Project 1 - Fundamentals of Neural Networks

5LSH0 - COMPUTER VISION AI AND 3D DATA ANALYSIS

Date	Version
09.12.2024	1

Author
Jakub Frydrych

Contents

1 Multi-Layer Perceptron

1.1 Implementation

This project focuses on basics of neural networks and its working. The aim of the first part of this work is to implement from scratch a Multilayer Perceptron to classify digits in the MNIST dataset. Designed layers should incorporate both forward and backward pass. Important part of this implementation is not relying on already existing solutions like PyTorch or similar libraries. Proposed solution has been programmed in Python using NumPy and Pandas libraries.

Proposed Neural Network consists of two layers (not counting input layer): hidden layer, output layer. Implemented Code allows for some form of modularity, with minimal modifications a 3 or more-layer network can be implemented.

1.1.1 Fully-connected layer with bias

The important thing to understand when dealing with MLP is that all calculations with regards to backward or forward propagation can be done in form of Matrix multiplication, so in fact a single perceptron isn't being programmed here, but multiple matrices consisting of weights, biases, inputs and outputs. These matrices are defined as NumPy arrays are named as follows:

Name	Explanation	Initial value range	Initialization
values	NN input in form of pixel values	[0, 1]	Scaled from [0, 255] pixel values
W1	First layer weights	[-0.5, 0.5]	random
b1	First layer biases	[-0.5, 0.5]	random
W2	Second layer weights	[-0.5, 0.5]	random
b2	Second layer biases	[-0.5, 0.5]	random
Z1	Linear Transformations	Undefined	Computed using forward propagation
Z2	Linear Transformations	Undefined	Computed using forward propagation
A1	Layer output after activation function	[0, ∞] or [0, 1]	Computed using ReLu/Sigmoid
A2	Layer output after activation function	[0, 1]	Computed using Softmax

Forward propagation is conducted with following equations:

$$Z_1 = W_1 * values + b_1$$
$$A_1 = ReLu(Z_1) \text{ or } A_1 = Sigmoid(Z_1)$$
$$Z_2 = W_2 * A_1 + b_2$$
$$A_2 = Softmax(Z_2)$$

Backward propagation is conducted with following equations:

$$dZ_2 = A_2 - one\_hot\_labels$$
$$dW_2 = \frac{1}{m} * dZ_2 * A_1^T$$
$$db_2 = \frac{1}{m} * \sum dZ_2$$

$$dZ_1 = W_2^T * dZ_2 * ReLu'(Z_1)$$
$$dW_1 = \frac{1}{m} * dZ_1 * values^T$$

$$db_1 = \frac{1}{m} * \sum dZ_1$$

Where:
m – number of training samples
one_hot_labels – One-hot encoded true labels
dZ1, dZ2 – layer errors
dW1 dW2 – layer gradient weights
db1, db2 – biases gradients
ReLu' – derivative of ReLu function

1.1.2 ReLU and Sigmoid activations

Both ReLu and Sigmoid activation functions have been implemented with their corresponding derivatives used in backpropagation. Derivatives for both activation functions are described by following equations:

$$ReLu'(x) = \begin{cases} 0 \; if \; x < 0 \\ 1 \; if \; x > 0 \end{cases}$$

$$Sigmoid'(x) = Sigmoid(x) * (1 - sigmoid(x))$$

1.1.3 Mini-batch Gradient Descent (SGD) and Cross-Entropy loss

First attempt focused on implementing simpler, Batch Gradient Descent (BGD) to ensure the rest of the code is functioning properly. Results from this attempt were also used for comparison between this simpler solution and Mini-batch Gradient Descent (MBGD).

