Emanuele Belli 03720425
Thomas Brunner 03675118
Luca Corbucci 03726968

Exercise Sheet 09

Machine Learning
January 5, 2020

.

Problem 1

a) $f_w(x_i) = w_{L+1}^T(R(W_L R(W_{L-1} R(....W_1 x_i))) = W_{L+1}^T f(\widetilde{W}_L, x_i)$

In general it holds:

$$min_W \frac{1}{2}\sum_i (f_w(x_i) - y_i)^2 =$$

$$= min_{\widetilde{W}_L, W_{L+1}} \frac{1}{2}\sum_i (w_{L+1}^T f(\widetilde{W}_{L+1}, x_i) - y_i)^2 \leq$$

(Set $\widetilde{W}_L$ s.t. $f(\widetilde{W}_{,\cdot}) = Id$)

$$\leq min_{\widetilde{W}_{L+1}} \frac{1}{2}\sum_i (w_{T+1}^T x_i - y_i)^2 = L_{LS}(W_{LS}) \tag{1}$$

In the case mentioned that $W > 0$ (for any weight matrix), each of the inputs is non negative our network collapses to a linera function.

$$L_{NN}(W_{NN}) = min_W \frac{1}{2}\sum_i (f_w(x_i) - y_i)^2 =$$

$$min_{w \in R_+^D} \sum_i (w^T x_i - y_i)^2 \geq$$

$$\geq min_{w \in R^D} \sum_i (w^T x_i - y_i)^2 = L_{LS}(W_{LS}^*)$$

Under these assumptions we conclude that:

$$L_{NN}(W_{NN}^*) = L_{LS}(W_{LS}^*)$$

b) In this case the relation is $\leq$ and it follows from the inequality 1.

In [1]:

```
import copy
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
```

PyTorch

In this notebook you will gain some hands-on experience with PyTorch (https://pytorch.org/), one of the major frameworks for deep learning. To install PyTorch run `conda install pytorch torchvision cudatoolkit=10.1 -c pytorch` , with cudatoolkit set to whichever CUDA version you have installed. You can check this by running `nvcc --version` . If you do not have an Nvidia GPU you can run `conda install pytorch torchvision cpuonly -c pytorch` instead. However, in this case we recommend using Google Colab (https://colab.research.google.com/).

You will start by re-implementing some common features of deep neural networks (dropout and batch normalization) and then implement a very popular modern architecture for image classification (ResNet) and improve its training loop.

1. Dropout

Dropout is a form of regularization for neural networks. It works by randomly setting activations (values) to 0, each one with equal probability `p` . The values are then scaled by a factor $\frac{1}{1-p}$ to conserve their mean.

Dropout effectively trains a pseudo-ensemble of models with stochastic gradient descent. During evaluation we want to use the full ensemble and therefore have to turn off dropout. Use `self.training` to check if the model is in training or evaluation mode.

Do not use any dropout implementation from PyTorch for this!

In [0]:

```
class Dropout(nn.Module):
    """
    Dropout, as discussed in the lecture and described here:
    https://pytorch.org/docs/stable/nn.html#torch.nn.Dropout

    Args:
        p: float, dropout probability
    """
    def __init__(self, p):
        super().__init__()
        self.p = p

    def forward(self, input):
        """
        The module's forward pass.
        This has to be implemented for every PyTorch module.
        PyTorch then automatically generates the backward pass
        by dynamically generating the computational graph during
        execution.

        Args:
            input: PyTorch tensor, arbitrary shape

        Returns:
            PyTorch tensor, same shape as input
        """
        # TODO: Set values randomly to 0.
        if self.training:
            scalar = 1/(1-self.p)
            N = len(input)
            probabilities = torch.from_numpy(np.random.binomial(1,
            return torch.mul(input, probabilities)
        else:
            return input
```

