FYS-3033/8033 Home Exam - Spring 2026

Anonymous Full Paper
Submission 79

1 Problem 1

1.1 Task 1a)

We can interpret equation 1 in the exam sheet as the sum of two expected values:

$$V(G,D) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(g(z))]. \tag{1}$$

Then assuming that we have reached the optimal solution, where the generator-distribution is equal to the data distribution we can write:

$$V(G,D) = \mathbb{E}_{x\sim p_{data(x)}}[\log D(x)] + \mathbb{E}_{x\sim p_g(x)}[\log(1-D(x)] \tag{2}$$

$$= \int_x [p_{data}(x)\log D(x) + p_g(x)\log(1-D(x))]\,dx. \tag{3}$$

We will now maximize the integrand, which we denote as y, with respect to the discriminator. We take the derivative w.r.t. D and set equal to zero:

$$\frac{\partial y}{\partial D} = \frac{p_{data}(x)}{D(x)} - \frac{p_g(x)}{1-D(x)} = 0 \tag{4}$$

$$\implies 0 = (1-D(x))p_{data}(x) - p_g(x)D(x) \tag{5}$$

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}. \tag{6}$$

With our assumption that $p_g = p_{data}$ we get the final result that the optimal discriminator value is $D(x) = \frac{1}{2}$. Inserting this value into 4 gives:

$$V(G,D) = -\log 2\left(\int_x p_{data}(x)dx + \int_x p_g(x)dx\right). \tag{7}$$

Note that both of the integrals have to be equal to one, since we are integrating PDFs over their entire range. Thus, the optimal loss is $V(G,D) = -2\log 2 = -\log 4$. Note that the optima is what we one gets when minimizing the Jensen-Shannon divergence between p_{data} and p_g.

1.2 Task 1b)

In order to train our VAE we need to have a way of describing the latent variable z as a deterministic function (so that we can differentiate and perform backpropagation), instead of a stochastic sampling. This is known as the reparametrization trick and is given by,

$$z = \mu_{z|x} + \sigma_{z|x}\epsilon \tag{8}$$

where $\epsilon \sim \mathcal{N}(0,1)$. So instead of sampling $z \sim q_\phi(z|x)$, we now have a deterministic function which lets the gradients flow through $\mu_{z|x}$ and $\sigma_{z|x}$. Note that we can rewrite 8 as:

$$\epsilon = \frac{z-\mu_{z|x}}{\sigma_{z|x}}. \tag{9}$$

Since $\epsilon \sim \mathcal{N}(0,1)$, this implies that $z \sim \mathcal{N}(\mu_{z|x}, \sigma^2_{z|x})$, because the right hand side of the equation is normalizing z.

1.3 Task 1c)

The Kullback-Leibler divergence is given by: [1]

$$KL(p,q) = \int_x p(x)\log\frac{p(x)}{q(x)}dx. \tag{10}$$

We start by computing $p(x)/q(x)$:

$$\frac{p(x)}{q(x)} = \frac{\sigma_2}{\sigma_1}\exp\left[-\frac{1}{2}\left(\frac{(x-\mu_1)^2}{\sigma_1^2} - \frac{(x-\mu_2)^2}{\sigma_2^2}\right)\right] \tag{11}$$

where we used the the density functions for the normal distributions [2]. Taking the logarithm on both sides, and then applying $\mathbb{E}_{p(x)}[*]$ on both sides gives:

$$KL(p,q) = \log\frac{\sigma_2}{\sigma_1} - \frac{\mathbb{E}[(x-\mu_1)^2]}{2\sigma_1^2} + \frac{\mathbb{E}[(x-\mu_2)^2]}{2\sigma_2^2}. \tag{12}$$

Now note that $\mathbb{E}[(x-\mu_1)^2] = \sigma_1^2$ by definition. We can rewrite,

$$E[(x-\mu_2)^2] = E[(x-\mu_1+\mu_1-\mu_2)^2] \tag{13}$$

$$= E[(x-\mu_1)^2 + (\mu_1-\mu_2)^2] \tag{14}$$

$$= \sigma_1^2 + (\mu_1-\mu_2)^2. \tag{15}$$

Inserting this into 12 gives the final expression:

$$KL(p,q) = \log\frac{\sigma_2}{\sigma_1} - \frac{1}{2} + \frac{\sigma_1^2+(\mu_1-\mu_2)^2}{2\sigma_2^2}. \tag{16}$$

The goal of VAEs is to learn some underlying prior data distribution p(x), so that we can generate new samples from the distribution. To do this we generate samples from a latent representation instead of directly from the other pixel values or tokens:

$$p_\theta(x) = \int p_\theta(z)p_\theta(x|z)dz \tag{17}$$

where $p_\theta(z)$ is a self-chosen prior distribution and $p_\theta(x|z)$ is the "code" that takes us from the latent-space to the "data-space". The integral is intractable, so we have to estimate it instead of computing it directly. We do this by replacing $p_\theta(z|x)$ with some simple distribution $q_\phi(z|x)$ (in Bayes' formula). Note that we now have two distributions that we can choose: $p_\theta(z)$ and $q_\phi(z|x)$. In order to find the optimal q_ϕ, one can optimize the ELBO: [3]

$$\mathcal{L} = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - KL(q_\phi(z|x)||p_\theta(z)) \quad (18)$$

By making both the distributions Gaussian, we can use 16 as a differentiable loss function to push the mean and standard deviation of q_ϕ closer to the $(\mu_z(x), \sigma_z(x))$ (since the first part of the equation is "set").

2 Problem 2

2.1 Task 2a)

The BERT model [4] has a self-attention mechanism, which means it can capture relationships between tokens across the entire input. Unlike for instance RNNs, the BERT model processes all the tokens in the input sequence simultaneously, which lets us parallelize the computation (and take advantage of the provided GPUs). When inputting a sequence into the BERT-model, the sequence will always start with the special [CLS] token. The purpose of this token, is that through every layer of self-attention the [CLS] token will attend to all the other tokens and opposite (bidirectional). Thus, the [CLS] token will essentially contain a compressed summary of the input sequence, which can then be used for classification. This fits our 4-class classification problem perfectly. Although the original paper [4] specifically states that **pre-trained** BERT-models are able to obtain state-of-the-art performance on language inference tasks, our dataset should be simple and large enough to obtain good results without pre-training and thus it is a good starting point.