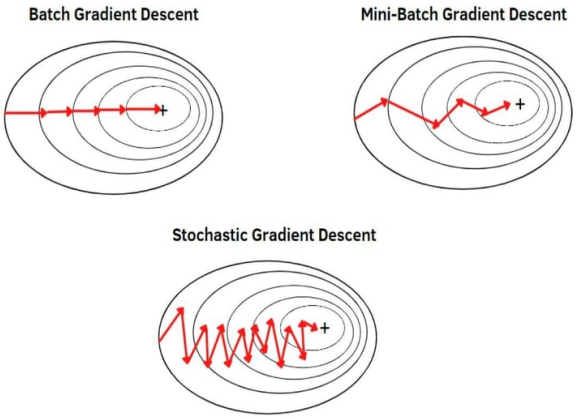

Figure 1: Comparison of gradient calculation methods

MBGD algorithm exists as a compromise between BGD in which gradient is being calculated across whole dataset, and SGD where a single data point is used. Not relying on whole batch in calculations introduces some noise into resulting gradient direction, helping avoid local minima but decreasing stability. Choosing larger mini-batches helps in reducing that noise. Usually choosing a multiple of CPU (or GPU/TPU) threads is recommended, however in this "from scratch" solution everything is being calculated sequentially. For this solution a batch size of 64 has been used.

For this implementation Cross-Entropy loss has been used. Compared to Sum of Squared Errors (SSE) it results in larger loss values for worse predictions, resulting in larger steps in parameter changes and in result quicker NN training.

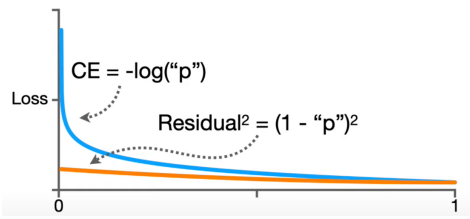

Figure 2: Loss function comparison

1.2 Training and testing

1.2.1 Train the MLP and report the training and test accuracies. Explain why there are differences in training and test performance?

First training the proposed MLP with Batch Gradient Descent.

Hyperparameters:

epochs: 2000, alpha: 0.2, activation: sigmoid

Results:

training time: 3432s, test accuracy: 0.925, test loss: 0.300

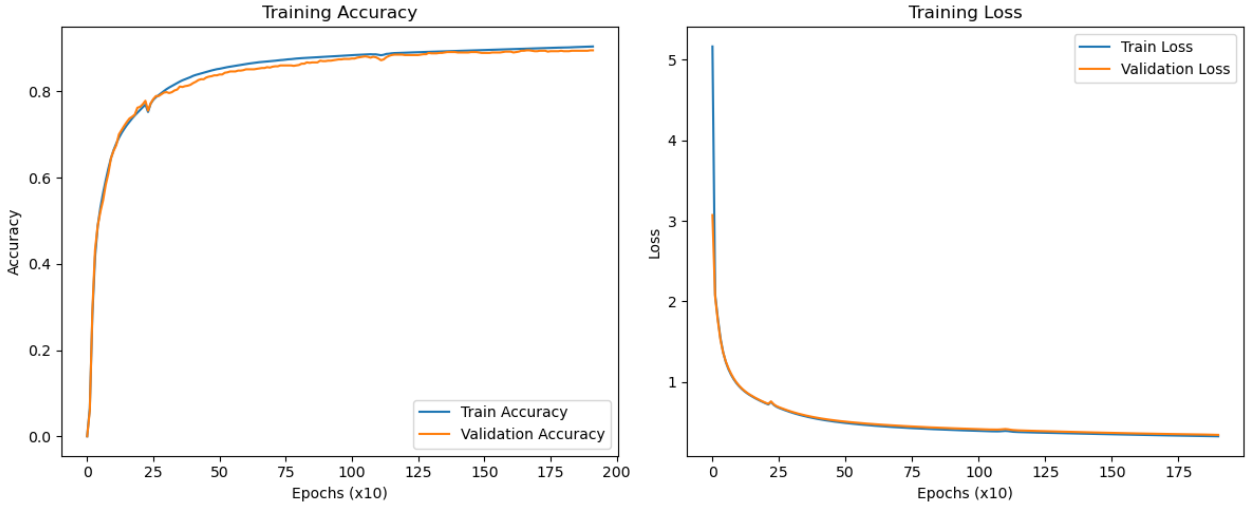

Figure 3 BGD example model training

Moving to Mini-Batch Gradient Descent:

Hyperparameters:

epochs: 200, alpha: 0.1, activation: sigmoid, batch size: 64

Results:

training time: 620s, test accuracy: 0.933, test loss: 0.237

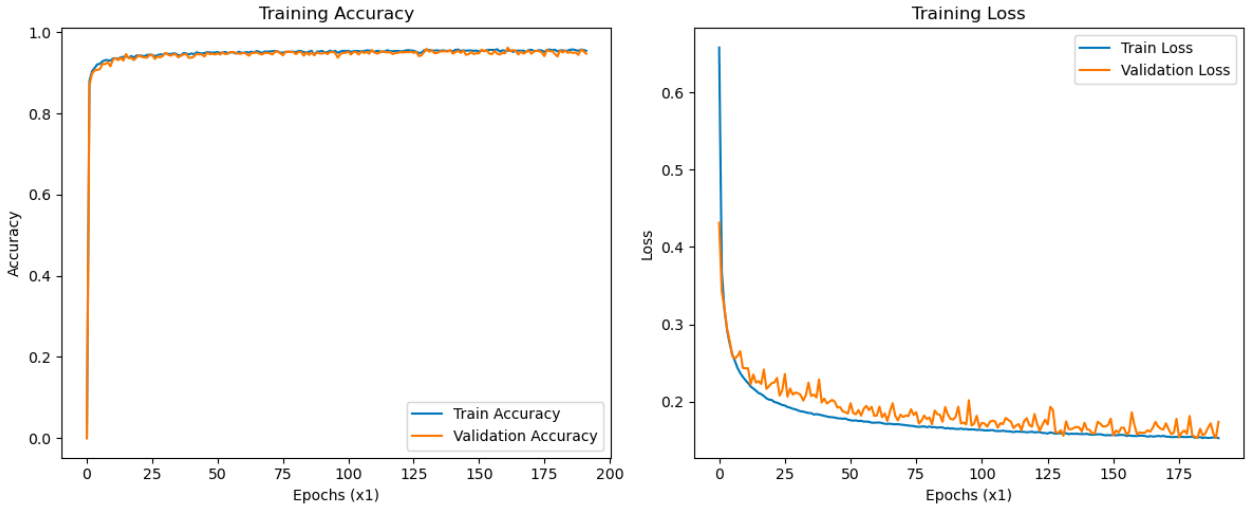

Figure 4: MBGD_64 model training example

The resulting training with mini batches of size 64 has been almost too fast. After just first epoch, the model achieved accuracy of over 0.85. This can be explained by the fact that in MBGD model parameters are being updated after each batch, so after first epoch for batch size 64 the model parameters were already updated

close to 1000 times (MNIST train dataset has close to 70000 datapoints, so in this case there were over 1000 batches per epoch).

Train loss is being calculated as a mean average of all batch's loss, so its naturally smoothed out, however validation loss is clearly noisy due to much more chaotic parameters updates.

Hyperparameters:

epochs: 200, alpha: 0.1, activation: sigmoid, batch size: 1024

Results:

training time: 583s, test accuracy: 0.932, test loss: 0.226

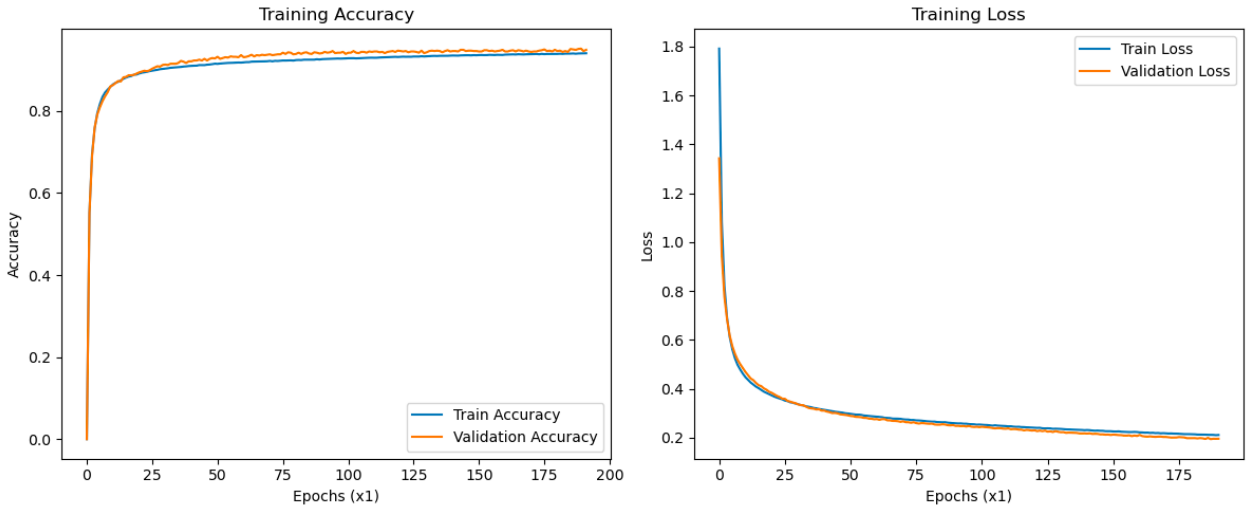

Figure 5: MBGD_1024 model training example

Results have been gathered in a table:

		BGD	MBGD_64	MBGD_1024
Accuracy	**Train**	0.921	0.956	0.941
	Test	0.925	0.938	0.932
Loss	**Train**	0.276	0.146	0.208
	Test	0.300	0.231	0.226

Figure 6: Results from different model trainings

A trend can be noticed for all MBGD models: train accuracy has been higher than test accuracy. Similarly resulting loss values were larger for test set compared to train set. For BGD model no significant difference in accuracy has been noticed. Surprisingly test loss has been higher than train loss.

These differences arise from models' ability to generalize. In this case slight overfitting to train set has been observed. Overfitting takes place when model learns training specific patterns that don't represent more general, task specific features. Strategies like data augmentation or early stopping could be implemented. If validation accuracy would visibly start to decrease compared to train accuracy, shorter training times would be suggested.

1.2.2 Calculate the number of trainable parameters of your model (you can implement it or calculate by hand) and explain how you have estimated it?

Trainable model parameters for MLP with fully-connected layers consist of weights and biases. In proposed solution 2 layers were implemented, with neuron count corresponding to pixel count of input. MNIST dataset consist of 28x28 images, so 784 pixels. The layer count goes as follows:

First Layer (input to hidden):

Input: 784, hidden layer size: 10, Parameters = (784*10) + 10 = 7850

Second Layer (Hidden to Output):

Output size: 10, Parameters = (10*10) +10 = 110

Together:

Total Parameters = 7850 + 110 = 7960

Which can be confirmed by simply running:
param_count = W1.size + b1.size + W2.size + b2.size

1.2.3 How would the number of parameters vary if you use a convolution layer with batch normalization layer instead of a fully-connected layer (explain)? What is the relation between a convolution and fully-connected layer, when are they the same/different?

Convolution layers act as a way of reshaping neural networks into filters. In that case one neuron isn't connected to all previous, but only a few neurons in its perceptive field. What's important here, is that wights here are shared to mimic the behaviour of filter sliding over the image. They aim to reduce the size of the following layers while preserving spatial information**,** effectively reducing the number of trainable parameters

However, batch normalization layer introduces additional trainable parameters corresponding to scale and shift for normalization. However, combination of convolution layer with batch normalization leads to reduction in trainable parameters compared to fully-connected layers. This results in higher computational efficiency while preserving useful information for tasks like digit classification.