In [0]:

```
# Test dropout
test = torch.ones(10_000)
dropout = Dropout(0.5)
test_dropped = dropout(test)
# These assertions can in principle fail due to bad luck, but
# if implemented correctly they should almost always succeed.
assert np.isclose(test_dropped.sum().item(), 10_000, atol=400)
assert np.isclose((test_dropped > 0).sum().item(), 5_000, atol=200)
```

2. Batch normalization

Batch normalization is a trick use to smoothen the loss landscape and improve training. It is defined as the function

$$y = \frac{x - \mu_x}{\sigma_x + \epsilon} \cdot \gamma + \beta$$

, where γ and β and learnable parameters and ϵ is a some small number to avoid dividing by zero. The Statistics μ_x and σ_x are taken separately for each feature. In a CNN this means averaging over the batch and all pixels.

Do not use any batch normalization implementation from PyTorch for this!

In [0]:

```
class BatchNorm(nn.Module):
    """
    Batch normalization, as discussed in the lecture and similar to
    https://pytorch.org/docs/stable/nn.html#torch.nn.BatchNorm1d

    Only uses batch statistics (no running mean for evaluation).
    Batch statistics are calculated for a single dimension.
    Gamma is initialized as 1, beta as 0.

    Args:
        num_features: Number of features to calculate batch statist
    """
    def __init__(self, num_features):
        super().__init__()

        # TODO: Initialize the required parameters
        self.gamma = nn.Parameter(torch.ones(num_features))
        self.beta = nn.Parameter(torch.zeros(num_features))
        self.num_features = num_features

    def forward(self, input):
        """
        Batch normalization over the dimension C of (N, C, L).

        Args:
            input: PyTorch tensor, shape [N, C, L]

        Return:
            PyTorch tensor, same shape as input
        """
        eps = 1e-5

        # TODO: Implement the required transformation

        mean_sub = input - input.mean(dim=[0,2]).view(1,2,1)
        var_e = np.sqrt(input.var(dim=[0,2]).view(1,2,1) + eps)


        return (mean_sub/var_e)*self.gamma[:,None]+self.beta[:,None
```

In [0]:

```
# Tests the batch normalization implementation
torch.random.manual_seed(42)
test = torch.randn(8, 2, 4)

b1 = BatchNorm(2)
test_b1 = b1(test)

b2 = nn.BatchNorm1d(2, affine=False, track_running_stats=False)
test_b2 = b2(test)

assert torch.allclose(test_b1, test_b2, rtol=0.02)
```

3. ResNet

ResNet is the models that first introduced residual connections (a form of skip connections). It is a rather simple, but successful and very popular architecture. In this part of the exercise we will re-implement it step by step.

Note that there is also an improved version of ResNet (https://arxiv.org/abs/1603.05027) with optimized residual blocks. Here we will implement the original version (https://arxiv.org/abs/1512.03385) for CIFAR-10. Your dropout and batchnorm implementations won't help you here. Just use PyTorch's own layers.

This is just a convenience function to make e.g. `nn.Sequential` more flexible. It is e.g. useful in combination with `x.squeeze()` .

In [0]:

```
class Lambda(nn.Module):
    def __init__(self, func):
        super().__init__()
        self.func = func

    def forward(self, x):
        return self.func(x)
```

We begin by implementing the residual blocks. The block is illustrated by this sketch:

Residual connection

Note that we use 'SAME' padding, no bias, and batch normalization after each convolution. You do not need `nn.Sequential` here. The skip connection is already implemented as `self.skip` . It can handle different strides and increases in the number of channels.