2.2 Task 2b)

For this task we will use the pre-built *BertForSequenceClassification* architecture [5]. Using the following code, we initialize the model with random weights, instead of the pre-trained ones:

```
from transformers import BertConfig, BertForSequenceClassification, BertTokenizer
import torch
from torch.utils.data import DataLoader, Dataset

#Using BertTokenizer for vocabulary
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
#Initializing random weights for Bert model
config = BertConfig(num_labels = 4)
model = BertForSequenceClassification(config)
```

Note that we also use the pre-built BertTokenizer as our "vocubulary". We tokenize the data using the following function:

```
def tokenize_data(texts, labels, tokenizer):
    data = []
    #Looping over all samples and tokenizing with BertTokenizer
    for txt, label in zip(texts, labels):
        #Max length 128 should be enough for news headlines
        encoded = tokenizer(txt, truncation=True, max_length=128, padding="max_length", return_tensors="pt")
        data.append({"input_ids": encoded["input_ids"].squeeze(),
        "attention_mask": encoded["attention_mask"].squeeze(),
        "labels": torch.tensor(label)})
    return data
```

Note that we use a max length of 128 tokens, which should be sufficient for this particular case. Next we do a 80/20 train-validation-split, before creating batches (size 32) using the PyTorch DataLoader. As our optimizer we use AdamW [6] with a learning rate $\gamma = 10^{-5}$, which is good for BERT models due to the way it applies weight decay separatly from the adaptive updates. We train the model for a total of 5 epochs. The training loop is in the following code,

```
#Training loop
epochs = []
train_loss = []
val_acc = []
val_loss = []
for epoch in tqdm(range(0, n_epochs)):
    model.train()
    curr_loss = 0

    for batch in train_loader:
        #Resetting gradients for batch
        optimizer.zero_grad()
        #Retrieving data from batch
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["labels"].to(device)

        out = model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
        #Computing loss, applying backpropagation and taking a GD-step
        batch_loss = out.loss
        batch_loss.backward()
        optimizer.step()

        curr_loss += batch_loss.item()

    train_loss.append(curr_loss / len(train_loader))
    epochs.append(epoch+1)

    #Testing on validation set
    model.eval()
    curr_val_loss = 0
    correct_preds = 0
    total_samples = 0

    with torch.no_grad():
        for batch in val_loader:
            input_ids = batch["input_ids"].to(device)
            attention_mask = batch["attention_mask"].to(device)
            labels = batch["labels"].to(device)
            out = model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
            batch_loss = out.loss
            curr_val_loss += batch_loss.item()

            #Predictions: argmax of the logits of 4-class output vector => highest probability
            preds = out.logits.argmax(dim=1)
            correct_preds_batch = sum(preds==labels).item()
            correct_preds += correct_preds_batch
            total_samples += batch_size

        accuracy = correct_preds / total_samples
        val_acc.append(accuracy)
        val_loss.append(curr_val_loss / len(val_loader))
```

In figure 1 we see how the training and validation loss/accuracy evolves with the number of epochs. The performance on the validation set is actually the best after only two epochs, but the training loss gets significantly better after more epochs. A validation accuracy of **91.5**% is satisfactory for our initial model without pre-training.

2.3 Task 2c)

In order to use pre-trained weights from Hugginface, the only change we have to make before training is the following line:

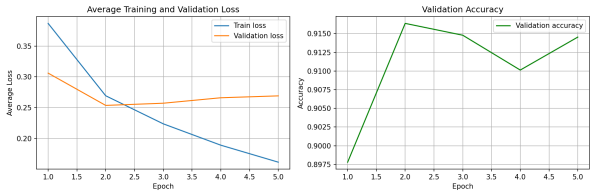

Figure 1. Left: Training and validation loss plotted as a function of epochs. Right: Validation accuracy plotted as a function of epochs.

```
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased",
    num_labels=4
)
```

which means we are **not** overwriting the pre-trained weights with random weights. Since we are now finetuning weights that we know are good for general NLP tasks we expect at least a slight increase in performance. As we can see in figure 2 we do in fact get a significant boost in the validation accuracy, which now peaks at **94.7**%. Once again we note that the model is starting to overfit after 3 epochs, so going forward we will only be using 3 epochs to avoid this and to also save computation-time. The total time for training and inference was 5 hours and 51 minutes. Note that we used the same hyperparameters as for the model we trained from scratch.

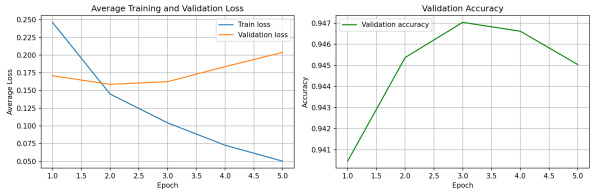

Figure 2. Training and validation loss/accuracy as a function of epochs, with pre-trained weights.

2.4 Task 2d)

To extract the representation from the 4 last transformer layers, we start by enabling hidden state outputs in our model. We then pass the entire validation set through the model and save the representations from these 4 layers. We try two different dimensionality reduction methods: (1) UMAP [7] and (2) Principal Component Analysis (PCA) [8]. The results from the UMAP reduction is shown in figure 3, while the PCA reduction is shown in 4. In the UMAP representation we clearly see that there is already minimal overlap between the classes in the 3rd last layer, which suggests that we can perhaps drop the last two layers. The PCA representation on the other hand shows a clear improvement in separability between each layer, so based only on that it would be hard to justify dropping any of the layers.

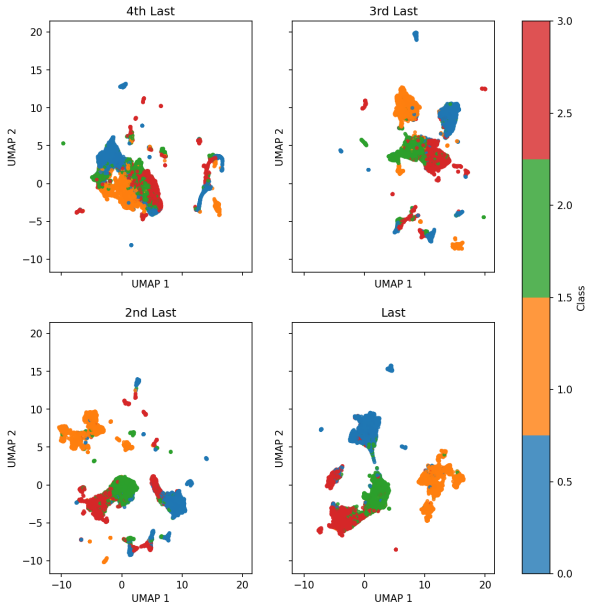

Figure 3. UMAP representation of the 4 last transformer layers.

