CS6501: Deep Learning for Visual Recognition

Neural Networks
Today’s Class

Neural Networks
• The Perceptron Model
• The Multi-layer Perceptron (MLP)
• Forward-pass in an MLP (Inference)
• Backward-pass in an MLP (Backpropagation)
Perceptron Model

Frank Rosenblatt (1957) - Cornell University

\[ f(x) = \begin{cases} 
1, & \text{if } \sum_{i=0}^{n} w_i x_i + b > 0 \\
0, & \text{otherwise} 
\end{cases} \]

More: https://en.wikipedia.org/wiki/Perceptron
Perceptron Model

Frank Rosenblatt (1957) - Cornell University

\[ f(x) = \begin{cases} 
1, & \text{if } \sum_{i=0}^{n} w_i x_i + b > 0 \\
0, & \text{otherwise} 
\end{cases} \]

More: https://en.wikipedia.org/wiki/Perceptron
Perceptron Model

Frank Rosenblatt (1957) - Cornell University

\[ f(x) = \begin{cases} 
1, & \text{if } \sum_{i=0}^{n} w_i x_i + b > 0 \\
0, & \text{otherwise} 
\end{cases} \]

More: https://en.wikipedia.org/wiki/Perceptron
Activation Functions

**Step(x)**

**Tanh(x)**

**Sigmoid(x)**

**ReLU(x) = max(0, x)**
Two-layer Multi-layer Perceptron (MLP)
**Linear Softmax**

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \]
\[ y_i = [1 \ 0 \ 0] \]
\[ \hat{y}_i = [f_c \ f_d \ f_b] \]

\[ g_c = w_{c1}x_{i1} + w_{c2}x_{i2} + w_{c3}x_{i3} + w_{c4}x_{i4} + b_c \]
\[ g_d = w_{d1}x_{i1} + w_{d2}x_{i2} + w_{d3}x_{i3} + w_{d4}x_{i4} + b_d \]
\[ g_b = w_{b1}x_{i1} + w_{b2}x_{i2} + w_{b3}x_{i3} + w_{b4}x_{i4} + b_b \]

\[ f_c = \frac{e^{g_c}}{e^{g_c} + e^{g_d} + e^{g_b}} \]
\[ f_d = \frac{e^{g_d}}{e^{g_c} + e^{g_d} + e^{g_b}} \]
\[ f_b = \frac{e^{g_b}}{e^{g_c} + e^{g_d} + e^{g_b}} \]
Linear Softmax

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \]

\[ y_i = [1 \ 0 \ 0] \]

\[ \hat{y}_i = [f_c \ f_d \ f_b] \]

\[ g_c = w_{c1}x_{i1} + w_{c2}x_{i2} + w_{c3}x_{i3} + w_{c4}x_{i4} + b_c \]

\[ g_d = w_{d1}x_{i1} + w_{d2}x_{i2} + w_{d3}x_{i3} + w_{d4}x_{i4} + b_d \]

\[ g_b = w_{b1}x_{i1} + w_{b2}x_{i2} + w_{b3}x_{i3} + w_{b4}x_{i4} + b_b \]

\[ w = \begin{bmatrix} w_{c1} & w_{c2} & w_{c3} & w_{c4} \\ w_{d1} & w_{d2} & w_{d3} & w_{d4} \\ w_{b1} & w_{b2} & w_{b3} & w_{b4} \end{bmatrix} \]

\[ b = [b_c \ b_d \ b_b] \]

\[ f_c = \frac{e^{g_c}}{e^{g_c} + e^{g_d} + e^{g_b}} \]

\[ f_d = \frac{e^{g_d}}{e^{g_c} + e^{g_d} + e^{g_b}} \]

\[ f_b = \frac{e^{g_b}}{e^{g_c} + e^{g_d} + e^{g_b}} \]
Linear Softmax

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \]

\[ y_i = [1 \ 0 \ 0] \]

\[ \hat{y}_i = [f_c \ f_d \ f_b] \]

\[ g = wx^T + b^T \]

\[ w = \begin{bmatrix} w_{c1} & w_{c2} & w_{c3} & w_{c4} \\ w_{d1} & w_{d2} & w_{d3} & w_{d4} \\ w_{b1} & w_{b2} & w_{b3} & w_{b4} \end{bmatrix} \]

\[ b = [b_c \ b_d \ b_b] \]

\[ f_c = e^{g_c}/(e^{g_c} + e^{g_d} + e^{g_b}) \]

\[ f_d = e^{g_d}/(e^{g_c} + e^{g_d} + e^{g_b}) \]

\[ f_b = e^{g_b}/(e^{g_c} + e^{g_d} + e^{g_b}) \]
Linear Softmax

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \]

\[ y_i = [1 \ 0 \ 0] \]