In [0]:

```
class ResidualBlock(nn.Module):
    """
    The residual block used by ResNet.

    Args:
        in_channels: The number of channels (feature maps) of the i
        out_channels: The number of channels after the first convol
        stride: Stride size of the first convolution, used for down
    """

    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        if stride > 1 or in_channels != out_channels:
            # Add strides in the skip connection and zeros for the
            self.skip = Lambda(lambda x: F.pad(x[:, :, ::stride, ::
                                               (0, 0, 0, 0, 0, out_
                                               mode="constant", val
        else:
            self.skip = nn.Sequential()

        # TODO: Initialize the required layers

        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_si
        self.batchNormalization1 = nn.BatchNorm2d(out_channels)

        self.activation_function = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_s
        self.batchNormalization2 = nn.BatchNorm2d(out_channels)


    def forward(self, input):
        # TODO: Execute the required layers and functions

        x = self.conv1(input)
        x = self.batchNormalization1(x)
        x = self.activation_function(x)

        x = self.conv2(x)
        x = self.batchNormalization2(x)
        x += self.skip(input)
        x = self.activation_function(x)

        return x
```

Next we implement a stack of residual blocks for convenience. The first layer in the block is the one changing the number of channels and downsampling. You can use `nn.ModuleList` to use a list of child modules.

In [0]:

```
class ResidualStack(nn.Module):
    """
    A stack of residual blocks.

    Args:
        in_channels: The number of channels (feature maps) of the i
        out_channels: The number of channels after the first layer
        stride: Stride size of the first layer, used for downsampli
        num_blocks: Number of residual blocks
    """

    def __init__(self, in_channels, out_channels, stride, num_block
        super().__init__()

        # TODO: Initialize the required layers (blocks)
        self.blocks = nn.ModuleList([ResidualBlock(in_channels, out

        for _ in range(num_blocks-1):
            self.blocks.append(ResidualBlock(out_channels, out_chan

    def forward(self, input):
        # TODO: Execute the layers (blocks)
        for block in self.blocks:
            input = block(input)
        return input
```

Now we are finally ready to implement the full model! To do this, use the `nn.Sequential` API and carefully read the following paragraph from the paper (Fig. 3 is not important):

ResNet CIFAR10 description

Note that a convolution layer is always convolution + batch norm + activation (ReLU), that each ResidualBlock contains 2 layers, and that you might have to `squeeze` the embedding before the dense (fully-connected) layer.

In [0]:

```
n = 5
num_classes = 10

# TODO: Implement ResNet via nn.Sequential
resnet = nn.Sequential(
    ResidualStack(3,16,1,1),

    ResidualStack(16,16,1,2*n),
    ResidualStack(16,32,2,2*n),
    ResidualStack(32,64,2,2*n),

    nn.AdaptiveAvgPool2d(1),
    Lambda(lambda x: torch.squeeze(x)),
    nn.Linear(64, num_classes)
)
```

Next we need to initialize the weights of our model.

In [0]:

```
def initialize_weight(module):
    if isinstance(module, (nn.Linear, nn.Conv2d)):
        nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
    elif isinstance(module, nn.BatchNorm2d):
        nn.init.constant_(module.weight, 1)
        nn.init.constant_(module.bias, 0)

resnet.apply(initialize_weight);
```

4. Training

So now we have a shiny new model, but that doesn't really help when we can't train it. So that's what we do next.

First we need to load the data. Note that we split the official training data into train and validation sets, because you must not look at the test set until you are completely done developing your model and report the final results. Some people don't do this properly, but you should not copy other people's bad habits.

In [0]:
```
class CIFAR10Subset(torchvision.datasets.CIFAR10):
    """
    Get a subset of the CIFAR10 dataset, according to the passed in
    """
    def __init__(self, *args, idx=None, **kwargs):
        super().__init__(*args, **kwargs)

        if idx is None:
            return

        self.data = self.data[idx]
        targets_np = np.array(self.targets)
        self.targets = targets_np[idx].tolist()
```

We next define transformations that change the images into PyTorch tensors, standardize the values according to the precomputed mean and standard deviation, and provide data augmentation for the training set.