When fully-connected layer and Convolution layer are the same (kernel size matches input dimension), this means that both can be treated as convolution layer with global filter size, and number of filters equal to number output neurons. One could argue that in that case, in fully-connected layer all global filters could have different weights and biases where in convolution network all filters would share them.

In case they are different, usually convolutional layers are used at the beginning while fully-connected layers are used at the end of Neural Network. This ensures that at first feature extraction with size reduction happens, after which classification or decision making happens.

1.3 PyTorch CNN implementation

1.3.1 Explain in detail the improvements and changes that you have added. Well reasoned answers will get more/full points.

With allowance of PyTorch usage several improvements have been implemented. Two main goals were improving model accuracy compared to previously implemented FCN, and decreasing training time. To achieve these goals, following improvements have been introduced:

Changed Network structure – Instead of FCN a CNN has been used. Its structure consists of two convolutional layers, each using ReLU activation function and following max pooling operations. The first layer (conv1) applies 32 filters of size 3x3, while the second layer (conv2) applies 64 filters of the same size. Max pooling with 2x2 kernel with stride of 2 is using after convolution operations to reduce spatial dimensions. Extracted features are flattened and passed through a FCN layers. First of them (fc1) has 128 neurons and is followed by another layer (fc2) with 10 output neurons. To achieve final classification a LogSoftmax activation function has been used. This structure helps capture spatial patterns in input data while reducing network complexity.

Using built in PyTorch functionalities – PyTorch provides optimized implementations of many operations commonly used when working with neural networks. Compared to initial implementations where each functionality has been programmed "from scratch", this solution leverages built in functions.

Using Adam optimizer – During lectures Adam optimizer has been recommended as better alternative to standard gradient descent solutions like SGD or MBGD. By comparison, Adam takes advantage of adaptive learning rate alternatives leading to faster convergence and better overall performance. This helped in reducing training time while maintaining resulting model accuracy.

Hardware accelerated training – "From scratch" implementation only leveraged single thread execution. PyTorch allows for further optimization by performing hardware accelerated NN training using either GPU or multiple CPU threads. This resulted in drastically faster training times by better utilizing available hardware resources.

These improvements led to much better model training and resulting performance. Results have been gathered below:

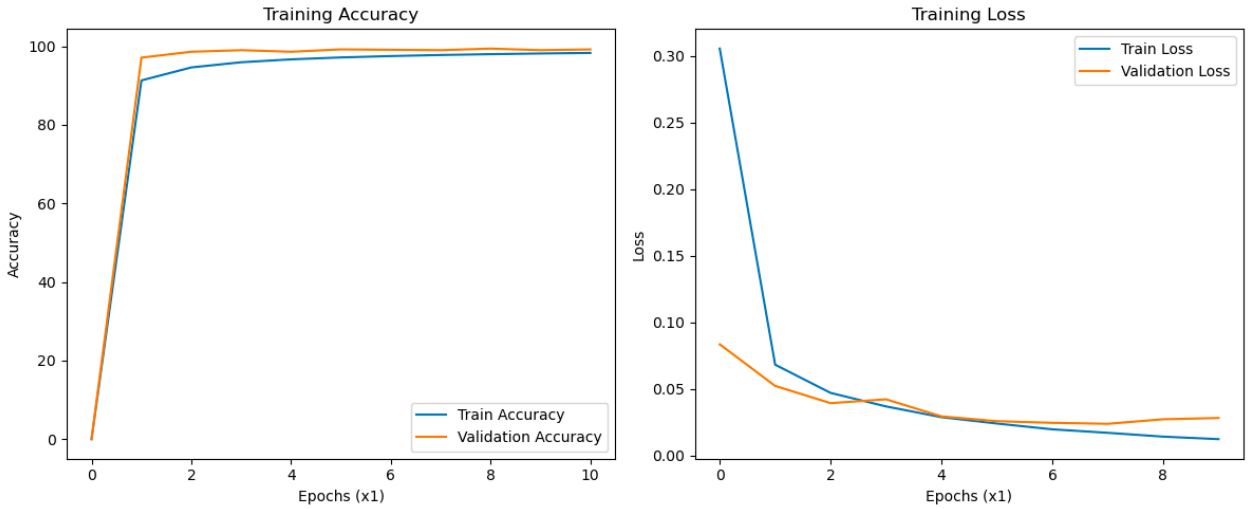

Figure 7: PyTorch CNN training example

		Best FCN: MBGD_64	PyTorch CNN
Accuracy	**Train**	0.956	0.998
	Test	0.938	0.994
Loss	**Train**	0.146	0.019
	Test	0.231	0.037
Training time		586 seconds	246 seconds