\[ \hat{y}_i = [f_c \ f_d \ f_b] \]

\[ w = \begin{bmatrix} w_{c1} & w_{c2} & w_{c3} & w_{c4} \\ w_{d1} & w_{d2} & w_{d3} & w_{d4} \\ w_{b1} & w_{b2} & w_{b3} & w_{b4} \end{bmatrix} \]

\[ b = [b_c \ b_d \ b_b] \]

\[ g = wx^T + b^T \]

\[ f = \text{softmax}(g) \]
Linear Softmax

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \quad \quad y_i = [1 \ 0 \ 0] \quad \quad \hat{y}_i = [f_c \ f_a \ f_b] \]

\[ f = \text{softmax}(wx^T + b^T) \]
Two-layer MLP + Softmax

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \]

\[ y_i = [1 \ 0 \ 0] \]

\[ \hat{y}_i = [f_c \ f_a \ f_b] \]

\[ a_1 = \text{sigmoid}(w_{[1]}x^T + b_{[1]}^T) \]

\[ f = \text{softmax}(w_{[2]}x^T + b_{[2]}^T) \]
N-layer MLP + Softmax

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \]

\[ y_i = [1 \ 0 \ 0] \]

\[ \hat{y}_i = [f_c \ f_a \ f_b] \]

\[ a_1 = \text{sigmoid}(w_{[1]}x^T + b_{[1]}) \]

\[ a_2 = \text{sigmoid}(w_{[2]}a_1^T + b_{[2]}) \]

... 

\[ a_k = \text{sigmoid}(w_{[k]}a_{k-1}^T + b_{[k]}) \]

... 

\[ f = \text{softmax}(w_{[n]}a_{n-1}^T + b_{[n]}) \]
How to train the parameters?

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \]

\[ y_i = [1 \ 0 \ 0] \]

\[ \hat{y}_i = [f_c \ f_d \ f_b] \]

\[ \begin{align*}
    a_1 &= \text{sigmoid}(w_{[1]}x^T + b_{[1]}) \\
    a_2 &= \text{sigmoid}(w_{[2]}a_1^T + b_{[2]}) \\
    \ldots \\
    a_k &= \text{sigmoid}(w_{[k]}a_{k-1}^T + b_{[k]}) \\
    \ldots \\
    f &= \text{softmax}(w_{[n]}a_{n-1}^T + b_{[n]}) 
\end{align*} \]
Forward pass (Forward-propagation)

\[ z_i = \sum_{i=0}^{n} w_{1ij} x_i + b_1 \]

\[ a_i = \text{Sigmoid}(z_i) \]

\[ p_1 = \sum_{i=0}^{n} w_{2ia} + b_2 \]

\[ y_1 = \text{Sigmoid}(p_i) \]

\[ \text{Loss} = L(y_1, \hat{y}_1) \]
How to train the parameters?

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \quad \quad y_i = [1 \ 0 \ 0] \quad \quad \hat{y}_i = [f_c \ f_d \ f_b] \]

\[
\begin{align*}
  a_1 &= \text{sigmoid}(w_{[1]}x^T + b_{[1]}^T) \\
  a_2 &= \text{sigmoid}(w_{[2]}a_1^T + b_{[2]}^T) \\
        &\quad \cdots \\
  a_k &= \text{sigmoid}(w_{[k]}a_{k-1}^T + b_{[k]}^T) \\
        &\quad \cdots \\
  f &= \text{softmax}(w_{[n]}a_{n-1}^T + b_{[n]}^T)
\end{align*}
\]

We can still use SGD

We need!

\[
\frac{\partial l}{\partial w_{[k]ij}} \quad \quad \frac{\partial l}{\partial b_{[k]i}}
\]
How to train the parameters?

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \quad y_i = [1 \ 0 \ 0] \quad \hat{y}_i = [f_c \ f_a \ f_b] \]

\[
a_1 = \text{sigmoid}(w_{[1]} x^T + b_{[1]})
\]
\[
a_2 = \text{sigmoid}(w_{[2]} a_1^T + b_{[2]})
\]
\[
... \]
\[
a_i = \text{sigmoid}(w_{[k]} a_{k-1}^T + b_{[k]})
\]
\[
... \]
\[
f = \text{softmax}(w_{[n]} a_{n-1}^T + b_{[n]}) \]
\[
l = \text{loss}(f, y) \]

We can still use SGD

We need!

\[
\frac{\partial l}{\partial w_{[k]ij}} \quad \frac{\partial l}{\partial b_{[k]i}}
\]
How to train the parameters?