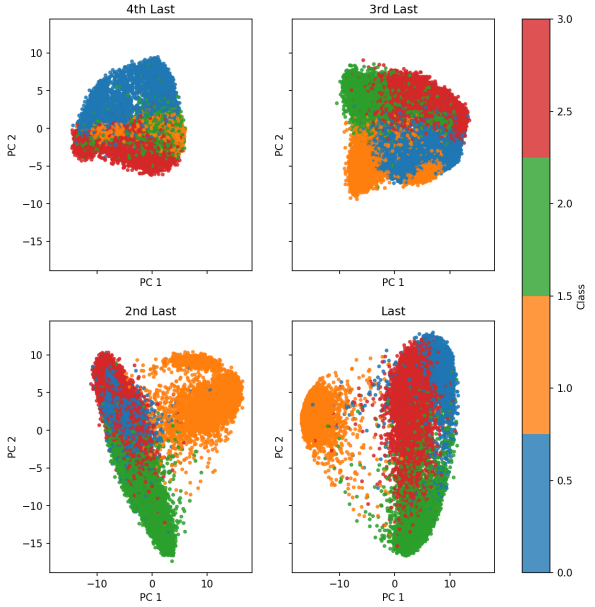

Figure 4. PCA representation of the 4 last transformer layers.

2.5 Task 2e)

Based on the UMAP representations from the previous task, we will try to remove the last two layers of the transformer. According to [9], the top-layer dropping strategy with two layers dropped generally only slightly reduces performance for most of the NLP tasks tested in the paper. We used to following code to setup our slim version of the BERT model:

```
#Setting up a "slim" BERT model with 10 layers
config = BertConfig.from_pretrained("bert-base-uncased")
config.num_hidden_layers = 10
config.num_labels = 4
#10 layer Encoder with pretrained weights
```

```
encoder_slim = BertModel.from_pretrained("bert-base-uncased",
    config=config)
#Wrapping encoder in BertForSequenceClassification model
model = BertForSequenceClassification(config=config)
model.bert = encoder_slim
```

In figure 5 we see the results from the training and inference. We note that the validation accuracy peaked at **94.55**% after two epochs, which is only a drop of 0.15% compared to the full model. The total training and inference time was also significantly reduced to about **34 minutes** per epoch, compared to 70 minutes per epoch for the full model. Note that due to GPU constraints we had to reduce the batch size from 32 to 16 for this particular run, but all the other hyperparameters are the same as for the regular BERT model. Our results show that we can cut the training time in (roughly) half, while barely dropping in performance.

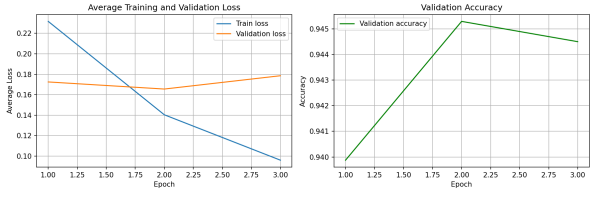

Figure 5. Training and validation loss/accuracy as a function of epochs for "slim" model with last two layers dropped.

2.6 Task 2f)

Monte Carlo Dropout [10] approximates the standard dropout method as Bayesian inference, which can then be used to model uncertainty. During testing one performs T stochastic forward passes for the same input $\mathbf{x}^*$, each with a separately sampled dropout mask. We can then think of the T outputs of the network, $\{\hat{\mathbf{y}}^{(t)}(\mathbf{x}^*)\}_{t=1}^T$, as Monte Carlo samples from the approximate predictive distribution,

$$q(\mathbf{y}^*|\mathbf{x}^*) = \int p(\mathbf{y}^*|\mathbf{x}^*, \omega) q(\omega) d\omega. \tag{19}$$

$q(\omega)$ is the dropout-induced approximate posterior of the weights. According to [10] we can approximate the variance (uncertainty) using the formula:

$$Var(\mathbf{y}^*) \approx \tau^{-1}\mathbf{I}_D + \frac{1}{T}\sum_t \hat{\mathbf{y}}^{(t)}\hat{\mathbf{y}}^{(t)^T} - E[\mathbf{y}^*]E[\mathbf{y}^*]^T, \tag{20}$$

where τ is the precision hyper-parameter determined by,

$$\tau = \frac{pl^2}{2N\lambda}. \tag{21}$$

λ is weight decay, p is dropout-probability, N is dataset size and l is the prior length-scale. We can also use the following approximation to compute $E[\mathbf{y}^*]$,

$$E[\mathbf{y}^*] \approx \frac{1}{T}\sum_{i=1}^T \hat{\mathbf{y}}^*(\mathbf{x}^*, \mathbf{W}_1^t, ..., \mathbf{W}_L^t). \tag{22}$$

Note that the expected values and variance in the previous equations are with respect to $q(\mathbf{y}^*|\mathbf{x}^*)$. In our BERT model, dropout is already being used. Thus, by keeping dropout active during testing and doing T forward passes per sample, we can obtain Monte Carlo samples of the predictive distribution and determine the uncertainty of all the validation predictions, by for example using entropy as our measure.

In practice, a system could for example send the top X% highest entropy predictions to human review. Depending on what one chooses X to be, this would significantly increase the overall accuracy of the system, while still saving a lot of time for employees. This would reduce the risk of completely wrong predictions on ambiguous samples.

2.7 Task 2g)

We chose to use Monte Carlo Dropout for uncertainty modeling, because it is easy to implement due to the fact that Dropout is already being used during training. This lets us re-enable dropout during inference to get the uncertainty estimates. Our approach is doing $T = 20$ forward passes with different dropout masks and computing the probability of each class. We then compute the predictive entropy,

$$H[y \,|\, \mathbf{x}, \mathcal{D}] = -\sum_c p_c \log p_c \tag{23}$$

where p_c is the probability of class c which we get from the estimated probability density:

$$p(y \,|\, \mathbf{x}, \mathcal{D}) \approx \frac{1}{T}\sum_{t=1}^T p(y \,|\, \mathbf{x}, \omega_t) \tag{24}$$