Set out accuracy for test set of above 99% has been achieved with improved training time. One additional improvement worth exploring would be implementing early training stopping. For PyTorch CNN manually setting training length has been unreliable, with validation loss often increasing towards later stages of training. This could be a sign of overfitting train data, which is an undesirable behaviour.

1.3.2 Given that you have unlimited resources for computation and no extra data, how would you improve performance on the MNIST dataset?

Given limited data availability but increased computational resources several improvements have been identified:

Data Augmentation – With limited data, artificial data generation can be performed by applying transformations such as rotation, translation, skewing, adding noise, or elastic deformations. These techniques help the model generalize better and improve robustness without requiring additional real-world data.

Hyperparameter Tuning – With unlimited compute, extensive hyperparameter searches can be conducted. Key parameters such as learning rate, batch size or optimizer type can be fine-tuned to achieve the best performance.

Further Model Optimization – Techniques such as regularization, weight decay or dropout could be used to prevent overfitting at cost of increasing training time.

Increased Model Complexity – increasing layer count and modifying their structure could lead to improved performance. Trying different model configurations could be a part of hyperparameter tuning. Additionally, knowledge transfer from other, already established neural networks could be considered.

2 Loss functions

2.1.1 Derive the derivative of the output y with respect to logits z.

Softmax function is defined as follows:

$$y = \sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Derivative is defined as follows:

$$\frac{\partial y_i}{\partial z_i} = \frac{\partial}{\partial z_i}\left(\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\right) = \frac{\partial}{\partial z_i}\left(\frac{e^{z_i}}{S}\right)$$

Using the quotient rule for $i = j$:

$$\frac{\partial y_i}{\partial z_i} = \frac{\partial}{\partial z_i}\left(\frac{e^{z_i}}{S}\right) = \frac{e^{z_i} * S - e^{z_i} * \frac{\partial S}{\partial z_i}}{S^2} = \frac{e^{z_i}(S - e^{z_i})}{S^2} = y_i\left(1 - \frac{e^{z_i}}{S}\right) = y_i(1 - y_i)$$

Using the quotient rule and chain rule for $i \neq j$:

$$\frac{\partial y_i}{\partial z_j} = \frac{\partial}{\partial z_j}\left(\frac{e^{z_i}}{S}\right) = e^{z_i} * \frac{\partial}{\partial z_j}\left(\frac{1}{S}\right) = e^{z_i} * \left(-\frac{1}{S^2} * \frac{\partial S}{\partial z_j}\right) = e^{z_i} * \left(-\frac{1}{S^2} * e^{z_j}\right) = -\frac{e^{z_i} * e^{z_j}}{S^2} = -y_i y_j$$

2.1.2 Use the condition when i= j, and derive the derivative of the loss function Lce with respect to the logits z.

Cross-entropy loss is defined as:

$$L_{ce}(y, t) = -t * \log(y) - (1 - t) * \log(1 - y)$$

Using chain rule:

$$\frac{\partial L_{ce}}{\partial z} = \frac{\partial L_{ce}}{\partial y} * \frac{\partial y}{\partial z} = \left(-\frac{t}{y} + \frac{1 - t}{1 - y}\right)\frac{\partial y}{\partial z}$$

Using softmax derivative for $i = j$:

$$\frac{\partial L_{ce}}{\partial z} = \left(-\frac{t}{y} + \frac{1 - t}{1 - y}\right)\frac{\partial y}{\partial z} = \left(-\frac{t}{y} + \frac{1 - t}{1 - y}\right) * y(1 - y) = -t(1 - y) + (1 - t)y = y - t$$