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \quad \quad \quad \quad y_i = [1 \ 0 \ 0] \quad \quad \quad \quad \hat{y}_i = [f_c \ f_a \ f_b] \]

\[
a_1 = \text{sigmoid}(w_{[1]}x^T + b_{[1]}^T)
\]

\[
a_2 = \text{sigmoid}(w_{[2]}a_1^T + b_{[2]}^T)
\]

\[ \quad \quad \quad \cdots \]

\[
a_i = \text{sigmoid}(w_{[k]}a_{k-1}^T + b_{[k]}^T)
\]

\[ \quad \quad \quad \cdots \]

\[
f = \text{softmax}(w_{[n]}a_{n-1}^T + b_{[n]}^T)
\]

\[
l = \text{loss}(f, y)
\]

We can still use SGD

We need!

\[ \frac{\partial l}{\partial w_{[k]ij}} \quad \quad \quad \quad \frac{\partial l}{\partial b_{[k]i}} \]
How to train the parameters?

\[ x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}] \]

\[ y_i = [1 \ 0 \ 0] \]

\[ \hat{y}_i = [f_c \ f_a \ f_b] \]

\[ a_1 = \text{sigmoid}(w_{[1]}x^T + b_{[1]}^T) \]

\[ a_2 = \text{sigmoid}(w_{[2]}a_1^T + b_{[2]}^T) \]

\[ \ldots \]

\[ a_i = \text{sigmoid}(w_{[k]}a_{k-1}^T + b_{[k]}^T) \]

\[ \ldots \]

\[ f = \text{softmax}(w_{[n]}a_{n-1}^T + b_{[n]}^T) \]

\[ l = \text{loss}(f, y) \]
Backward pass (Back-propagation)

\[
\frac{\partial L}{\partial x_k} = \left( \frac{\partial}{\partial x_k} \sum_{i=0}^{n} w_{1ij} x_i + b_1 \right) \frac{\partial L}{\partial z_i}
\]

\[
\frac{\partial L}{\partial z_i} = \frac{\partial}{\partial z_i} \text{Sigmoid}(z_i) \frac{\partial L}{\partial \alpha_k}
\]

\[
\frac{\partial L}{\partial \alpha_k} = \left( \frac{\partial}{\partial \alpha_k} \sum_{i=0}^{n} w_{2i} a_i + b_2 \right) \frac{\partial L}{\partial p_1}
\]

\[
\frac{\partial L}{\partial w_{2i}} = \frac{\partial a_k}{\partial w_{2i}} \frac{\partial L}{\partial a_k}
\]

\[
\frac{\partial L}{\partial \gamma_1} = \frac{\partial}{\partial \gamma_1} L(\gamma_1, \hat{\gamma}_1)
\]
# This class combines Softmax + Negative-log likelihood loss.
# Similar to the previous lab, but this implementation works for
# batches of inputs and not just individual input vectors.
# Here "inputs" is batchSize x sizePredictionScores, and
# "labels" is a vector of size batchSize.
class toynn_CrossEntropyLoss(object):
    # Forward pass: -log softmax(input_{label})
    def forward(self, scores, labels):
        # 1. Computing the softmax: exp(x) / sum (exp(x))
        max_val = scores.max()  # This is to avoid variable overflows.
        exp_inputs = (scores - max_val).exp()
        # This is different than in the previous lab. Avoiding for loops here.
        denominators = exp_inputs.sum(1).repeat(scores.size(1), 1).t()
        self.predictions = torch.mul(exp_inputs, 1 / denominators)

        # 2. Computing the loss: -log(y_label).
        # Check what gather does. Just avoiding another for loop here.
        return -self.predictions.log().gather(1, labels.view(-1, 1)).mean()

    # Backward pass: y_hat - y
    def backward(self, scores, labels):
        # Here we avoid computing softmax again in backward pass.
        grad_inputs = self.predictions.clone()

        # Ok, Here we will use a for loop (but it is avoidable too).
        for i in range(0, scores.size(0)):
            grad_inputs[i][labels[i]] = grad_inputs[i][labels[i]] - 1
        return grad_inputs
class toynn_Linear(object):
    def __init__(self, numInputs, numOutputs):
        # Allocate tensors for the weight and bias parameters.
        self.weight = torch.Tensor(numInputs, numOutputs).normal_(0, 0.01)
        self.weight_grads = torch.Tensor(numInputs, numOutputs)
        self.bias = torch.Tensor(numOutputs).zero_()
        self.bias_grads = torch.Tensor(numOutputs)

        # Forward pass, inputs is a matrix of size batchSize x numInputs.
        # Notice that compared to the previous assignment, each input vector
        # is a row in this matrix.
        def forward(self, inputs):
            # This one needs no change, it just becomes
            # a matrix x matrix multiplication
            # as opposed to just vector x matrix multiplication as we had before.
            return torch.matmul(inputs, self.weight) + self.bias