where ω_t is the sampled set of weights (dropout mask). A higher entropy value implies that the model is more uncertain about its classification of the sample. Our implementation is provided in the code below:

```
#Helper function to enable dropout
def enable_dropout(model):
    if isinstance(model, torch.nn.Dropout):
        model.train()

def mc_dropout_unc(model, dataloader, device, T=30):
    model.to(device)
    model.eval()
    #Enabling dropout during inference
    model.apply(enable_dropout)

    sample_entropy = []

    with torch.no_grad():
        for batch in tqdm(dataloader):
            input_ids = batch["input_ids"].to(device)
            attention_mask = batch["attention_mask"].to(device)
            preds_T = []

            #Doing T forward passes through network
            for i in range(0,T):
                output = model(input_ids, attention_mask)
                pred = F.softmax(output.logits, dim=-1)
                preds_T.append(pred.unsqueeze(0))

            #Converting to tensor
            preds_T = torch.cat(preds_T, dim=0)
            mean_preds = preds_T.mean(dim=0)
            #Computing per sample predictive entropy in batch
            entropy = -(mean_preds*(mean_preds+1e-12).log()).sum(dim=-1)
            sample_entropy.append(entropy.cpu())
    #Removing batch dimension
    sample_entropy = torch.cat(entropies, dim=0)
    return sample_entropy
```

We also need to select some criteria for which samples are "uncertain enough" to be flagged for human analysis. We can do this by setting some cut-off value for the entropy or flagging a percentile of the samples with the highest entropy. Below we show the 4 validation samples with the highest uncertainty, along with their entropy, true label and predicted label:

```
olice department suspends use of pepper pellet gun At least one big-city police department has suspended use of pepper-spray pellet guns blamed for the death of a 21-year-old college student
Entropy: 1.3659993410110474
True Label: 1
Predicted Label: 1
-----
Does a NY Supreme Court judge say that the Central Park lawn is more important than free speech? Here is the coverage in The New York Times and the New York Law Journal about the denial of United for Peace and Justice's Sunday rally in Central Park (United for Peace and Justice v. Bloomberg, 111893/04), and here is what NY Supreme Court1 Justice Silbermann said:      ... the evidence established that the department's determination was based on entirely content-neutral factors, to wit: that the Great Lawn2 was not an appropriate venue for a demonstration of this magnitude.     ...  The Parks Department appropriately applied content-neutral regulations while leaving plaintiff with a reasonable alternate site.
Entropy: 1.3181042671203613
True Label: 3
Predicted Label: 2
-----
Teen accused of making threats about school, possessing weapons A 17-year-old charged with making terrorist threats against his high school, fellow students and a police officer said at his arraignment that he would do anything  quot;to help the community quot; in the wake of his arrest.
Entropy: 1.1795392036437988
True Label: 3
Predicted Label: 3
-----
Why do Workouts Work? Most machines don't improve with use. Old pickup trucks don't gradually become Ferraris just by driving them fast, and a pocket calculator won't change into a supercomputer by crunching lots of numbers. The human body is different...
Entropy: 1.1735734939575195
True Label: 3
Predicted Label: 1
```

The first sample doesn't really fit into any of the classes, so it makes sense that the model is uncertain (although it gets it right). For the second sample, the uncertainty likely comes from the fact that the text is poorly structured with random numbers and symbols placed all around the place. The same applies to the third sample. The fourth sample is likely predicted as "Sports" due to the word "Workout" being used, and the model is uncertain due to the title being a bit non-sensical.

2.8 Task 2h)

For the Kaggle competition we try a couple of different options for our model. The resulting accuracies of all these approaches are shown in table 1, where the first entry is simply the pre-trained model from task 2c which gave an accuracy of 94.03%.

Next we tried using a RoBERTa (Robustly optimized BERT pretraining Approach) model [11], which is a slightly larger model than the regular BERT model and it is also pre-trained for longer and in a *smarter* way. RoBERTa models can match or exceed the performance of regular BERT models [11]. There are 4 key differences in the pre-training process of RoBERTa models: (1) dynamic masking instead of static masking, (2) larger mini-batches, (3) larger byte-pair encoding (larger vocabularies) and (4) no Next Sentence Prediction (NSP) loss during training. We used a batch size of 64, learning rate = 1e-5, weight decay = 0.01, 5 epochs and AdamW as the optimizer.

Lastly, we tried XLNet-large [12] which is a significantly larger pre-built model than BERT and RoBERTa. We used the same hyperparameters as for RoBERTa, but only 3 epochs due to the long training time (∼2 hours per epoch). The key difference between XLNet and the previous models we used is the pre-training method. XLNet uses a generalized autoregressive method that maximizes the expected likelihood over all possible orders that the masked tokens are predicted in the autoregressive decomposition [12].

As we can see in table 1, RoBERTa clearly performed the best both on the test and validation set. The models were imported using the following code:

Model	Val. Acc.	Test Acc.
PT-BERT (Task 2c)	94.70%	94.04%
RoBERTa	94.95%	94.96%
XLNet-Large	94.75%	94.64%

Table 1. Validation and test accuracy of the used models.

```
from transformers import XLNetTokenizer, XLNetForSequenceClassification
tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
model = XLNetForSequenceClassification.from_pretrained('xlnet-large-cased', num_labels=4)
```

```
from transformers import RobertaTokenizer, RobertaForSequenceClassification
tokenizer = RobertaTokenizer.from_pretrained("roberta-base")
model = RobertaForSequenceClassification.from_pretrained("roberta-base", num_labels=4)
```

The training loops and data pre-processing were identical to the ones shown in section 2.2.

3 Problem 3

3.1 Task 3a)

ResNet50 [13] is a so called *Residual Network* that implements the idea of *skip connections*. The benefit of skip connections is that optimization of deep networks becomes more stable, since one avoids vanishing/exploding gradients. This is especially important when we are training the network from scratch (instead of finetuning).

Our task is to classify CT images (512x512 pixels) of a liver into 3 classes (no liver, liver present, tumor present), which requires the model to capture relatively high-level context from the image, like organ boundaries and tumor regions. ResNet50 is a convolutional neural network (CNN), and is therefore well suited when it comes to extracting local textures, patterns and shapes. A medium-sized model like ResNet50 should be capable enough to capture patterns of such complexity. ResNet50 is also widely

used and is a well-tested architecture, which makes it a natural starting point for our task.