2.1.3 Show that… (dice loss derivative)

Dice loss is defined as:

$$L_{dice}(y, t) = 1 - \frac{2yt}{y + t}$$

Using chain rule:

$$\frac{\partial L_{dice}}{\partial z} = \frac{\partial L_{dice}}{\partial y} * \frac{\partial y}{\partial z} = \frac{\partial}{\partial y}\left(1 - \frac{2yt}{y + t}\right) * \frac{\partial y}{\partial z} = \left(\frac{2t(y + t) - 2yt}{(y + t)^2}\right) * \frac{\partial y}{\partial z} = \left(\frac{2t^2}{(y + t)^2}\right) * \frac{\partial y}{\partial z}$$

Using softmax derivative

$$\frac{\partial L_{dice}}{\partial z} = \left(\frac{2t^2}{(y + t)^2}\right) * \frac{\partial y}{\partial z} = -\frac{2t^2}{(y + t)^2} * y(1 - y) = -\frac{2t^2 y(1 - y)}{(y + t)^2}$$

From L_{dice}definition:

$$1 - L_{dice} = \frac{2yt}{y + t}$$

$$(1 - L_{dice})^2 = \frac{4y^2t^2}{(y + t)^2} \quad \rightarrow \quad \frac{2t^2}{(y + t)^2} = \frac{(1 - L_{dice})^2}{2y^2}$$

Combining:

$$\frac{\partial L_{dice}}{\partial z} = -\frac{2t^2 y(1-y)}{(y+t)^2} = -\frac{(1-L_{dice})^2(1-y)}{2y} = \frac{(y-1)}{2y} * (1-L_{dice})^2$$

2.1.4 Expand the generic to the binary version of focal loss.

Generic focal loss is defined as:

$$L_{focal}(y,t) = -(1-y)^{\gamma} * t * log(y)$$

For the binary case:

$$L_{focal-binary}(y,t) = \begin{cases} -(1-y)^{\gamma} * \log(y) & for\ t = 1 \\ -y^{\gamma} \log(1-y) & for\ t = 0 \end{cases}$$

Combining:

$$L_{focal}(y,t) = -t * (1-y)^{\gamma} * \log(y) - (1-t) * y^{\gamma} * \log(1-y)$$

2.1.5 Show that … (focal loss derivative)

Using binary version of focal loss:

$$L_{focal-binary} = -t * (1-y)^{\gamma} * \log(y) - (1-t) * y^{\gamma} * \log(1-y)$$

Computing derivatives for two terms:

$$\frac{\partial}{\partial y}[-t * (1-y)^{\gamma} * \log(y)] = -t\left[\frac{(1-y)^{\gamma}}{y} - \gamma * (1-y)^{\gamma-1} * \log(y)\right]$$

$$\frac{\partial}{\partial y}[-(1-t) * y^{\gamma} * \log(1-y)] = -(1-t)\left[\frac{y^{\gamma}}{1-y} + \gamma * y^{\gamma-1} * \log(1-y)\right]$$

$$\frac{L_{focal-binary}}{\partial y} = -t\left[\frac{(1-y)^{\gamma}}{y} - \gamma * (1-y)^{\gamma-1} * \log(y)\right] - (1-t)\left[\frac{y^{\gamma}}{1-y} + \gamma * y^{\gamma-1} * \log(1-y)\right]$$

Using $\mathrm{E}_p = -p * \log(p)$:

$$\log(y) = -\frac{\mathrm{E}_y}{y}, \qquad \log(1-y) = -\frac{\mathrm{E}_{1-y}}{1-y}$$

Substituting:

$$\frac{L_{focal-binary}}{\partial y} = -\frac{1}{y(1-y)}\left[t * (1-y)^{\gamma} * \left(\gamma * E_y + 1 - y\right) + (t-1) * y^{\gamma} * (\gamma * E_{1-y} + y\right]$$

2.1.6 Under what condition is focal loss and dice loss equal? What happens when γ value is too high or low in focal loss? Explain in detail how it would impact performance.

