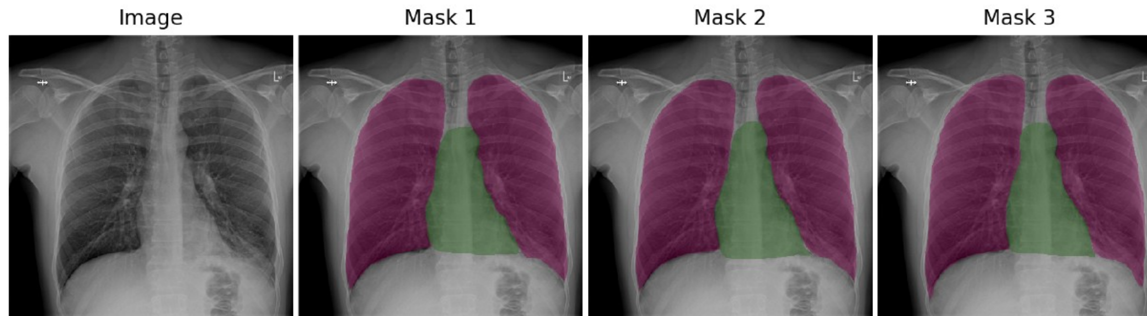


Fig. 4. The original image and three annotations by different radiologists from the HEART dataset.

Evaluating Uncertainty

In this study, we extended nnU-Net framework to train the following deep models for predicting segmentation (un)certainty:

- Ensemble models,
- MC dropout models,
- Regression models.

Evaluating Uncertainty

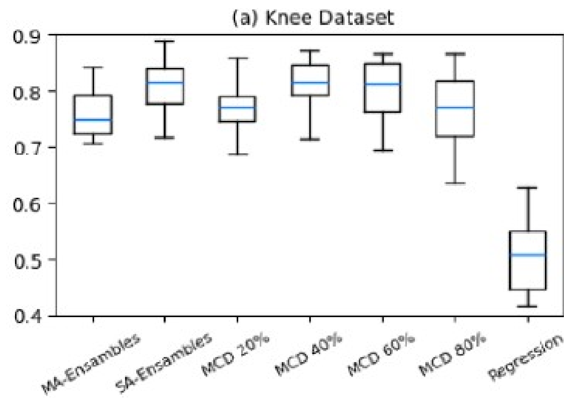
Metrics:

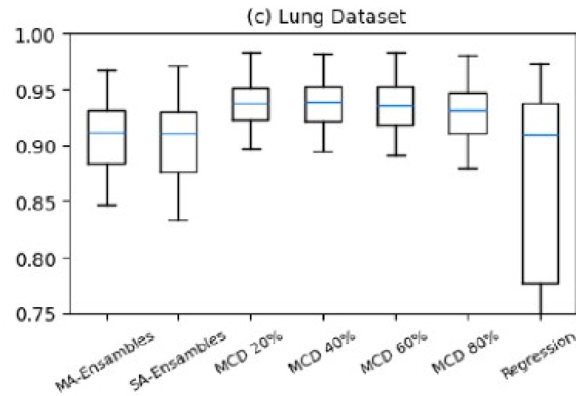
- Gray level Dice coefficient

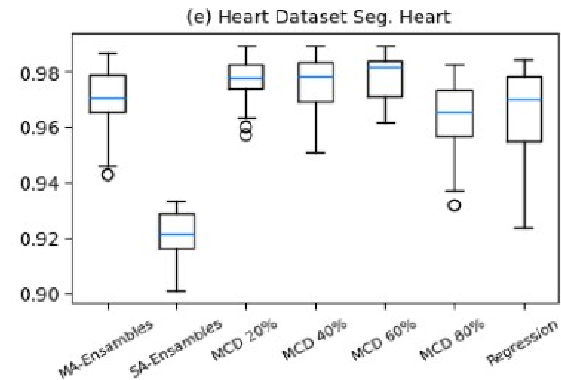
$$GDC = \frac{2 \times \sum_{i=1}^{M} \sum_{j=1}^{N} \sqrt{Ref_{(i,j)} * Seg_{(i,j)}}}{\sum_{i=1}^{M} \sum_{j=1}^{N} Ref_{(i,j)} + \sum_{i=1}^{M} \sum_{j=1}^{N} Seg_{(i,j)}}$$