In [0]:
```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, 4),
    transforms.ToTensor(),
    normalize,
])
transform_eval = transforms.Compose([
    transforms.ToTensor(),
    normalize
])
```

In [13]:
```
ntrain = 45_000
train_set = CIFAR10Subset(root='./data', train=True, idx=range(ntra
                          download=True, transform=transform_train)
val_set = CIFAR10Subset(root='./data', train=True, idx=range(ntrain
                        download=True, transform=transform_eval)
test_set = torchvision.datasets.CIFAR10(root='./data', train=False,
                                        download=True, transform=tr
```

```
0it [00:00, ?it/s]

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.g
z (https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz) to ./d
ata/cifar-10-python.tar.gz

170500096it [00:09, 18398091.48it/s]

Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
Files already downloaded and verified
```

In [0]:

```
dataloaders = {}
dataloaders['train'] = torch.utils.data.DataLoader(train_set, batch
                                                   shuffle=True, nu
                                                   pin_memory=True)
dataloaders['val'] = torch.utils.data.DataLoader(val_set, batch_siz
                                                 shuffle=False, num
                                                 pin_memory=True)
dataloaders['test'] = torch.utils.data.DataLoader(test_set, batch_s
                                                  shuffle=False, nu
                                                  pin_memory=True)
```

Next we push the model to our GPU (if there is one).

In [0]:

```
device = torch.device('cuda') if torch.cuda.is_available() else tor
resnet.to(device);
```

Next we define a helper method that does one epoch of training or evaluation. We have only defined training here, so you need to implement the necessary changes for evaluation!

```
In [0]: def run_epoch(model, optimizer, dataloader, train):
            """
            Run one epoch of training or evaluation.

            Args:
                model: The model used for prediction
                optimizer: Optimization algorithm for the model
                dataloader: Dataloader providing the data to run our model
                train: Whether this epoch is used for training or evaluatio

            Returns:
                Loss and accuracy in this epoch.
            """
            # TODO: Change the necessary parts to work correctly during eva
            model.zero_grad()
            device = next(model.parameters()).device

            # Set model to training mode (for e.g. batch normalization, dro

            epoch_loss = 0.0
            epoch_acc = 0.0

            model.train() if train == True else model.eval()

            for xb, yb in dataloader:
                xb, yb = xb.to(device), yb.to(device)

                if train == True:
                    # zero the parameter gradients
                    optimizer.zero_grad()

                # forward
                with torch.set_grad_enabled(True):

                    pred = model(xb)
                    loss = F.cross_entropy(pred, yb)
                    top1 = torch.argmax(pred, dim=1)
                    ncorrect = torch.sum(top1 == yb)

                    if train == True:
                        loss.backward()
                        optimizer.step()

                # statistics
                epoch_loss += loss.item()
                epoch_acc += ncorrect.item()

            epoch_loss /= len(dataloader.dataset)
            epoch_acc /= len(dataloader.dataset)

            return epoch_loss, epoch_acc
```

Next we implement a method for fitting (training) our model. For many models early stopping can save a lot of training time. Your task is to add early stopping to the loop (based on validation accuracy). Early stopping usually means exiting the training loop if the validation accuracy hasn't improved for `patience` number of steps. Don't forget to save the best model parameters according to validation accuracy. You will need `copy.deepcopy` and the `state_dict` for this.

In [0]:

```
def fit(model, optimizer, lr_scheduler, dataloaders, max_epochs, pa
    """
    Fit the given model on the dataset.

    Args:
        model: The model used for prediction
        optimizer: Optimization algorithm for the model
        lr_scheduler: Learning rate scheduler that improves trainin
                      in late epochs with learning rate decay
        dataloaders: Dataloaders for training and validation
        max_epochs: Maximum number of epochs for training
        patience: Number of epochs to wait with early stopping the
                  training if validation loss has decreased

    Returns:
        Loss and accuracy in this epoch.
    """

    best_acc = 0
    curr_patience = 0


    for epoch in range(max_epochs):
        train_loss, train_acc = run_epoch(model, optimizer, dataloa
        lr_scheduler.step()
        print(f"Epoch {epoch + 1: >3}/{max_epochs}, train loss: {tr

        val_loss, val_acc = run_epoch(model, None, dataloaders['val
        print(f"Epoch {epoch + 1: >3}/{max_epochs}, val loss: {val_

        # TODO: Add early stopping and save the best weights (in be

        if best_acc < val_acc:
            best_model_weights = copy.deepcopy(model.state_dict())
            best_acc = val_acc
            curr_patience = 0
        else:
            curr_patience += 1

        if(curr_patience >= patience or epoch == max_epochs):
            break

    print(best_model_weights)
    model.load_state_dict(best_model_weights)
```