3.2 Task 3b)

We start by setting up the pre-built ResNet50 model with randomly initialized weights:

```python
#Randomly initialized ResNet50
model = resnet50(weights=None)
#Changing number of output classes to 3
n_classes = 3
model.fc = nn.Linear(model.fc.in_features, n_classes)
```

The only change we have to make to the architecture is the number of output classes in the final linear classifier. Note that ResNet50 expects an input with dimensions $(N, C = 3, H, W)$ where C is the number of channels, thus we have to add this extra dimension to our dataset. Another thing to note is that the validation data has pixelwise labels, which we have to convert to a single label per image. We do this with the following function:

```python
def pixelwise_to_class(img_dir, label_dir):
    X = []
    y = []
    img_files = sorted(img_dir.glob("*.npy"))
    label_files = sorted(label_dir.glob("*.npy"))

    for img_path, label_path in zip(img_files, label_files):
        img = np.load(img_path)
        pixelwise = np.load(label_path)
        X.append(img)
        #If any pixel labelled 2 => class 2
        if np.any(pixelwise == 2):
            y.append(2)
        #If no pixel has 2, but 1 exists => class 1
        elif np.any(pixelwise == 1):
            y.append(1)
        else:
            y.append(0)
    #Converting X and y to tensors
    X = np.stack(X)
    y = np.array(y)
    X = torch.from_numpy(X).float()
    y = torch.from_numpy(y).long()
    X = X.unsqueeze(1).repeat(1,3,1,1)
    return X, y
```

For the training we use the following hyperparameters: **batch_size=32**, **learning rate=1e-4** and **100 epochs**. As our optimizer we use AdamW [6], because the decoupled weight decay makes it ideal for training deep networks (especially from scratch). We use cross-entropy loss [14] as our loss function/criterion, which is a standard choice for single-label classification problems. The training loop is given in the following code:

```python
for epoch in tqdm(range(n_epochs)):
    model.train()
    curr_loss = 0
    samples = 0

    #Training
    for batch_x, batch_y in train_loader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)
        optimizer.zero_grad()
        #forward pass
        logits = model(batch_x)
        #Using Cross-entropy loss as loss function
        loss = loss_func(logits, batch_y)
        #Backpass + backprop
        loss.backward()
        optimizer.step()
        #Multiplying by size of current batch and dividing by total set
        ↪  size later
        curr_loss += loss.item() * batch_x.size(0)
        samples += batch_x.size(0)
    training_loss.append(curr_loss/samples)

    #Validation
    model.eval()
    total_samples = 0
    total_correct = 0
    correct = np.array([0,0,0])
    n_class = np.array([0,0,0])


    with torch.no_grad():
        for batch_x, batch_y in val_loader:
            batch_x = batch_x.to(device)
            batch_y = batch_y.to(device)
            logits = model(batch_x)
            preds = torch.argmax(logits, dim=1)
            #Counting number of samples and correcting preds
            total_samples += batch_x.size(0)
            total_correct += (preds == batch_y).sum().item()
            #Counting classwise correct preds
            for pred, label in zip(preds, batch_y):
                if label == 0:
                    n_class[0] += 1
                    if pred == 0:
                        correct[0] += 1
                elif label == 1:
                    n_class[1] += 1
                    if pred == 1:
                        correct[1] += 1
                elif label == 2:
                    n_class[2] += 1
                    if pred == 2:
                        correct[2] += 1
    val_acc0.append(correct[0]/n_class[0])
    val_acc1.append(correct[1]/n_class[1])
    val_acc2.append(correct[2]/n_class[2])
    val_acc_tot.append(total_correct/total_samples)
    current_val_acc = val_acc_tot[-1]
    if current_val_acc > best_val_acc:
        best_val_acc = current_val_acc
        best_epoch = epoch + 1
        best_checkpoint = {
        "epoch": n_epochs,
        "model_state_dict": model.state_dict(),
        "train_losses": training_loss,
        "val_acc_tot": val_acc_tot,
        "val_acc0": val_acc0,
        "val_acc1": val_acc1,
        "val_acc2": val_acc2}
        torch.save(best_checkpoint, "best_problem3_checkpoint.pth")
```

Since 100 epochs is potentially too many, we use *checkpointing* in order to save the model state with the highest validation accuracy. In fact, as we can see in figure 6 the total validation accuracy peaks at epoch 46 with **78.85**%. The class-wise accuracies for this model state were 96% (class 0), 72% (class 1) and 43% (class 2). This big spread in performance across classes likely comes from the fact that there is a class imbalance in the training (and validation) set. 50% of the samples are class 0, while 33.8% and 16.2% are class 1 and 2 respectively. Thus, the model will do reasonably well if it just guesses that most samples are class 0.

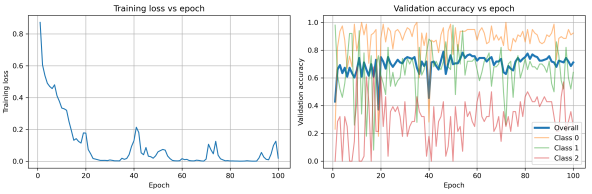

Figure 6. Training results from scratch. Left: Training loss as a function of epochs. Right: Total validation accuracy (blue), class 0 (yellow), class 1 (green), class 2 (red).

3.3 Task 3c)

We load the pre-trained DINO ResNet50 [15], which outputs a 2048 dimensional feature vector. Thus, we need to add an additional linear classification layer with (in_dim, out_dim) = $(2048, 3)$:

```python
dino = torch.hub.load('facebookresearch/dino:main', 'dino_resnet50',
↪  pretrained=True)
model = nn.Sequential(dino, nn.Linear(2048, 3))
```

Initially we run the training loop with AdamW as our optimizer. This gave a peak validation accuracy of 78.18%, which is worse than the model we trained from scratch. In the documentation [15] they use

SGD [16] (with weight decay) as the optimizer, so we will try that instead. We also copy their hyperparameters: learning rate = 0.03, weight decay = 1e-4 and 300 epochs. The results of the training are shown in figure 7. We can see that the class 0 and total validation accuracy pretty much converge, while the class 1 and 2 accuracies never quite stabilize during the training. This is still an improvement from the model we trained from scratch, although the comparison is not really fair since we have now done more epochs and used a different optimizer. The total validation accuracy peak was **85.26**% (epoch 243), which is a significant improvement. The class specific accuracies were 96% (+0%), 86% (+14%) and 54% (+11%) respectively.

Ideally we would run the first model with the same optimizer and hyperparameters in order to get a more fair comparison.

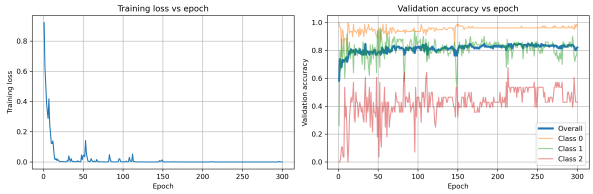

Figure 7. Training results for pre-trained ResNet50. Left: Training loss as a function of epochs. Right: Total validation accuracy (blue), class 0 (yellow), class 1 (green), class 2 (red).

