MathDNN Homework 6

Department of Computer Science and Engineering
2021-16988 Jaewan Park

Problem 1

Let A_i, b_i the parameters for the linear relationship between layers y_i and y_{i+1}. ($A_i \in \mathbb{R}^{m\times n}$, $b_i \in \mathbb{R}^m$, $y_i \in \mathbb{R}^n$, $y_{i+1} \in \mathbb{R}^m$) Then applying dropout before the activation function results in

$$y_{i+1} = \sigma(A_i y_i' + b_i), \quad [y_i']_j = \begin{cases} 0 & \text{(with probability p)} \\ \dfrac{[y_i]_j}{1-p} & \text{(otherwise)} \end{cases}$$

while the opposite gives

$$[y_{i+1}]_j = \begin{cases} 0 & \text{(with probability p)} \\ \dfrac{[\sigma(A_i y_i + b_i)]_j}{1-p} & \text{(otherwise)} \end{cases}.$$

For notational convenience, denote the resulting y_{i+1} from each cases y_{i+1}^1 and y_{i+1}^2. We should find the appropriate σ where $\mathbf{E}\Big[\big(y_{i+1}^1\big)_j\Big] = \mathbf{E}\Big[\big(y_{i+1}^2\big)_j\Big]$ for $j = 1, 2, \cdots, m$. We can generally obtain the following.

$$\begin{aligned}
\mathbf{E}\Big[\big[y_{i+1}^1\big]_j\Big] &= \mathbf{E}\Big[\big[\sigma(A_i y_i' + b_i)\big]_j\Big] = \mathbf{E}\Big[\sigma\Big(\big[A_i y_i' + b_i\big]_j\Big)\Big] \\
&= \mathbf{E}\Big[\sigma\Big(\big[A_i\big]_{j,:} \cdot y_i' + \big[b_i\big]_j\Big)\Big] \\
&= \mathbf{E}\Big[\sigma\Big(\big[A_i\big]_{j,1}[y_i']_1 + \cdots + \big[A_i\big]_{j,n}[y_i']_n + \big[b_i\big]_j\Big)\Big] \\
&= \mathbf{E}\Big[\sigma\Big(\big[A_i\big]_{j,1}[y_i']_1\Big)\Big] + \cdots + \mathbf{E}\Big[\sigma\Big(\big[A_i\big]_{j,n}[y_i']_n\Big)\Big] + \mathbf{E}\Big[\sigma\Big(\big[b_i\big]_j\Big)\Big] \\
&= \sigma\left(\big[A_i\big]_{j,1}\frac{[y_i]_1}{1-p}\right) \times (1-p) + \cdots + \sigma\left(\big[A_i\big]_{j,n}\frac{[y_i]_n}{1-p}\right) \times (1-p) + \sigma\Big(\big[b_i\big]_j\Big) \\
\mathbf{E}\Big[\big[y_{i+1}^2\big]_j\Big] &= \frac{[\sigma(A_i y_i + b_i)]_j}{1-p} \times (1-p) \\
&= \sigma\Big(\big[A_i\big]_{j,1}[y_i]_1\Big) + \cdots + \sigma\Big(\big[A_i\big]_{j,n}[y_i]_n\Big) + \sigma\Big(\big[b_i\big]_j\Big)
\end{aligned}$$

For the three cases, results differ as the following.

(1) `nn.ReLU()`

For $a > 0$, $\mathrm{ReLU}(ax) = a\mathrm{ReLU}(x)$ from the definition of ReLU. Therefore the two results are equivalent.

$$\begin{aligned}
\mathbf{E}\Big[\big[y_{i+1}^1\big]_j\Big] &= \sigma\left(\big[A_i\big]_{j,1}\frac{[y_i]_1}{1-p}\right) \times (1-p) + \cdots + \sigma\left(\big[A_i\big]_{j,n}\frac{[y_i]_n}{1-p}\right) \times (1-p) + \sigma\Big(\big[b_i\big]_j\Big) \\
&= \sigma\Big(\big[A_i\big]_{j,1}[y_i]_1\Big) + \cdots + \sigma\Big(\big[A_i\big]_{j,n}[y_i]_n\Big) + \sigma\Big(\big[b_i\big]_j\Big) \\
&= \mathbf{E}\Big[\big[y_{i+1}^2\big]_j\Big]
\end{aligned}$$

(2) `nn.Sigmoid()`

Generally, $\sigma\left([A_i]_{j,k}\dfrac{[y_i]_k}{1-p}\right)(1-p) = \sigma\left([A_i]_{j,1}[y_i]_1\right)$ does not stand for the sigmoid function. Therefore changing the order gives nonequivalent results.