For $\frac{\partial L_{focal}}{\partial y} = \frac{\partial L_{ce}}{\partial y}$ to be true, following equation must be true:

$$\frac{\partial L_{focal}}{\partial y} = -\frac{1}{y * (1-y)}\left[t(1-y)^{\gamma} * \left(\gamma * E_y + 1 - y\right) + (t-1) * y^{\gamma} * (\gamma * E_{1-y} + y)\right] = \left(-\frac{t}{y} + \frac{1-t}{1-y}\right) = \frac{\partial L_{ce}}{\partial y}$$

This is only true for $\gamma = 0$, reducing focal loss to cross-entropy loss.

Gamma parameter in focal loss can be understood as a way of measuring how difficult, therefore important given sample is during training. For values close to 0, focal loss behaves similarly to standard cross entropy loss. Therefore, there isn't a big distinction between difficult and "easier" samples, and both have same impact on resulting gradient. This might be undesired in cases of large class imbalance, where samples from one class can overpower less sampled classes.

Too high values of gamma can lead to focal loss aggressively suppress contribution of "easier" samples, making their gradient contribution too small. While it forces model to focus on "hard" examples, it can lead to instability during training, slow convergence or overfitting to "difficult". Theres always a risk that those "hard"

samples which could be simply mislabelled or be in fact outliers, further confusing the model and leading to worse results.

2.1.7 List the applications where each loss function would be useful and explain why?

Cross-Entropy Loss – useful in classification tasks with balanced classes, results in well behaving gradients and larger "steps" for bigger differences compared to sum of squared errors (SSE).

Dice Loss – used for cases with class imbalance, especially image segmentation where interesting regions consist of much less pixels than background. This comes at a cost of much nosier gradient descent leading to potential instability.

Focal Loss – can be interpreted as compromise between dice and cross-entropy losses. Gamma can be treated as tuneable hyperparameter allowing for more control over model training and balancing "easy" and "difficult" samples.

2.1.8 Discuss (in detail) the behaviour of precision and recall for each of the listed loss functions above. Use the proofs from questions 2, 3 and 5 to support your answers.

Cross-Entropy Loss – In balanced datasets, cross-entropy loss balances precision and recall by treating all samples equally. This changes for imbalanced datasets where the majority class gets favorized. This leads to low recall for minority classes due to underpenalized false negatives. Average precision may remain stable minority class precision may suffer. For tasks like segmentation, cross-entropy loss may lead to failure in detecting small objects. Looking at proof discussed in 2.1.2, there is no mechanism to prioritise "hard" examples. Calculated derivative $\frac{\partial L_{ce}}{\partial z} = y - t$ has a linear behaviour and resulting gradient is proportional to calculated error.

Dice Loss – It's designed for imbalanced tasks where majority class is considerable larger than the rest. Its value increases when predictions and true labels disagree, making it particularly sensitive to false negatives. This makes dice loss particularly effective at improving recall due to false negatives punishment, however this comes at the cost of precision. Proof discussed in 2.1.3 shows how Dice Loss derivative includes $(1 - L_{dice})^2$ term, which diminishes as predictions improve. However, even small changes to predictions can introduce a lot of noise to resulting gradient.

Focal Loss – Calculations from 2.1.5 clearly shows extra factors $(1 - y)^\gamma$ for positives and y^γ for negatives that modify resulting loss. By tunning γ parameter, Focal loss sensitivity to dataset imbalance can be adjusted. High γ values aggressively suppress "easy examples" improving recall for minority classes at cost of lower precision. Low γ values make focal loss less sensitive for class imbalances, while $\gamma = 0$ mimics cross-entropy loss. Adjustable loss parameter is a powerful tool allowing for class imbalance compensation while ensuring minimal noise and faster model convergence.