3.4 Task 3d)

The purpose of XAI is to reinforce our trust in a given AI model. Most state-of-the-art networks have many layers and millions/billions of parameters and are essentially "black boxes", meaning we as humans cannot comprehend or understand the reasoning of a model. This is where explainable AI comes in, which is a field of research that aims to create ways for an AI model to *explain* **why** it does something. This can be done by for example highlighting regions of interest in an image or generating a text-explanation. In our case, we are interested in knowing why it classifies a certain CT image as having a tumor and want it to highlight the ROI. In general, interpretability is a very important factor in medical AI settings, since mistakes done by the model can be fatal.

Gradient-weighted Class Activation Mapping (Grad-CAM) [17] is an XAI technique that uses the gradients with respect to any target concept, for instance "tumor", to highlight influential regions for the classification in a given image (can also be used for text, etc.). The original paper [17] tests Grad-CAM on ResNet-based architectures with good success, making it a natural choice for us.

For a chosen class c we take the gradient of the class score, y^c (before softmax is applied), with respect to the feature map activation of the final convolutional layer A^k. The gradients are then global average pooled:

$$\alpha_k^c = \frac{1}{Z}\sum_i\sum_j \frac{\partial y^c}{\partial A_{ij}^k} \tag{25}$$

where Z is a normalization constant and (i, j) denotes a pixel. The heatmap is then constructed by taking a linear combination of these "weights" and the feature maps:

$$L^c_{Grad-CAM} = \sum_k \alpha_k^c A^k. \tag{26}$$

This matrix will highlight both positive and negative influences of the given classification, but we are interested in only the positive ones. Thus, the last step is passing $L^c_{Grad-CAM}$ through a ReLU function.

3.5 Task 3e)

We implement GradCAM using the PyTorch GradCAM library:

```
backbone = model[0] #extracting ResNet50 part of our model
layer = backbone.layer4[-1]
grad_cam = GradCAM(model=model, target_layers = [layer])
```

We are using the gradients from the final convolutional layer to create the heatmap (eq. 26), which is layer 4 in this case. These gradients will be used in equation 25 to determine the contribution (weight) of a particular region in the image for the given classification. We apply GradCAM to 5 validation images that were labeled as tumors, using the following code:

```
for i in range(0,5):
    idx = val_idxs[i]
    img_path = val_file_names[idx]
    img = np.load(img_path)
    lbl = val_y[idx]
    #Selecting target class for GradCAM as the predicted class
    target = val_preds[idx]
    targets = [ClassifierOutputTarget(target)]
    #Applying grayscale GradCAM to tensor representation of image
    X = torch.tensor(img,
    ↪  dtype=torch.float32).unsqueeze(0).unsqueeze(0).to(device)
    X = X.repeat(1,3,1,1)
    gs_cam = grad_cam(input_tensor = X, targets = targets)
    # ... plotting
```

The resulting images are shown in figure 8. The first two examples were wrong predictions, with "liver" being the true label, while the last three examples are correctly predicted as "tumor". To an untrained eye, the "blobs" that the model activates on seem rather similar in all of the examples. Without medical expertise it is hard to tell what exactly the model is doing wrong in these examples.

3.6 Task 3f)

In order to turn our GradCAM output into a mask for pixelwise classification, we will start by normalizing the output. We will then choose a threshold in $[0, 1]$ that will determine if we classify a pixel as tumor/liver or background. In order to choose this

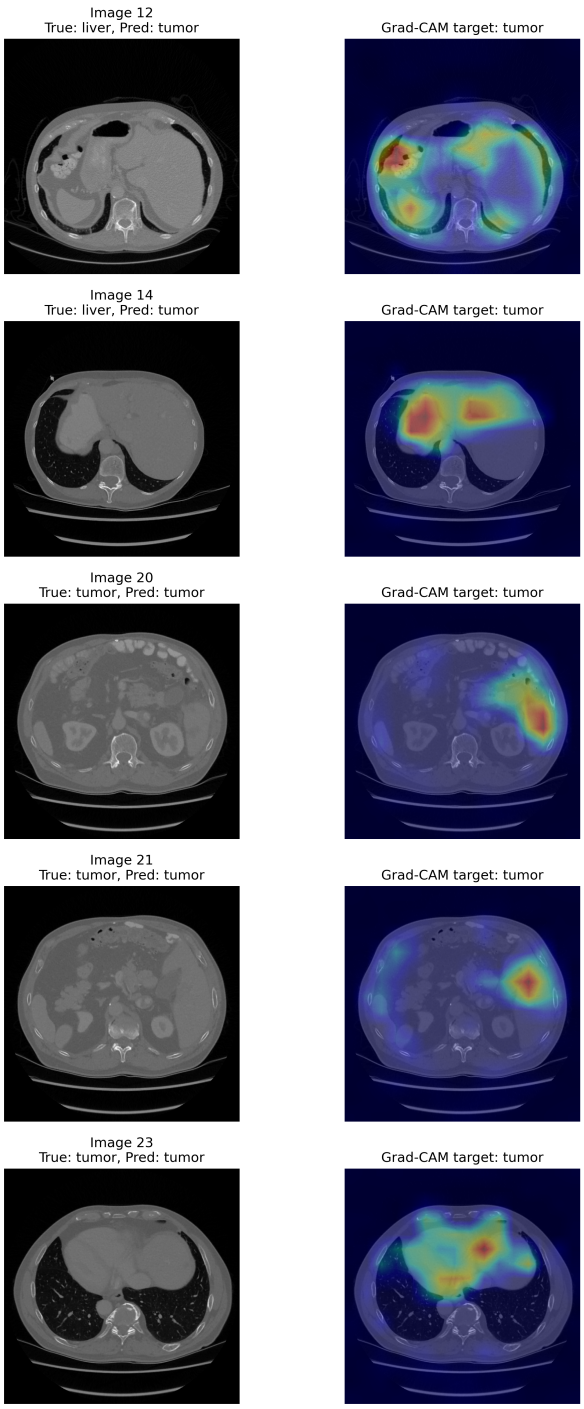

Figure 8. GradCAM applied to 5 validation images that were classified as tumors by the DINO-ResNet50 model. Left: Original image. Right: GradCAM applied.