(3) `nn.LeakyReLU()`

For $a > 0$, $\text{LeakyReLU}(ax) = a\text{LeakyReLU}(x)$ from the definition of leaky ReLU. Therefore similar to the ReLU function, changing the order gives equivalent results.

Therefore in cases of ReLU and leaky ReLU, the order of dropout and activation functions does not matter.

Problem 2

The weight initializing code of PyTorch's `nn.Linear` model is given as the following.

```
def reset_parameters(self) -> None:
    # Setting a=sqrt(5) in kaiming_uniform is the same as initializing with
    # uniform(-1/sqrt(in_features), 1/sqrt(in_features)). For details, see
    # https://github.com/pytorch/pytorch/issues/57109
    init.kaiming_uniform_(self.weight, a=math.sqrt(5))
    if self.bias is not None:
        fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        init.uniform_(self.bias, -bound, bound)
```

The weight and bias elements are all initialized from $\mathcal{U}(-\texttt{bound}, \texttt{bound})$ where $\texttt{bound} = \dfrac{1}{\sqrt{\texttt{fan\_in}}}$. In this case, $\texttt{fan\_in} = n_\ell$ for each step. Therefore all elements of $[A_k]$ and $[b_k]$ follow $\mathcal{U}\left(-\dfrac{1}{\sqrt{n_{k-1}}}, \dfrac{1}{\sqrt{n_{k-1}}}\right)$, and have mean 0, variance $\dfrac{1}{3n_{k-1}}$.

For $j = 1, 2, \cdots, n_0$, $\mathbf{E}\left[[x]_j\right] = 0$ and $\mathbf{Var}\left[[x]_j\right] = 1$. Since all variables are uncorrelated, for $j = 1, 2, \cdots, n_1$,

$$\begin{aligned}
\mathbf{E}\left[[y_1]_j\right] &= \mathbf{E}\left[[A_1]_{j,1}[x]_1 + \cdots + [A_1]_{j,n_0}[x]_{n_0} + [b_1]_j\right] \\
&= \mathbf{E}\left[[A_1]_{j,1}\right]\mathbf{E}[[x]_1] + \cdots + \mathbf{E}\left[[A_1]_{j,n_0}\right]\mathbf{E}\left[[x]_{n_0}\right] + \mathbf{E}\left[[b_1]_j\right] \\
&= 0 \\
\mathbf{Var}\left[[y_1]_j\right] &= \mathbf{Var}\left[[A_1]_{j,1}[x]_1 + \cdots + [A_1]_{j,n_0}[x]_{n_0} + [b_1]_j\right] \\
&= \mathbf{Var}\left[[A_1]_{j,1}\right]\mathbf{Var}[[x]_1] + \cdots + \mathbf{Var}\left[[A_1]_{j,n_0}\right]\mathbf{Var}\left[[x]_{n_0}\right] + \mathbf{Var}\left[[b_1]_j\right] \\
&= \frac{n_0+1}{3n_0}.
\end{aligned}$$

Let $y_0 = x$. Then generally, we can inductively show that for $l = 1, 2, \cdots, L$, all elements of y_ℓ have the

same mean and variance as the following.

$$\begin{aligned}
\mathbf{E}\Big[[y_\ell]_j\Big] &= \mathbf{E}\Big[[A_\ell]_{j,1}\Big]\mathbf{E}[[y_{\ell-1}]_1] + \cdots + \mathbf{E}\Big[[A_\ell]_{j,n_{\ell-1}}\Big]\mathbf{E}\Big[[y_{\ell-1}]_{n_{\ell-1}}\Big] + \mathbf{E}\Big[[b_\ell]_j\Big] \\
&= 0 \\
\mathbf{Var}\Big[[y_\ell]_j\Big] &= \mathbf{Var}\Big[[A_\ell]_{j,1}\Big]\mathbf{Var}[[y_{\ell-1}]_1] + \cdots + \mathbf{Var}\Big[[A_\ell]_{j,n_{\ell-1}}\Big]\mathbf{Var}\Big[[y_{\ell-1}]_{n_{\ell-1}}\Big] + \mathbf{Var}\Big[[b_\ell]_j\Big] \\
&= \frac{\mathbf{Var}[[y_{\ell-1}]_1]\cdot n_{\ell-1}+1}{3n_{\ell-1}}
\end{aligned}$$