        # Backward pass, in addition to compute gradients for the weight and bias.
        # It has to compute gradients with respect to inputs.
        def backward(self, inputs, scores_grads):
            self.weight_grads = torch.matmul(inputs.t(), scores_grads)
            self.bias_grads = scores_grads.sum(0)
            return torch.matmul(scores_grads, self.weight.t())
```python
class toynn_ReLU(object):
    # Forward operation: f(x_i) = max(0, x_i)
def forward(self, inputs):
    outputs = inputs.clone()
    outputs[outputs < 0] = 0
    return outputs

    # Make sure the backward pass is absolutely clear.
def backward(self, inputs, outputs_grad):
    inputs_grad = outputs_grad.clone()  # 1 * previous_grads
    inputs_grad[inputs < 0] = 0         # or zero.
    return inputs_grad
```
Two-layer Neural Network – Forward Pass

```python
# Setup the input variable x.
img, label = trainset[0]
x = img.view(1, 1 * 28 * 28)

# Setup the number of inputs, hidden neurons, and outputs.
nInputs = 1 * 28 * 28
nHidden = 512
nOutputs = 10

# Create the model here.
linear_fn1 = toynn_Linear(nInputs, nHidden)
relu_fn = toynn_ReLU()
linear_fn2 = toynn_Linear(nHidden, nOutputs)

# Make predictions.
x = linear_fn1.forward(x)
x = relu_fn.forward(x)
x = linear_fn2.forward(x)

# Show the prediction scores for each class.
# Yes, pytorch tensors already come with a softmax function.
# We need it here because we hard-coded the softmax inside
# the loss function.
print(x.softmax(dim = 1))
```
Two-layer Neural Network – Backward Pass

```python
# Create the model here.
linear_fn1 = toynn_Linear(nInputs, nHidden)
relu_fn = toynn_ReLU()
linear_fn2 = toynn_Linear(nHidden, nOutputs)
loss_fn = toynn_CrossEntropyLoss()

# Make predictions (forward pass).
a = linear_fn1.forward(x)
z = relu_fn.forward(a)
yhat = linear_fn2.forward(z)

# Compute loss.
loss = loss_fn.forward(yhat, label)
yhat_grads = loss_fn.backward(yhat, label)

# Compute gradients (backward pass).
z_grads = linear_fn2.backward(z, yhat_grads)
a_grads = relu_fn.backward(a, z_grads)
x_grads = linear_fn1.backward(x, a_grads)

# Update parameters:
learningRate = 0.2
linear_fn1.weight.add_(-learningRate, linear_fn1.weight_grads)
linear_fn1.bias.add_(-learningRate, linear_fn1.bias_grads)
linear_fn2.weight.add_(-learningRate, linear_fn2.weight_grads)
linear_fn2.bias.add_(-learningRate, linear_fn2.bias_grads)
```
Convolutional Layer

Input image: 

$$
\begin{array}{cccccc}
4 & 5 & 7 & 6 & 6 \\
3 & 2 & 8 & 0 & 7 \\
6 & 7 & 7 & 1 & 5 \\
3 & 0 & 1 & 1 & 1 \\
4 & 3 & 2 & 1 & 7 \\
\end{array}
$$

Weights: 

$$
\begin{array}{ccc}
0 & 0 & 0 \\
1 & 0 & 1 \\
0 & 0 & 0 \\
\end{array}
$$

Output image: 

$$
\begin{array}{ccc}
11 & 2 & 15 \\
13 & 8 & 12 \\
\end{array}
$$
Convolutional Layer

Weights
Convolutional Layer

Weights
Convolutional Layer
Convolutional Layer (with 4 filters)

Input: 1x224x224  weights: 4x1x9x9  Output: 4x224x224
if zero padding, and stride = 1
Convolutional Layer (with 4 filters)

Input: 1x224x224

weights: 4x1x9x9

Output: 4x112x112

if zero padding, but stride = 2
Convolutional Layer in Torch

SpatialConvolution

module = nn.SpatialConvolution(nInputPlane, nOutputPlane, kW, kH, [dW], [dH], [padW], [padH])

Input

nInputPlane (e.g. 3 for RGB inputs)

Output

nOutputPlane (equals the number of convolutional filters for this layer)
Convolutional Layer in Keras

Convolution2D(nOutputPlane, kW, kH, input_shape = (3, 224, 224), subsample = 2, border_mode = valid)
Convolutional Layer in pytorch

class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True) [source]
Automatic Differentiation

You only need to write code for the forward pass, backward pass is computed automatically.

Pytorch (Facebook -- mostly): https://pytorch.org/
Tensorflow (Google -- mostly): https://www.tensorflow.org/
DyNet (team includes UVA Prof. Yangfeng Ji): http://dynet.io/
Questions?