threshold, we will try 8 different ones for each class, and choose the ones that give the best IoU score. The implementation of the GradCAM pixelmask and IoU-score computation:

```python
def gradcam_pixelmask(gradcam, threshold):
    #Normalizing the gradcam heatmap
    gradcam = (gradcam - gradcam.min())/(gradcam.max() - gradcam.min() + 1e-9)
    #gradcam value above threshold -> mask=1, else 0
    mask = np.where(gradcam >= threshold, 1, 0)
    #Smoothing the mask
    #removing noise outside main object, 3x3 kernel
    mask = binary_opening(mask, structure=np.ones((5,5)))
    #Filling small holes inside object, 3x3 kernel
    mask = binary_closing(mask, structure=np.ones((5,5)))
    return mask

def iou_score(pred, true):
    #Turning masks into boolean arrays so we can use np.logical funcs
    pred_mask = pred.astype(bool)
    true_mask = true.astype(bool)
    #computing intersection and union
    inter = np.logical_and(pred_mask, true_mask).sum()
    union = np.logical_or(pred_mask, true_mask).sum()
    iou = inter / (union + 1e-9)
    return iou
```

The thresholds were determined using the following code:

```python
thresholds = np.linspace(0.5, 0.99, 20)
best_iou_liver = 0
best_iou_tumor = 0
best_threshold_tumor = 0
best_threshold_liver = 0
for threshold in tqdm(thresholds):
    iou_tumor = []
    #Looping over validation images with tumor
    for i in range(len(val_file_names)):
        if val_y[i] == 2:
            #Loading image and pixelwise labelling
            img = np.load(val_file_names[i])
            true_labels = np.load(val_label_file_names[i])
            #Converting image to tensor with right dimensions
            X = torch.tensor(img,
            ↪  dtype=torch.float32).unsqueeze(0).unsqueeze(0).to(device).repeat(1,3,1,1)
            #Defining tumor as target
            target_tumor = ClassifierOutputTarget(2)
            #GradCAM output for tumor class for sample
            gradcam_tumor = grad_cam(input_tensor=X, targets =
            ↪  [target_tumor])[0]
            #Computing true and prediction mask
            pred_mask_tumor = gradcam_pixelmask(gradcam_tumor, threshold)
            true_mask_tumor = np.where(true_labels == 2, 1, 0)
            #Computing IoU
            iou_tumor.append(iou_score(pred_mask_tumor, true_mask_tumor))
    iou_tumor = np.array(iou_tumor)
    mean_iou_tumor = np.mean(iou_tumor)
    if mean_iou_tumor >= best_iou_tumor:
        best_iou_tumor = mean_iou_tumor
        best_threshold_tumor = threshold
#...Same for liver class
```

As we can see in table 2, the results are not very good with an IoU score of 13.96% and 11.69% for the tumor and liver classes respectively. The reason for this is likely that the representations of the image have been down-sampled too much in the final layer in order to get accurate segmentations. Tumors in CT images are generally small and fuzzy, while Grad-CAM produces low-resolution blobs that are meant to be used as a visual aid. As discussed in [18], GradCAM is meant for visualization and not for image segmentation tasks. For image segmentation there are specialized models that will perform way better, like for example U-Net (specialized for medical images) and SAM [19].

Across all the validation samples the highest IoU achieved was 60.29%, while the worst was 0%. Here we used a threshold of 0.76, which gave the highest mean IoU. These examples are shown in figure 9. In the good example we see that the true position of the tumor aligns very well with the most dense region on the GradCAM heatmap. What is dragging the IoU score down for this sample are the random weak activations in other regions of the image. The example of bad overlap is a bit brutal, because the tumor is extremely small and the highlighted region is just below the true position.

Our conclusion is that GradCAM should only be used as a decision-support and visualization tool in clinical practice, and NOT be used as an automatic replacement of a radiologist.

Method (Class)	Mean IoU Score
GradCAM (Tumor)	0.1396
GradCAM (Liver)	0.1169

Table 2. Mean IoU scores.

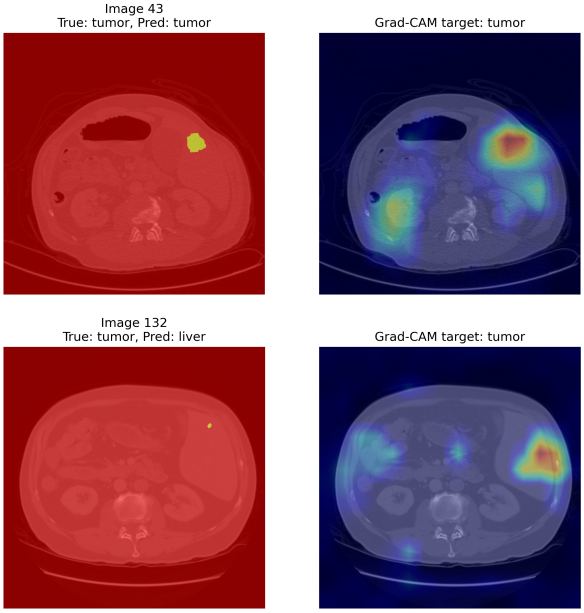

Figure 9. Left: True label, tumor highlighted in yellow. Right: Grad-CAM heatmap of same image. Top example shows the best overlap, while the bottom example shows 0 overlap.

3.7 Task 3g)

Initially we pass the test data through our model from task 3c. This resulted in an accuracy of 83.116%.

In order to improve the performance, we will use a pre-built model from Hugginface called ConvNeXT [20] which is based on the ConvNet architecture [21]. In [21] (table 10) they show the steps they add to their baseline model, which is ResNet50, in order to end up at the ConvNeXT architecture. They also show how much each of these extra steps improve accuracy on the ImageNET-1k dataset. Some examples of these improvements are: Using GELU [22] instead of ReLU, using fewer activations and norms, depthwise convolutions and a larger kernel size on the filters. Thus, we can assume that using ConvNeXT will be an improvement for our image classification task.

Due to the class imbalance in the training set we will also implement class-weighting into our Cross-Entropy loss function. To try to prevent overfitting and model overconfidence we will also add label smoothing (=0.1). We use the following hyperparameters: learning rate = 1e-4, 50 epochs (with checkpointing), weight decay = 0.01 and batch size 8. We used AdamW [6] as the optimizer because it was used in [21]. The training results are shown in figure 10. The peak validation accuracy was 88.46% (epoch 3). We imported the model using the following code, and changed the final linear classification layer to fit our problem:

```
weights = ConvNeXt_Base_Weights.DEFAULT
model = convnext_base(weights=weights)
#Tuning the linear classifier so it has correct dimensions
in_features = model.classifier[2].in_features
model.classifier[2] = nn.Linear(in_features, 3)
```

On the Kaggle test data ConvNeXT performed significantly better than our previous model, achieving an accuracy of **90.909**%. The classwise accuracies are shown in table 3.