The variance cannot be further simplified, but if the bias is 0, we can simply say $\mathbf{Var}\Big[[y_\ell]_j\Big] = \frac{1}{3^\ell}$.

Problem 3

Notation For a matrix or vector X, the notation $[X]_{i,j}$ or $[X]_i$ refers to the element of X at that index, and $\{f(i,j)\}_{i,j}$ refers to a matrix of which element at (i,j) is $f(i,j)$.

(1) For $\ell = L$,

$$\frac{\partial y_\ell}{\partial y_{\ell-1}} = \frac{\partial y_L}{\partial y_{L-1}} = A_L.$$

For $\ell = 2, \cdots, L-1$,

$$\begin{aligned}
\frac{\partial y_\ell}{\partial y_{\ell-1}} &= \frac{\partial}{\partial y_{\ell-1}}(\sigma(A_\ell y_{\ell-1} + b_\ell) + y_{\ell-1}) \\
&= \left\{ \frac{\partial}{\partial [y_{\ell-1}]_j}\Big[\sigma(A_\ell y_{\ell-1} + b_\ell) + y_{\ell-1}\Big]_i \right\}_{i,j} \\
&= \left\{ \frac{\partial}{\partial [y_{\ell-1}]_j}\Big(\sigma([A_\ell y_{\ell-1}]_i + [b_\ell]_i) + [y_{\ell-1}]_i\Big) \right\}_{i,j} \\
&= \left\{ \sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \cdot \frac{\partial}{\partial [y_{\ell-1}]_j}([A_\ell y_{\ell-1}]_i + [b_\ell]_i) + 1 \right\}_{i,j} \\
&= \left\{ \sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \cdot \frac{\partial}{\partial [y_{\ell-1}]_j}\left(\sum_{k=1}^{n_{\ell-1}} [A_\ell]_{i,k}[y_{\ell-1}]_k + [b_\ell]_i\right) \right\}_{i,j} \\
&= \Big\{ \sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \cdot [A_\ell]_{i,j} + 1 \Big\}_{i,j} \\
&= \text{diag}\Big(\sigma'(A_\ell y_{\ell-1} + b_\ell)\Big) A_\ell + \{1\}_{i,j}.
\end{aligned}$$

(2) For $\ell = L$,

$$\frac{\partial y_L}{\partial b_\ell} = \frac{\partial y_L}{\partial b_L} = \frac{\partial}{\partial b_L}(A_L y_{L-1} + b_L) = 1.$$

For $\ell = 1, \cdots, L-1$, we can obtain the following.

$$\begin{aligned}
\frac{\partial y_\ell}{\partial b_\ell} &= \frac{\partial}{\partial b_\ell}(\sigma(A_\ell y_{\ell-1} + b_\ell) + y_{\ell-1}) = \frac{\partial}{\partial b_\ell}\sigma(A_\ell y_{\ell-1} + b_\ell) \\
&= \left\{ \frac{\partial}{\partial [b_\ell]_j}[\sigma(A_\ell y_{\ell-1} + b_\ell)]_i \right\}_{i,j} = \left\{ \frac{\partial}{\partial [b_\ell]_j}\sigma([A_\ell y_{\ell-1} + b_\ell]_i) \right\}_{i,j} = \left\{ \frac{\partial}{\partial [b_\ell]_j}\sigma([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \right\}_{i,j} \\
&= \left\{ \sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \cdot \frac{\partial}{\partial [b_\ell]_j}([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \right\}_{i,j} \\
&= \{\sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i) \cdot \delta_{ij}\}_{i,j} \quad (\delta_{ij}\text{: Kronecker Delta}) \\
&= \text{diag}\left(\sigma'(A_\ell y_{\ell-1} + b_\ell)\right)
\end{aligned}$$