In most cases you should just use the Adam optimizer for training, because it works well out of the box. However, a well-tuned SGD (with momentum) will in most cases outperform Adam. And since the original paper gives us a well-tuned SGD we will just use that.

```
In [18]: optimizer = torch.optim.SGD(resnet.parameters(), lr=0.1, momentum=0
         lr_scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, mile

         # Fit model
         fit(resnet, optimizer, lr_scheduler, dataloaders, max_epochs=200, p
```

```
Epoch 172/200, train loss: 2.99e-05, accuracy: 99.93%
Epoch 172/200, val loss: 3.15e-03, accuracy: 93.16%
Epoch 173/200, train loss: 2.84e-05, accuracy: 99.95%
Epoch 173/200, val loss: 3.09e-03, accuracy: 93.34%
Epoch 174/200, train loss: 2.76e-05, accuracy: 99.95%
Epoch 174/200, val loss: 3.15e-03, accuracy: 93.14%
Epoch 175/200, train loss: 2.97e-05, accuracy: 99.93%
Epoch 175/200, val loss: 3.14e-03, accuracy: 93.12%
Epoch 176/200, train loss: 3.02e-05, accuracy: 99.93%
Epoch 176/200, val loss: 3.12e-03, accuracy: 93.14%
Epoch 177/200, train loss: 2.99e-05, accuracy: 99.94%
Epoch 177/200, val loss: 3.17e-03, accuracy: 93.10%
Epoch 178/200, train loss: 2.88e-05, accuracy: 99.94%
Epoch 178/200, val loss: 3.15e-03, accuracy: 93.22%
Epoch 179/200, train loss: 2.93e-05, accuracy: 99.94%
Epoch 179/200, val loss: 3.18e-03, accuracy: 93.28%
Epoch 180/200, train loss: 3.04e-05, accuracy: 99.93%
Epoch 180/200, val loss: 3.19e-03, accuracy: 93.32%
Epoch 181/200, train loss: 3.04e-05, accuracy: 99.93%
Epoch 181/200, val loss: 3.16e-03, accuracy: 93.34%
Epoch 182/200, train loss: 2.89e-05, accuracy: 99.94%
```

Once the model is trained we run it on the test set to obtain our final accuracy. Note that we can only look at the test set once, everything else would lead to overfitting. So you *must* ignore the test set while developing your model!

```
In [19]: test_loss, test_acc = run_epoch(resnet, None, dataloaders['test'],
         print(f"Test loss: {test_loss:.1e}, accuracy: {test_acc * 100:.2f}%
```

```
Test loss: 2.8e-03, accuracy: 92.13%
```

That's almost what was reported in the paper (92.49%) and we didn't even train on the full training set.

Optional task: Squeeze out all the juice!

Can you do even better? Have a look at A Recipe for Training Neural Networks (https://karpathy.github.io/2019/04/25/recipe/) and at the EfficientNet architecture (https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html) we discussed in the lecture. Play around with the possibilities PyTorch offers you and see how close you can get to the state of the art on CIFAR-10 (https://paperswithcode.com/sota/image-classification-on-cifar-10).

Hint: You can use Google Colab (https://colab.research.google.com/) to access some free GPUs for your experiments.

Type *Markdown* and LaTeX: α^2