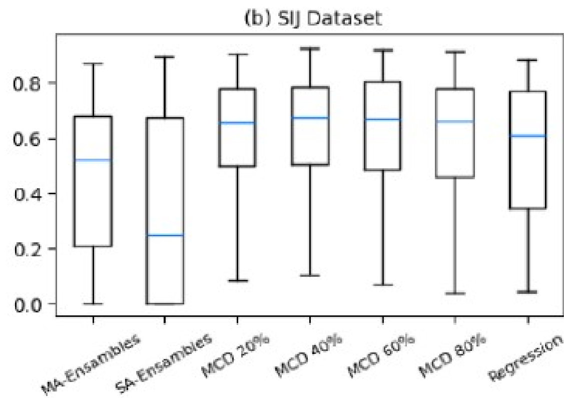
- Normalized Root Mean Square Error
- Normalized Mutual Information

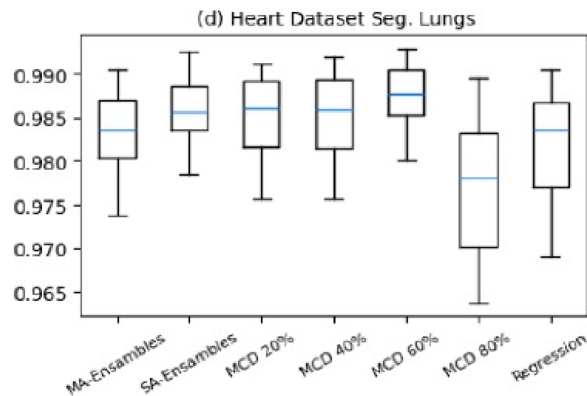
Results - GDC



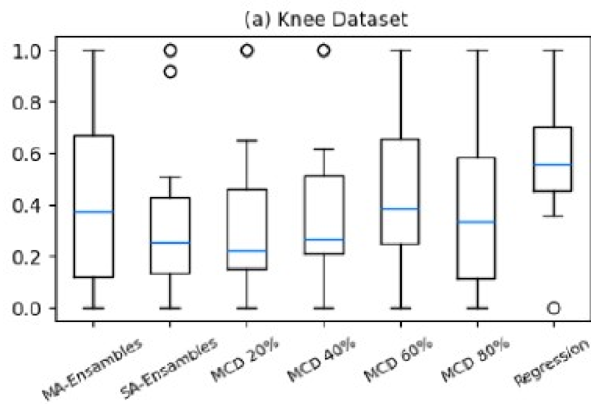


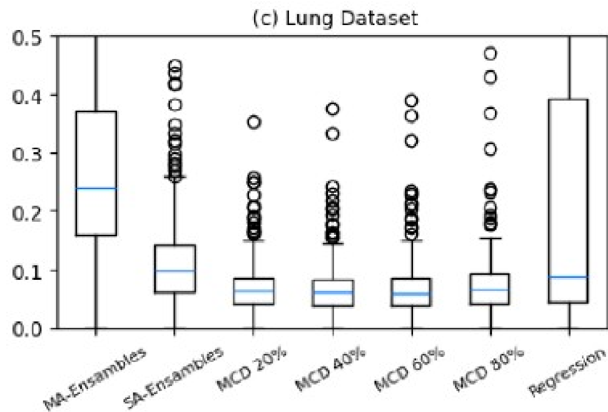


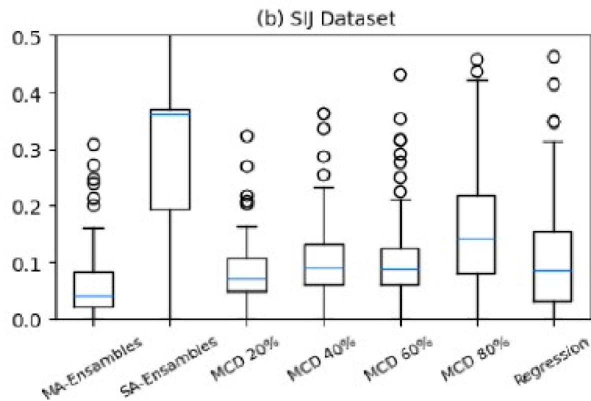


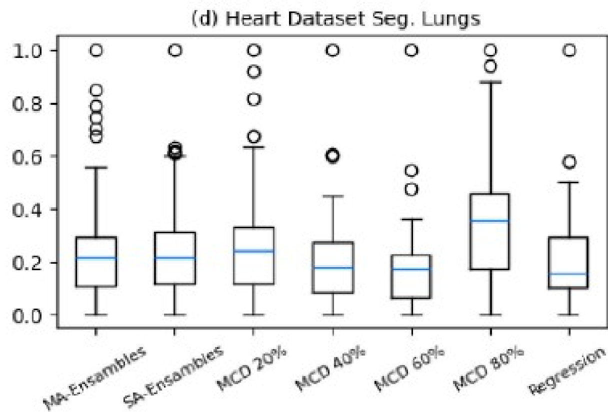


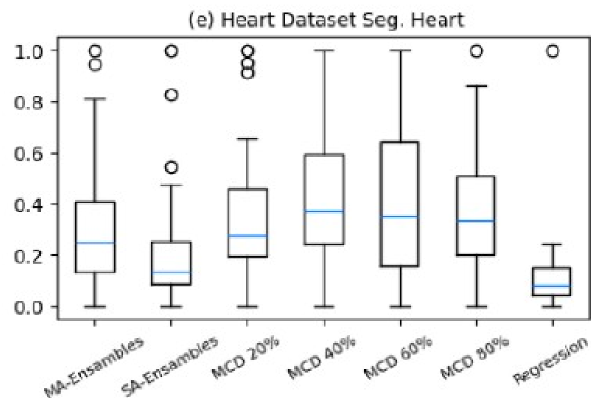
Results - NRMSE



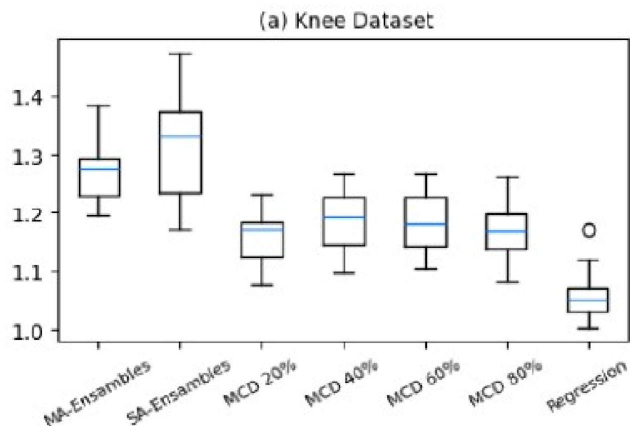


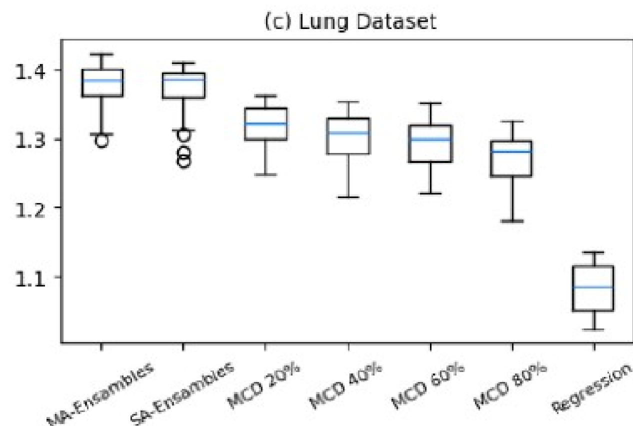