Therefore

$$\begin{aligned}
\frac{\partial y_L}{\partial b_\ell} &= \frac{\partial y_L}{\partial y_{L-1}}\frac{\partial y_{L-1}}{\partial y_{L-2}} \cdots \frac{\partial y_{\ell+1}}{\partial y_\ell}\frac{\partial y_\ell}{\partial b_\ell} \\
&= A_L\Big(\text{diag}\left(\sigma'(A_{L-1}y_{L-2} + b_{L-1})\right)A_{L-1} + \{1\}_{i,j}\Big) \\
&\qquad \cdots \Big(\text{diag}\left(\sigma'(A_{\ell+1}y_\ell + b_{\ell+1})\right)A_{\ell+1} + \{1\}_{i,j}\Big)\text{diag}(\sigma'(A_\ell y_{\ell-1} + b_\ell)).
\end{aligned}$$

For $\ell = L$,

$$\frac{\partial y_L}{\partial A_\ell} = \frac{\partial y_L}{\partial A_L} = \frac{\partial}{\partial A_L}(A_L y_{L-1} + b_L) = y_{L-1}.$$

For $\ell = 1, \cdots, L-1$, we can obtain the following.

$$\begin{aligned}
\left[\frac{\partial y_L}{\partial A_\ell}\right]_{i,j} &= \frac{\partial y_L}{\partial [A_\ell]_{i,j}} = \frac{\partial y_L}{\partial y_\ell}\frac{\partial y_\ell}{\partial [A_\ell]_{i,j}} = \frac{\partial y_L}{\partial y_\ell}\left\{\frac{\partial [y_\ell]_k}{\partial [A_\ell]_{i,j}}\right\}_k \\
&= \frac{\partial y_L}{\partial y_\ell}\left\{\frac{\partial}{\partial [A_\ell]_{i,j}}\Big(\sigma([A_\ell y_{\ell-1}]_k + [b_\ell]_k) + [y_{\ell-1}]_k\Big)\right\}_k = \frac{\partial y_L}{\partial y_\ell}\left\{\frac{\partial}{\partial [A_\ell]_{i,j}}\sigma([A_\ell y_{\ell-1}]_k + [b_\ell]_k)\right\}_k \\
&= \frac{\partial y_L}{\partial y_\ell}\left\{\sigma'([A_\ell y_{\ell-1}]_k + [b_\ell]_k)\left(\frac{\partial}{\partial [A_\ell]_{i,j}}([A_\ell y_{\ell-1}]_k + [b_\ell]_k)\right)\right\}_k \\
&= \frac{\partial y_L}{\partial y_\ell}\begin{bmatrix} 0 \\ \vdots \\ \sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i)[y_{\ell-1}]_j \\ \vdots \\ 0 \end{bmatrix} \quad (\text{All elements except the } i\text{-th element are } 0.) \\
&= \left[\frac{\partial y_L}{\partial y_\ell}\right]_i \sigma'([A_\ell y_{\ell-1}]_i + [b_\ell]_i)[y_{\ell-1}]_j \\
&= [\sigma'(A_\ell y_{\ell-1} + b_\ell)]_i \left[\frac{\partial y_L}{\partial y_\ell}\right]_i [y_{\ell-1}]_j
\end{aligned}$$

Therefore

$$\begin{aligned}\frac{\partial y_L}{\partial A_\ell} &= \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1}+b_\ell)\right)\left(\frac{\partial y_L}{\partial y_\ell}\right)^{\intercal} y_{\ell-1}^{\intercal} \\ &= \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1}+b_\ell)\right)\left(\frac{\partial y_L}{\partial y_{L-1}}\right)^{\intercal}\left(\frac{\partial y_{L-1}}{\partial y_{L-2}}\right)^{\intercal}\cdots\left(\frac{\partial y_{\ell+1}}{\partial y_\ell}\right)^{\intercal} y_{\ell-1}^{\intercal} \\ &= \operatorname{diag}\left(\sigma'(A_\ell y_{\ell-1}+b_\ell)\right)A_L^{\intercal}\Big(\operatorname{diag}\Big(\sigma'(A_{L-1}y_{L-2}+b_{L-1})\Big)A_{L-1}+\{1\}_{i,j}\Big)^{\intercal} \\ &\qquad\qquad \cdots\Big(\operatorname{diag}\Big(\sigma'(A_{\ell+1}y_\ell+b_{\ell+1})\Big)A_{\ell+1}+\{1\}_{i,j}\Big)^{\intercal} y_{\ell-1}^{\intercal}.\end{aligned}$$