The key step in our implementation of ConvNeXT is the data pre-processing. Everything else is pretty much the same as for the other model. ConvNeXT is pre-trained on ImageNET, so we have to pre-process the data in the same way as ImageNET. Here is how we do that on the training data:

```
#ConvNeXt takes in input with dims (N, C, H, W). Adding C dimension (3 channels):
train_x = train_x.unsqueeze(1).repeat(1,3,1,1)
#Normalizing training set
train_x = train_x / 255
#ImageNET mean and std RGB values (using same normalization as ImageNET)
mean = torch.tensor([0.485, 0.456, 0.406]).view(1, 3, 1, 1)
std = torch.tensor([0.229, 0.224, 0.225]).view(1, 3, 1, 1)
train_x.sub_(mean)
train_x.div_(std)
```

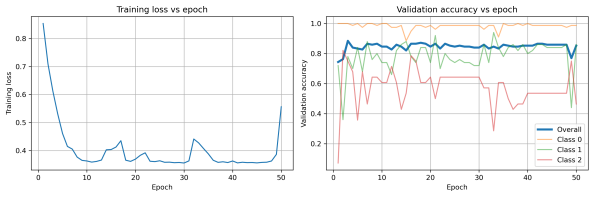

Figure 10. Training curves for ConvNeXT model.

Class	Validation Accuracy (%)
Total	88.46%
Background (0)	100%
Liver (1)	78%
Tumor (2)	75%

Table 3. Validation accuracy of ConvNeXT model.

4 AI Declaration

For this submission Perplexity AI [23] was used as aid. The AI was used to generate code for plotting, like for example figures 3 and 4 (only the tedious matplotlib blocks of code). The AI was also used to generate BibTeX entries for various sources and converting handwritten formulas to LateX code.

References

[1] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951). DOI: 10.1214/aoms/1177729694.

[2] Wikipedia contributors. *Normal distribution*. https://en.wikipedia.org/wiki/Normal_distribution. Accessed: 2026-04-11. 2025.

[3] Wikipedia contributors. *Evidence lower bound*. https://en.wikipedia.org/wiki/Evidence_lower_bound. Accessed: 2026-04-11. 2026.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by J. Burstein, C. Doran, and T. Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423/.

[5] Hugging Face. *BERT — Hugging Face Transformers Documentation*. Accessed: 2026-04-14. 2024. URL: https://huggingface.co/docs/transformers/model_doc/bert.

[6] Y. Hao, Z. Tang, Y. Tian, Y. Zhang, and Z. Zhou. *AdamW*. Cornell University Computational Optimization Open Textbook - Optimization Wiki. ChemE 6800 Fall 2024. 2024. URL: https://optimization.cbe.cornell.edu/index.php?title=AdamW (visited on 04/14/2026).

[7] L. McInnes, J. Healy, and J. Melville. "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction". In: *arXiv preprint arXiv:1802.03426* (2018). DOI: 10.48550/arXiv.1802.03426. arXiv: 1802.03426 [stat.ML].

[8] A. Maćkiewicz and W. Ratajczak. "Principal components analysis (PCA)". In: *Computers & Geosciences* 19.3 (1993), pp. 303–342. ISSN: 0098-3004. DOI: 10.1016/0098-3004(93)90090-R. URL: https://www.sciencedirect.com/science/article/pii/009830049390090R.

[9] H. Sajjad, F. Dalvi, N. Durrani, and P. Nakov. "On the effect of dropping layers of pre-trained transformer models". In: *Computer Speech & Language* 77 (2023), p. 101429. ISSN: 0885-2308. DOI: 10.1016/j.csl.2022.101429. URL: https://www.sciencedirect.com/science/article/pii/S0885230822000596.

[10] Y. Gal and Z. Ghahramani. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by M. F. Balcan and K. Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1050–1059. URL: https://proceedings.mlr.press/v48/gal16.html.

[11] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *arXiv preprint arXiv:1907.11692* (2019).

[12] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le. "XLNet: Generalized Autoregressive Pretraining for Language Understanding". In: *CoRR* abs/1906.08237 (2019). arXiv: 1906.08237. URL: http://arxiv.org/abs/1906.08237.

[13] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[14] PyTorch Contributors. *CrossEntropyLoss — PyTorch 2.11 Documentation*. Accessed: 2026-04-23. PyTorch. 2024. URL: https://docs.pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html.

[15] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin. "Emerging Properties in Self-Supervised Vision Transformers". In: *Proceedings of the International Conference on Computer Vision (ICCV)*. 2021.

[16] PyTorch Team. *SGD — PyTorch 2.11 Documentation*. Accessed: 2026-04-23. PyTorch. 2024. URL: https://docs.pytorch.org/docs/stable/generated/torch.optim.SGD.html.

[17] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization". In: *arXiv preprint arXiv:1610.02391* (2016).

[18] T. Rheude, A. Wirtz, A. Kuijper, and S. Wesarg. "Leveraging CAM Algorithms for Explaining Medical Semantic Segmentation". In: *arXiv preprint arXiv:2409.20287* (2024). URL: https://arxiv.org/abs/2409.20287.

[19] W. Yao, J. Bai, W. Liao, Y. Chen, M. Liu, and Y. Xie. "From CNN to Transformer: A Review of Medical Image Segmentation Models". In: *Journal of Imaging Informatics in Medicine* 37.4 (2024), pp. 1529–1547. DOI: 10.1007/s10278-024-00981-7. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC11300773/.

[20] Hugging Face. *ConvNeXT · Hugging Face Transformers Documentation*. Model released 2022-01-10, added to Transformers 2022-02-07. 2022. URL: https://huggingface.co/docs/transformers/model_doc/convnext.

[21] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie. "A ConvNet for the 2020s". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022.

[22] D. Hendrycks and K. Gimpel. "Gaussian Error Linear Units (GELUs)". In: *arXiv preprint arXiv:1606.08415* (2016). URL: https://arxiv.org/abs/1606.08415.

[23] Perplexity. *Perplexity AI Assistant (powered by GPT-5.1)*. https://www.perplexity.ai/. Accessed: 2026-04-21. 2026.