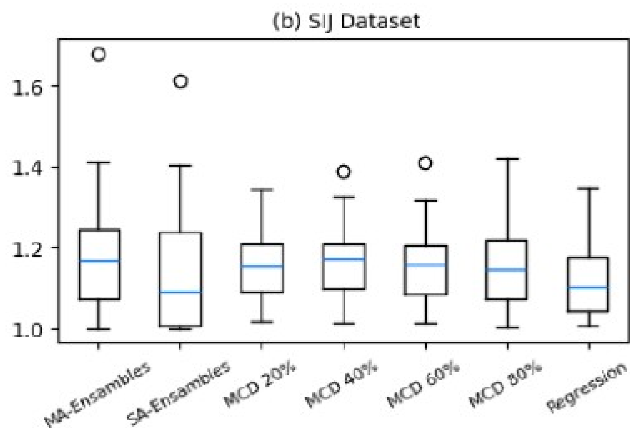


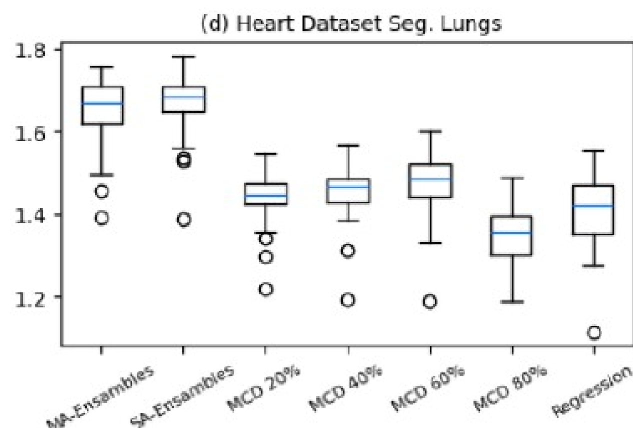


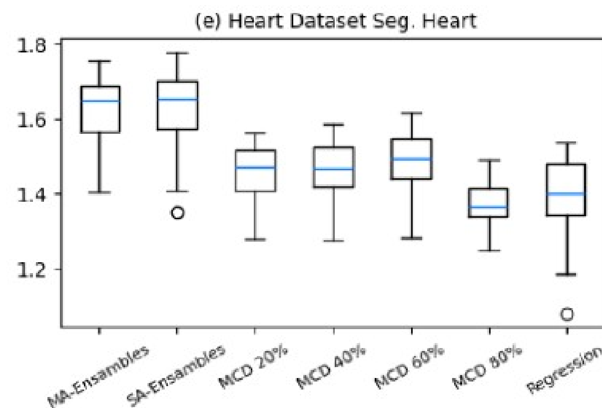
Results - NMI











Future plans

(Please visit our new website: https://qubiq21.grand-challenge.org/)

Quantification of Uncertainties in Biomedical Image Quantification Challenge

(poster/method description and videos presented in the challenge session at MICCAI 2020)

what?

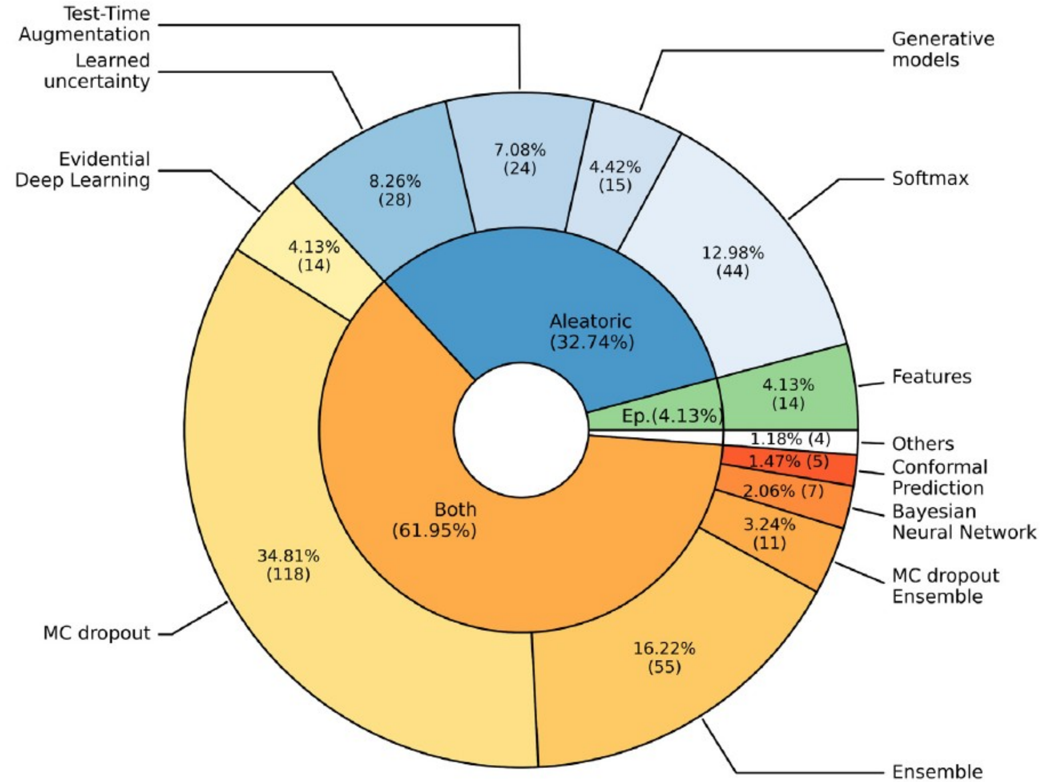
The QUBIQ challenge deals with benchmarking algorithms that quantify uncertainties in biomedical image segmentation. Participants will work on binary segmentation tasks, all of which with multiple annotations from domain experts. To be segmented are various pathologies and anatomical structures, such as brain, kidney, or prostate, in MR or CT image data. A successful algorithm will be able to segment these structures, and to reproduce the distribution of the experts' annotations.