(3) For $i \le \ell$,

$$\begin{aligned}\frac{\partial y_L}{\partial b_i} &= A_L\Big(\operatorname{diag}\Big(\sigma'(A_{L-1}y_{L-2}+b_{L-1})\Big)A_{L-1}+\{1\}_{i,j}\Big) \\ &\qquad\qquad \cdots\Big(\operatorname{diag}\Big(\sigma'(A_{i+1}y_i+b_{i+1})\Big)A_{i+1}+\{1\}_{i,j}\Big)\operatorname{diag}(\sigma'(A_i y_{i-1}+b_i)) \\ \frac{\partial y_L}{\partial A_i} &= \operatorname{diag}\left(\sigma'(A_i y_{i-1}+b_i)\right)A_L^{\intercal}\Big(\operatorname{diag}\Big(\sigma'(A_{L-1}y_{L-2}+b_{L-1})\Big)A_{L-1}+\{1\}_{i,j}\Big)^{\intercal} \\ &\qquad\qquad \cdots\Big(\operatorname{diag}\Big(\sigma'(A_{i+1}y_i+b_{i+1})\Big)A_{i+1}+\{1\}_{i,j}\Big)^{\intercal} y_{i-1}^{\intercal}.\end{aligned}$$

The given conditions guarantee that for any ℓ and $i \le \ell$, at least one of $A_{L-1}, \cdots, A_{i+1}$ and one of $\operatorname{diag}\Big(\sigma'(A_{L-1}y_{L-2}+b_{L-1})\Big), \cdots, \operatorname{diag}\Big(\sigma'(A_{i+1}y_i+b_{i+1})\Big)$ is 0. However, due to the $\{1\}_{i,j}$ term in each calculation, the gradients do not vanish, rather remain in large values.

Problem 4

(1) The number of trainable parameters between two layers can be calculated as $C_{\text{out}} \times (C_{\text{in}} \times F \times F + 1)$, where F is the size of the convolution filter and C_{in} and C_{out} are each the number of channels for input and output. The addition of 1 is made due to the bias. In the original construction, the total number of parameters is

$$128 \times (256 \times 1 \times 1 + 1) + 128 \times (128 \times 3 \times 3 + 1) + 256 \times (128 \times 1 \times 1 + 1) = 213504.$$

In the modified construction, the total number of parameters is

$$\Big(4 \times (256 \times 1 \times 1 + 1)\Big) \times 32 + \Big(4 \times (4 \times 3 \times 3 + 1)\Big) \times 32 + \Big(256 \times (4 \times 1 \times 1 + 1)\Big) \times 32 = 78592.$$

Therfore the modified version reduces the number of trainable parameters considerably.

(2) The convolution model can be implemented via PyTorch as the following. (The final summation is done for 32 objects, thus for convenience the repetition is abbreviated as '...' in the code.)

```
class STMConvLayer(nn.Module):
    def __init__(self):
        super(STMConvLayer , self).__init__()
        self.conv1layers = [nn.Conv2d(256, 4, 1) for _ in range(32)]
        self.conv2layers = [nn.Conv2d(4, 4, 3) for _ in range(32)]
        self.conv3layers = [nn.Conv2d(4, 256, 1) for _ in range(32)]
    def forward(self, x):
        # [apply 1x1conv with 4 output channels
        #  apply 3x3conv with 4 output channels (with padding=1)
        #  apply 1x1conv with 256 output channels] X 32
        # Add all 32 outputs
        out_list = [nn.ReLU(self.conv1layers[i](x)) for i in range(32)]
        out_list = [nn.ReLU(self.conv2layers[i](out_list[i])) for i in range(32)]
        out_list = [nn.ReLU(self.conv3layers[i](out_list[i])) for i in range(32)]
        out = out_list[0] + out_list[1] + ... + out_list[31]
        return out
```

Problem 5

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
"""
Step 1 : Generate Toy data
"""

d = 35
n_train, n_val, n_test = 300, 60, 30
np.random.seed(0)
beta = np.random.randn(d)
beta_true = beta / np.linalg.norm(beta)
# Generate and fix training data
X_train = np.array([np.random.multivariate_normal(np.zeros(d), np.identity(d)) for _ in␣
↪range(n_train)])
Y_train = X_train @ beta_true + np.random.normal(loc = 0.0, scale = 0.5, size =␣
↪n_train)
# Generate and fix validation data (for tuning lambda).
X_val = np.array([np.random.multivariate_normal(np.zeros(d), np.identity(d)) for _ in␣
↪range(n_val)])
Y_val = X_val @ beta_true
# Generate and fix test data
X_test = np.array([np.random.multivariate_normal(np.zeros(d), np.identity(d)) for _ in␣
↪range(n_test)])
Y_test = X_test @ beta_true
```

[3]:
```
"""
Step 2 : Solve the problem
"""

def solve_lsqr(X, y, lamda):
    # return  np.linalg.inv(np.dot(X.T, X) + lamda * np.identity(X.shape[1])) @ X.T @ y␣
↪
    # using backsolve is faster than computing explicit inverse
    return  np.linalg.solve(np.dot(X.T, X) + lamda * np.identity(X.shape[1]), X.T @ y)

def get_error(tilde_x, y, theta) :
    """
    Calculate £RMSE£ for given test dataset (tilde_x, y) and infered theta vector.
    """
    return np.linalg.norm(tilde_x @ theta - y)


def ReLU(x):
    """
    Custom ReLU function that is applied elementwisely.
    """
    return np.where(np.asarray(x) > 0, x, 0)

lambda_list = [2 ** i for i in range(-6, 6)]
num_params = np.arange(1,1501,10)
```

```
li = 10
prev_error_val = 10
grad = 1
errors_opt_lambda = []
errors_fixed_lambda = []
for p in num_params :
    weight_matrix = np.random.normal(loc = 0.0, scale = 1 / np.sqrt(p), size = (p, d))
    x_val_tilde = ReLU(X_val @ weight_matrix.T)
    theta_val = solve_lsqr(x_val_tilde, Y_val, lambda_list[li])
    error_val = get_error(x_val_tilde, Y_val, theta_val)
    li = li + int((error_val - prev_error_val) * grad)
    if li < 0 : li = 0
    if li >= len(lambda_list) : li = len(lambda_list) - 1
    grad = 1 if error_val - prev_error_val > 0 else -1
    prev_error_val = error_val

    weight_matrix = np.random.normal(loc = 0.0, scale = 1 / np.sqrt(p), size = (p, d))
    x_tilde = ReLU(X_train @ weight_matrix.T)
    theta = solve_lsqr(x_tilde, Y_train, lambda_list[li])
    X_test_tilde = ReLU(X_test @ weight_matrix.T)
    error = get_error(X_test_tilde, Y_test, theta)
    errors_opt_lambda.append(error)

    weight_matrix = np.random.normal(loc = 0.0, scale = 1 / np.sqrt(p), size = (p, d))
    x_tilde = ReLU(X_train @ weight_matrix.T)
    theta = solve_lsqr(x_tilde, Y_train, 0.01)
    X_test_tilde = ReLU(X_test @ weight_matrix.T)
    error = get_error(X_test_tilde, Y_test, theta)
    errors_fixed_lambda.append(error)
```

[4]:
```
"""
Step 3 : Plot the results
"""

plt.figure(figsize = (24, 8))
plt.rc('text', usetex = True)
plt.rc('font', family = 'serif')
plt.rc('font', size = 24)


plt.scatter(num_params, errors_fixed_lambda, color = 'black',
            label = r"Test error with fixed $\lambda = 0.01$",
            )
plt.legend()

plt.plot(num_params, errors_opt_lambda, 'k', label = r"Test error with tuned␣
 ↪$\lambda$")
plt.legend()
plt.xlabel(r'$\#$ parameters')
plt.ylabel('Test error')
plt.title(r'Test error vs. $\#$ params')
```

```
# plt.savefig('double_descent.png')
plt.show()
```