Future plans

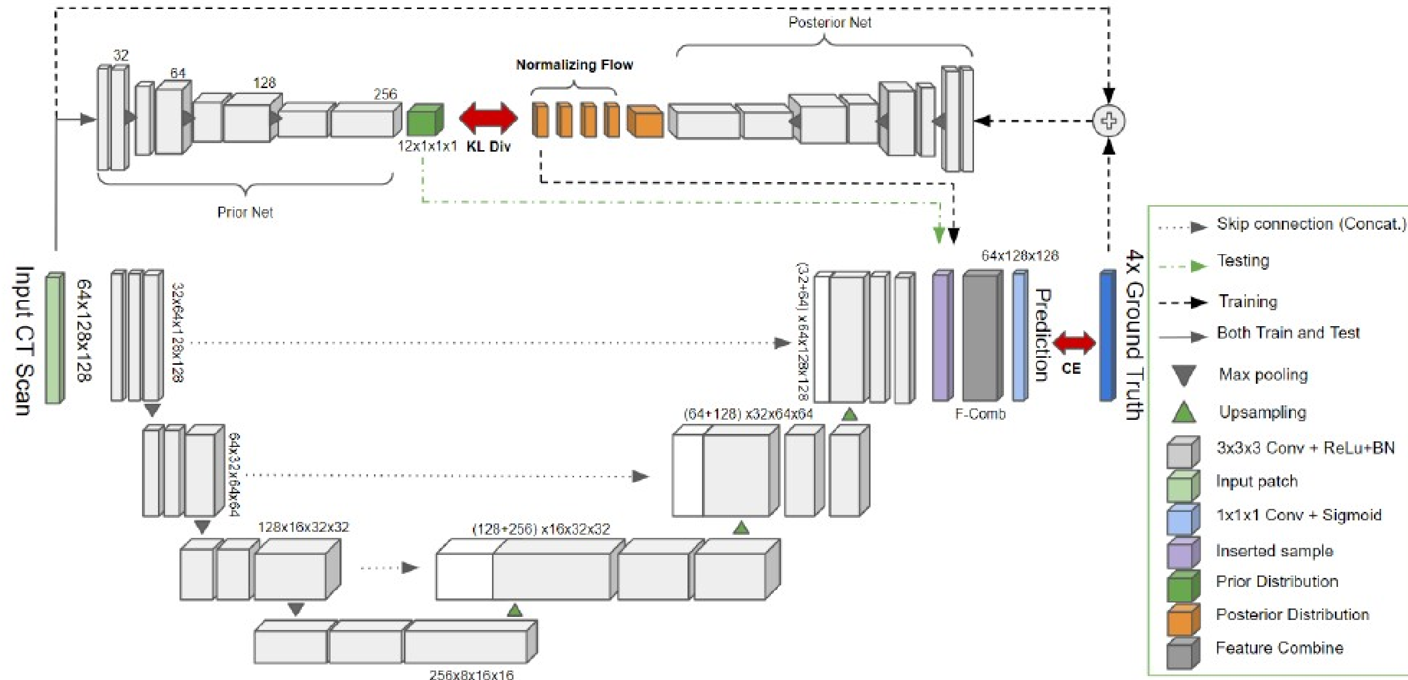
Datasets:

- BRAIN_GROWTH
- BRAIN_TUMOR
- HEART
- KIDNEY
- KNEE
- LUNG
- PANCREAS
- PANCREATIC_LESION
- PROSTATE
- SIJ

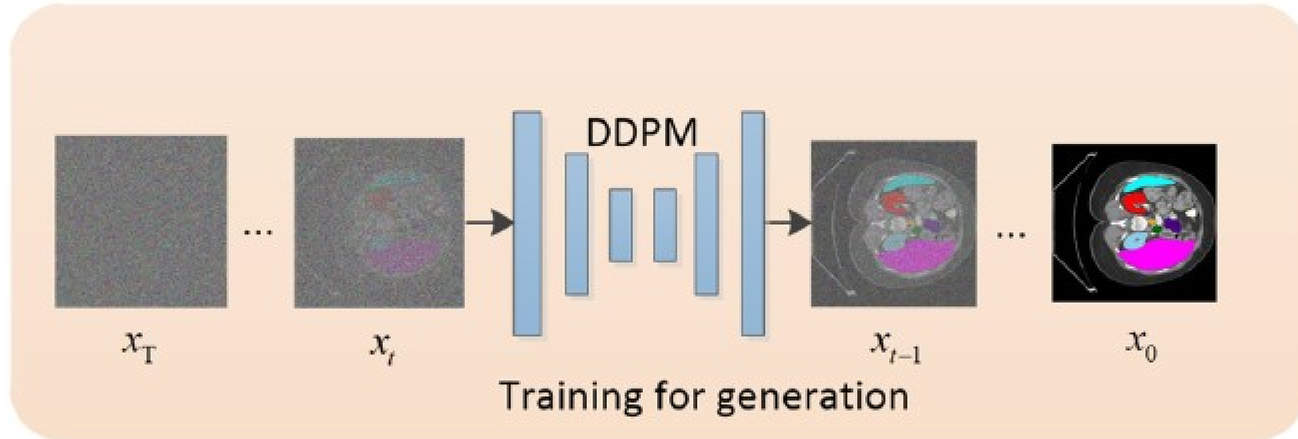
Future plans



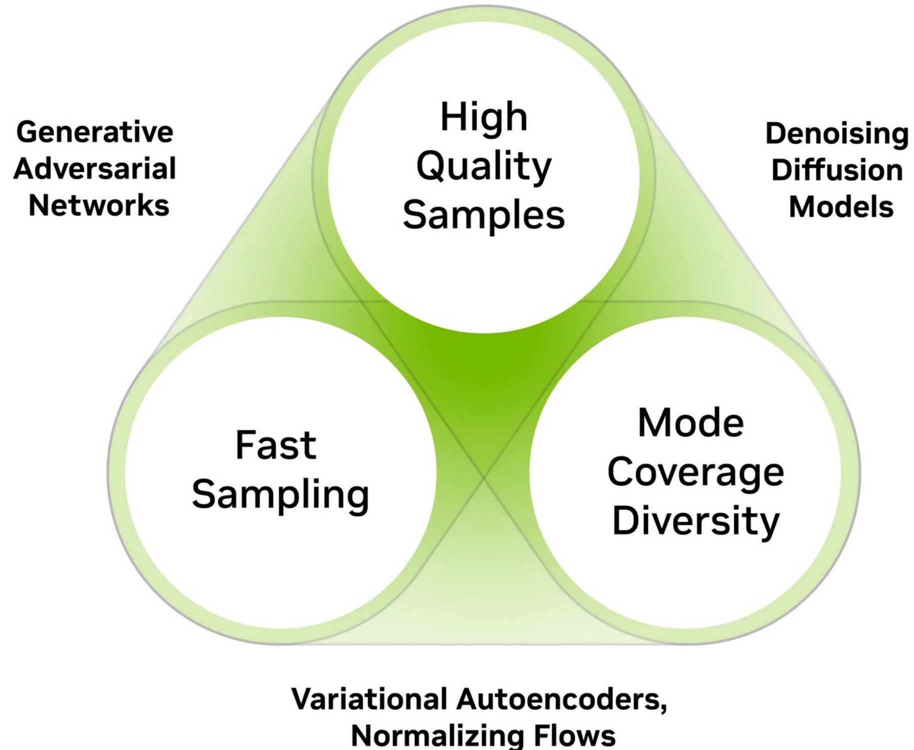
Future plans



Future plans



Future plans



Acknowledgements

- We gratefully acknowledge the funding support by the program “Excellence Initiative—Research University” for the AGH University in Krakow and the ARTIQ project: UMO-2021/01/2/ST6/00004 and ARTIQ/0004/2021. The funding source was not involved in planning, conducting the research, or preparing the article.
- We gratefully acknowledge Polish high-performance computing infrastructure PLGrid (HPC Center: ACK Cyfronet AGH) for providing computer facilities and support within computational grant no. PLG/2023/016441